PROJECT REPORT

Frequent Itemset Mining Algorithms for Big Data

Ashish Gupta (16419CMP008)

Smriti (16419CMP026)

Ankit Yadav (16419CMP004)

Department of Computer Science

Institute of Science, Banaras Hindu University (BHU)

Varanasi-221005, U.P., India

Frequent Itemset Mining Algorithms for Big Data

Mentor

By:

Dr Sudhakar Singh

Department of Computer Science

Institute of Science, Banaras Hindu University (BHU)

A mini project report is submitted in Department of Computer Science of the Institute of Science of Banaras Hindu University (BHU) in partial fulfilment of the requirements for the degree of Master of Science in Computer Science.

Varanasi, India

Jan 08 2018

Approved by

Mentor:

Dr Sudhakar Singh

External Examiner:

ABSTRACT

Very huge quantity of Big Data from variety of different sources such as IT industries, internet applications, hospital history records, microphones, sensor network, social media feeds etc. is continuously generated and with rapid speed. Frequent pattern mining is an essential data mining task, with a goal of discovering knowledge in the form of repeated patterns. Many efficient single-node pattern mining algorithms have been discovered in the last two decades such as the well-known FP growth algorithm but these algorithms are inefficient on large scale datasets. Multi-node pattern mining algorithms have been developed, exploiting the advantages of distributed computing frameworks, such as Apache Hadoop and Spark. Yet most do not scale to the type of data we are presented with today. In this project, we present a review on existing parallel versions of FP-Growth Algorithm and implement a parallel version of FP-Growth based on MapReduce framework using Apache Spark.

Keywords*: Big data; Data mining; Frequent Itemset Mining; Apache Hadoop; Apache Spark; MapReduce*

ACKNOWLEDGEMENTS

The satisfaction that accompanies the successful completion of this project would be in complete without the mention of the people who made it possible. We would like to express our deepest appreciation to all those who provided us the possibility to complete this report. We convey thanks to our project guide Dr Sudhakar Singh of Department of Computer Science, Banaras Hindu University for providing encouragement, constant support and guidance which was of a great help to complete this project successfully. We would also like to thank our Head of the Department Prof. S. Karthikeyan and committee Coordinator Dr Manoj Kumar Singh for their valuable support throughout the project.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTODUCTION

Due to growth of IT industries, services, technologies and data, the huge amount of complex data is generated from the various sources that can be in various form. Such complex and massive data is difficult to handle and process that contain the billion records of million user and product information that includes the online selling data, audios, images, videos of social media, news feeds, product price and specification etc. Big data analytics analyse the huge amount of data by different mining algorithms and reveals the hidden patterns, trends and the other meaningful information.

Frequent itemsets play an essential role in many data mining tasks that try to find interesting patterns from databases. Association rules describe how often items are purchased together. There are two steps in mining association rules. First step is to find all frequent itemsets, and the second step is to generate the association rules from each frequent itemsets. In the beginning, there are many number of Frequent itemset mining algorithms. But unfortunately, the cost of computation and space is expensive when the size of the data is large. Parallel FP-Growth algorithm on distributed machines reduces the cost of computation and space. The main challenge is devising a smart partitioning of the problem in independent subproblems, each one based on a subset of the data, to exploit the computation power of a cluster of servers in parallel. But, it causes very high I/O overhead for iterative computations. A new parallel version of FP-Growth algorithm [6], S-FPG (Spark FP-Growth) [4] using Apache Spark that is an in-memory-based and iterative computing framework. It requires a cluster manager and a distributed storage system.

The rest of the dissertation is organized as follows. Section 1.2 and 1.3 discuss about motivation and literature review respectively. Section 2 deals with the background methods to solve the problem. Section 3 describes program details, source code and formulae used in the program. Section 4 discusses about the result and conclusion. The performance and efficiency of the algorithm.

1.1. MOTIVATION

Since the introduction of association rule mining in 1993 by Agrawal Imielinski and Swami [14], the frequent itemset mining has been the most buzzed topic and a topic for intense research in past few years. Fast and Efficient algorithm are being designed to mine frequent itemset from large datasets, every new paper claims to run faster than previously existing algorithms, based on their experimental testing.

FP-Growth [6] method is efficient and scalable for mining long and short frequent patterns, unfortunately when the dataset is large, the memory use and computational cost increase exponentially. So, there is a need to parallelize the existing algorithm using some distributed computing framework.

1.2. LITERATURE REVIEW

Some previous efforts [2] [5] [6] [10] for Frequent Itemset Mining (FIM) [6] which is an essential data mining task, with many real-world applications such as market basket analysis, outlier direction, etc. have been done. Big Data do not refer to the data only in size. It is extravagant amount of uncertainty data containing different formats from different sources with rapid speed [1]. Generating frequent patterns in Big Data and other database in field of data mining is done by using Apriori algorithm by candidate set generation. When there is large number of patterns, it is very costly to generate frequent patterns by this method [6]. Frequent Pattern tree mine the complete set of frequent patterns and generate an efficient FP-tree and gives hidden information. But unfortunately, the cost of computation and space is expensive when the size of the data is large.

Hadoop has been developed for processing large and extravagant data in distributed and parallel fashion [1]. It handles fault tolerance, data distribution, parallelization and load balancing task. There are number of sub-project that provides specific services and work on top of Hadoop such as: Mahout, HBase, Hive, etc. Apache is not only organization that develop tools and project for Big Data such as: Cloudera, HortonWork, MapR, etc. Hadoop can also be set up and configured in cloud and virtualization infrastructures [1]. PFP-Growth algorithm on distributed machines divides computation that each machine computes a single independent group of data. By dividing, it removes the computational dependencies [8]. MapReduce parallel programming framework provides faster idea for handling Big Data but it causes very high I/O overhead for iterative computations because it is a disk-based model.

To overcome, a new parallel version of FP-Growth algorithm, S-FPG (for spark FP-Growth) using Apache Spark. Apache Spark requires a cluster manger such as: Hadoop YARN or Apache Mesos and a distributed storage system such as: HDFS or Amazon S3. S-FPG algorithm can scale well and efficiently process large datasets [4].

1.3. KEY CONTRIBUTION

The contributions are the following.

- Literature review on frequent itemset mining algorithms on Hadoop and Spark.
- We implemented the parallel projection of FP Growth method as described by author in [6].
- An experimental analysis of the FP-Growth has been carried out to address the itemset mining problem in the Big Data context by means of Apache Spark, with the analysis of their expected impact on main memory usage.
- An extensive evaluation campaign to assess the reliability of our expectations. Precisely, we ran more than 20 experiments on 2 synthetic datasets and 3 real datasets to evaluate the execution time of parallel itemset mining implementations.
- The identification of strengths and weaknesses of the algorithm with respect to the input dataset features (e.g., data distribution, average transaction length, number of records), and specific parameter settings.
- The discussion of promising open research directions for the parallelization of the itemset mining problem, to be carried out in major project.

2. BACKGROUND METHODS TO SOLVE THE PROBLEM

2.1. ASSOCIATION RULE MINING

Data gathered from a variety of data sources are often a series of isolated data, correlation analysis naturally becomes an important foundation for data mining and big data science. Association rule mining [14] was proposed to discover certain interesting correlation relationships among the item sets of the data.

An Association rule defines relation between two sets of items for e.g. {Diapers}->{Beer}. The rule suggests that a strong relationship exists between the sale of diapers and sale of beers because many customers who buy diapers also buy beer.

Association analysis is not only limited to market basket data, association analysis is also applicable to application domains such as bioinformatics, medical diagnosis, web mining and scientific data analysis.

Mining association rule consists of following two steps:

- Frequent Itemset Generation: The frequent item sets are set of those items whose support (sup (item)) in the data set is greater than the minimum required support (min_sup). Considering the above example all two diapers, beer belongs to frequent itemset and sup {diapers} and sup {beer} would be greater than the min_sup. The support of an itemset is defined as proportion of transactions which contains the itemset.
- Rule Generation: Generating the interesting rules from the frequent itemsets on the basis of confidence (conf). The confidence of the above rule will be sup {diapers} divided by sup{beer}. If the confidence of the rule is greater than the required confidence, the rule can be considered as an interesting one. For a given rule X->Y, the higher the confidence the more likely it is for Y to be present in transactions that contain X.

Finding frequent itemset from a large transactional database described above can be computationally expensive. A dataset containing k items can generate up to 2^k-1 frequent itemsets.

There are several algorithms for mining frequent itemset efficiently by reducing the computational complexity of frequent itemset generation.

1. Apriori Algorithm [13]
2. FP Growth algorithm [6]

3. Tree Projection Algorithm [17]
4. Eclat Algorithm [16]

Eclat Algorithm: It performs mining from vertical transposition of the dataset [2].

FP-Growth algorithm is the most efficient algorithm [6] among the first three above-mentioned algorithms. Therefore, in this work we focus on FP Growth Algorithm

2.2. FP-GROWTH ALGORITHM

The frequent itemset required for generation of association rule can be generated by using this algorithm. FP-Growth [6], which uses a prefix-tree-based main memory compressed representation of the input dataset, is the most popular depth-first based approach. The algorithm is based on a recursive visit of the tree-based representation of the dataset with a "divide and conquer" approach. The Algorithm for this method is reported in [6].

Algorithm 1 (FP-tree construction).

Input: A transaction database DB and a minimum support threshold ξ .

Output: FP-tree, the frequent-pattern tree of DB.

Method: The FP-tree is constructed as follows.

1. Scan the transaction database DB once. Collect F, the set of frequent items, and the support of each frequent item. Sort F in support-descending order as FList, the list of frequent items.

2. Create the root of an FP-tree, T , and label it as "null". For each transaction Trans in DB do the following.

Select the frequent items in Trans and sort them according to the order of FList. Let the sorted frequent-item list in Trans be [p | P], where p is the first element and P is the remaining list. Call insert tree([p | P], T).

Algorithm 2 (FP-growth: Mining frequent patterns with FP-tree by pattern fragment
growth).

Input: A database DB, represented by FP-tree constructed according to Algorithm 1, and a minimum support threshold ξ .

Output: The complete set of frequent patterns.

Method: call FP-Growth(FP-tree, null).

Procedure FP-Growth(Tree, α)

{

(1) if Tree contains a single prefix path // Mining single prefix-path FP-tree

(2) then {

(3) let P be the single prefix-path part of Tree;

(4) let Q be the multipath part with the top branching node replaced by a null root;

(5) for each combination (denoted as β) of the nodes in the path P do

(6) generate pattern β ∪ α with support = minimum support of nodes in β;

(7) let freq pattern set(P) be the set of patterns so generated; }

(8) else let Q be Tree;

(9) for each item ai in Q do { // Mining multipath FP-tree

(10) generate pattern β = ai ∪ α with support = ai .support;

(11) construct β’s conditional pattern-base and then β’s conditional FP-tree Treeβ ;

(12) if Treeβ= ∅

(13) then call FP-growth(Treeβ , β);

(14) let freq pattern set(Q) be the set of patterns so generated; }

(15) return(freq pattern set(P) ∪ freq pattern set(Q) ∪ (freq pattern set(P)
× freq pattern set(Q)))

}

The FP-Growth described above is a main memory based method. However, when the dataset is huge or minimum support is low, the fp tree of a transactional database cannot fit into main memory. There is a huge number of frequent itemsets which needs to generated, this task cannot be performed on a single node machine.

So here we can see that there is a need for parallel implementation of FP-Growth algorithm on a distributed system.

2.3. MAPREDUCE FRAMEWORK

MapReduce [3] is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. A MapReduce program is composed of a Map() procedure (method) that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name) and a Reduce() method that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).

2.4. HADOOP

Apache Hadoop [15] software is open-source software for reliable, scalable, distributed computing. The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. The library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

The base Apache Hadoop framework is composed of the following modules:

- *Hadoop Common* – contains libraries and utilities needed by other Hadoop modules;
- *Hadoop Distributed File System (HDFS)* – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- *Hadoop YARN* – a platform responsible for managing computing resources in clusters and using them for scheduling users' applications.
- *Hadoop MapReduce* – an implementation of the MapReduce programming model for large-scale data processing [15].

MapReduce-based programs implemented on Hadoop do not fit well iterative processes because each iteration requires a new reading phase from disk. This feature is critical when dealing with huge datasets. This issue led to the introduction of Spark, which enables the nodes of the cluster to cache data and intermediate results in memory, instead of reloading them from the disk at each iteration.

2.5. APACHE SPARK

Apache Spark [] is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming [11].

So here we see algorithm for our parallel implementation of FP-Growth on Apache spark.

2.6. ALGORITHM

Input: Dataset as DatasetFile, a minimum support threshold as supportCount

Output: Complete set of frequent patterns

START

1: StartTime = System current time in milli seconds

2: NoTransactions =No of transactions in DatasetFile

3: minSupport=supportCount*NoTransactions

4: freqItems = find all the frequent items in the dataset and sort it in

descending order in respect to their support value

5: For each transaction in DatasetFile Do

6: transaction1=remove infrequent items from the transaction

and sort it according to freqItems

7: n =length(transaction1)

8: i= n-1

9: while i >= 0

10: item = transaction1(i)

11: output(item) = transaction1.slice(0, i)

12: i=i-1

13: End While

14: EndFor

15: data2=output.groupByKey

16: For item, CPBase in data2 do

17: a= Merge all the conditional transaction stored in CPBase and

then remove the frequent ones

18: ConditionalT=CPBase

19: For i=0 to ConditionalT.length do

20: ConditionalT[i]=CPBase[i]-a

```
End For
list=ConditionalT.groupBy(identity).mapvalues(.length)
Declare Patterns as List(String,int)
For each key, value in list do
Declare temp as List(String)
For each itemset in AllPossibleCombination(key) do
temp=temp+ (itemset U item).ToString
End For
Patterns=Patterns+temp.map(word=(word,value))
End For
Patterns=Patterns.reduceByKey()
Patterns=Patterns.filter(>minSupport)
For itemset, support in patterns do
Print itemset, support
EndFor
End For
EndTime=System.currentTimeMillis
Time Taken=EndTime-StartTime
Print (Time Taken)
STOP
```

3. EXPERIMENT

In this section we will discuss about program details, formula used in the source code to solve the problem and see the source code.

3.1. PROGRAM DETAILS

Map inputs (transactions) key="": value	Sorted transactions (with infrequent items eliminated)	Map outputs (conditional transactions) key: value
f a c d g i m p	f c a m p	p: f c a m m: f c a a: fc c: f
a b c f l m o	f c a b m	m: f c a b b: f c a a: f c c: f
b f h j o	f b	b: f
b c k s p	c b p	p: c b b: c
a f c e l p m n	f c a m p	p: f c a m m: f c a a: f c c: f

Reduce inputs (conditional databases) key: value	Conditional FP−trees
p: { f c a m / f c a m / c b }	{(c:3)} | p
m: { f c a / f c a / f c a b }	{ (f:3, c:3, a:3) } | m
b: { f c a / f / c }	{} | b
a: { fc / f c / fc }	{ (f:3, c:3) } | a
c: { f / f / f }	{ (f:3) } | c

Figure 1: A simple example of distributed FP-Growth.

Figure 1 shows a simple example of our implementation of FP-Growth as reported in [6]. The example DB has five transactions composed of lower-case alphabets. The first step that FP-Growth performs is to sort items in transactions with infrequent items removed. In this example, we set $\xi = 3$ and hence keep alphabets f, c, a, b, m, p. After this step, for example, T1 (the first row in the figure) is pruned from {f, a, c, d, g, i, m, p} to {f, c, a, m, p}. FP-Growth then compresses these “pruned” transactions into a prefix tree, which root is the most frequent item f. Each path on the tree represents a set of transactions that share the same prefix; each node corresponds to one item.

3.2. SOURCE CODE

```
package com.ashish

import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
```

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.{HashPartitioner, Partitioner, SparkContext,
SparkException}
import scala.reflect.ClassTag
import scala.collection.mutable
import java.{util => ju}
import scala.collection.JavaConverters._
import java.io._


object test3
{
     def main(args: Array[String])
     {
          val conf  = new
SparkConf().setAppName("TwitterPopularTags").setMaster("local[2]")
          val sc    = new SparkContext(conf)

          val StartTime=System.currentTimeMillis()

          val datasetFile=sc.textFile(args(0))
          val writer = new PrintWriter(new File(args(1) ))
          val NoTransaction=datasetFile.count

          def getMinSupport(minSupport:Double=0.6): Double=
          {
            require(0.0 <= minSupport && minSupport <= 1.0)
            return (minSupport*NoTransaction)
          }

          def getNumPartitions(numPartitions:Int): Int=
          {
            require (numPartitions > 0)
              return numPartitions
          }
```

```
        val dataset=datasetFile.flatMap(t=>t.split(" "))
        val
minSupport=math.ceil(getMinSupport(args(2).toDouble)).toLong
        println(minSupport)
        val numparts=getNumPartitions(5)
        val numParts1 = if (numparts> 0) numparts else
dataset.partitions.length
        val partitioner = new HashPartitioner(numParts1)



        val  freqItems=dataset.map(word=>(word,1))
.reduceByKey(partitioner,_+_).filter(_._2>=minSupport).sortBy(-_._2)

        val itemToRank =
freqItems.map(_._1).zipWithIndex.collect.toMap
        val itemToRankReverse=itemToRank.map(_.swap)


        val data1=datasetFile.flatMap(transaction=>{

                      val output = mutable.Map.empty[String,
Array[String]]
                      var transaction1=transaction.split("
").map(x=>x.toString).toArray
                      var filtered =
transaction1.flatMap(itemToRank.get)
                      ju.Arrays.sort(filtered)

transaction1=filtered.flatMap(itemToRankReverse.get).map(x=>x.toStri
ng)

                      val n = transaction1.length
                      var i = n - 1
                      while (i >= 0)
                      {
                          val item = transaction1(i)
```

```
                    output(item) = transaction1.slice(0, i)
                    i=i-1
                }
                output
            })


        val data2=data1.groupByKey().mapValues(_.toArray).collect()
        val w=freqItems.collect.toMap

        for((i,j)<- data2)
        {
            var patterns:List[(String,Int)]=List()
            var conditionalT:List[List[String]]=List()
            var a=sc.parallelize(j.flatten.map(word=>
(word,1))).reduceByKey(_+_).filter(_._2<minSupport).map(_._1).collec
t

            writer.write(List(i)+":"+w(i).toString+"\n")

            for (z<- 0 to j.length-1)
            {
              conditionalT=(j(z).diff(a)).toList::conditionalT
            }

            Var list=conditionalT.groupBy(identity
.mapValues(_.length)

            for((b,k)<-list)
            {
                var result:List[String]=List()
                for(k<-1 to b.size)
                {
                    for( l<-b.combinations(k) )
                    {
                      var temp=(i::l.toList).toString
                      result=temp::result
```

```
                    }
                }

                patterns=patterns:::result.map(word=>(word,k))
            }


patterns=sc.parallelize(patterns).reduceByKey(_+_).collect.toList

            val patterns1=patterns.filter(_._2>=minSupport)

            for((x,y)<-patterns1)
            {
              writer.write(x+":"+y+"\n")
            }

        }



        val EndTime=System.currentTimeMillis()
        val TT=EndTime-StartTime
        writer.write("Time Taken:- "+TT+"\n")
        writer.close()
         sc.stop()
    }
}
```

4. RESULT AND DISCUSSION

In this section we will see the output of the program and discussion on the output.

4.1. OUTPUT

TABLE 1. DATABASE CHARACTERISTIC

Dataset	**Number of Transaction(N)**	**Number of items(I)**	**Size of dataset**
Retail	88162	16470	4.2 MB
T40I10D100K	100000	942	15.5 MB
T10I4D100K	100000	870	4 MB
mushroom	8124	119	570.4 KB
pumsb_star	49046	2088	11.3 MB

In this section, the results of experimental comparison have been presented. We have evaluated the performance of the Parallel FP-Growth implementation provided in ML for Spark 2.20 and our implementation of Parallel Projection method of FP-Growth as described in [6]. We have evaluated both the implementation on local Spark-2.2.0 cluster installed at personal laptop running Ubuntu 17.04 64bit, where the laptop has 4 core Intel i3 processors running at 2.10 GHz and RAM 8 GB.

We have both used synthetic and real-life datasets in our experiment from [12]. Table 1 describes the important characteristics of the datasets used. Figure 1-5 shows the comparison of running time of our Implementation and spark-ml implementation for varying value of minimum support on datasets mushroom, retail, pumsb_star, T40I10D100K, T10I4D100K respectively.

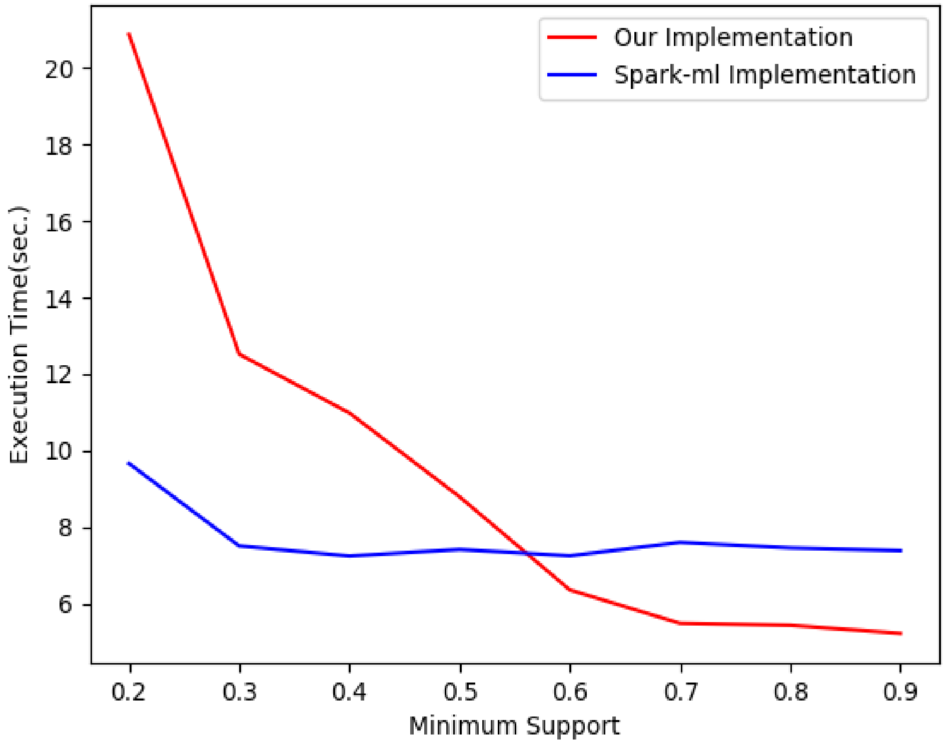

Figure 2: Mushroom Dataset Analysis

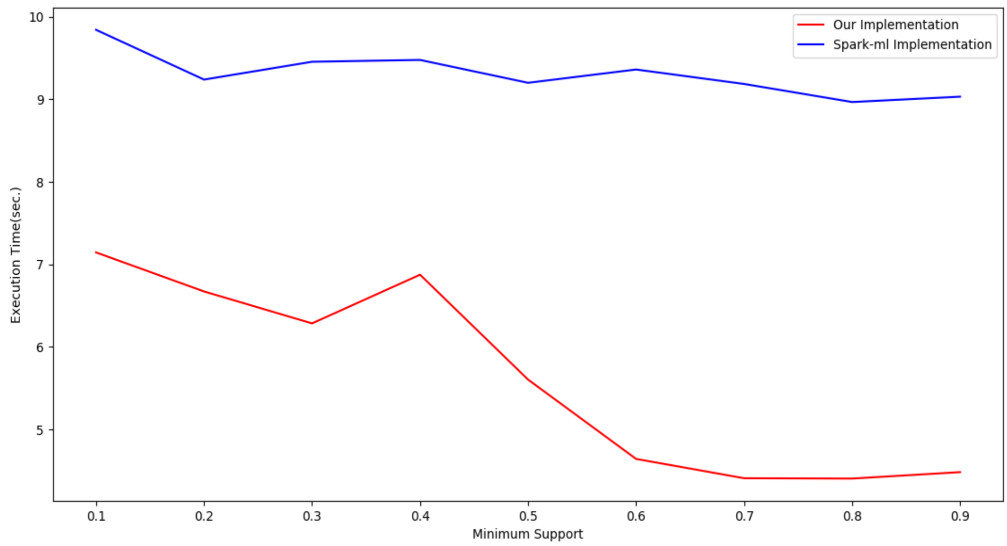

Figure 3: Retail Dataset Analysis

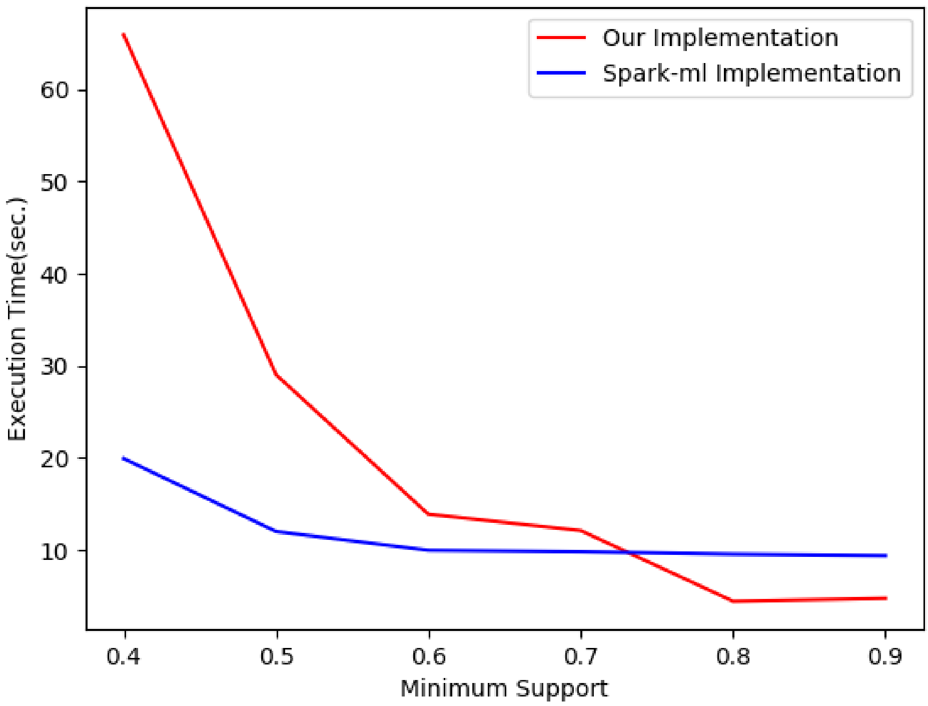

Figure 4: Pumsb_star Dataset Analysis

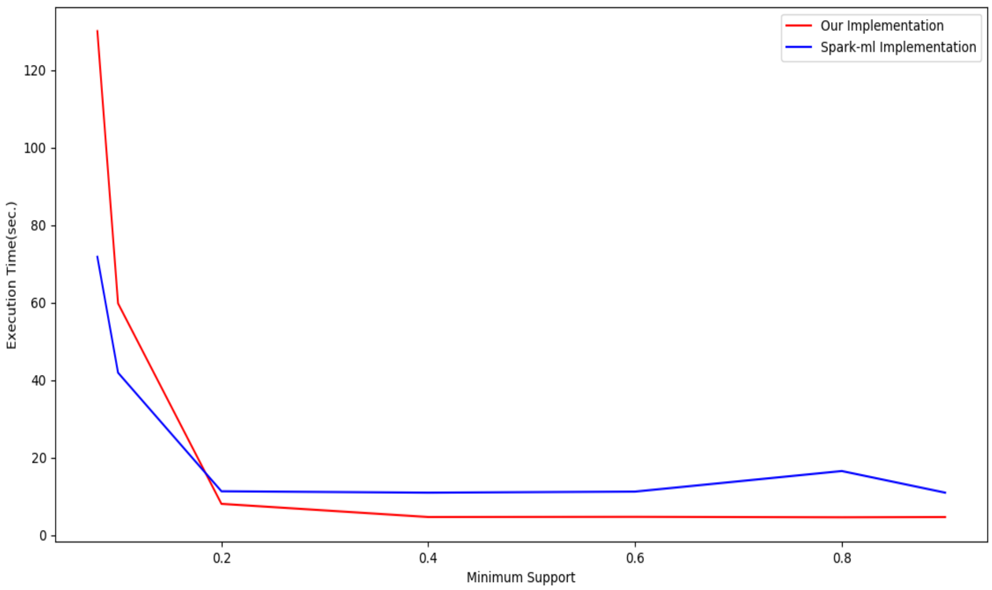

Figure 5: T40I10D100K Dataset Analysis

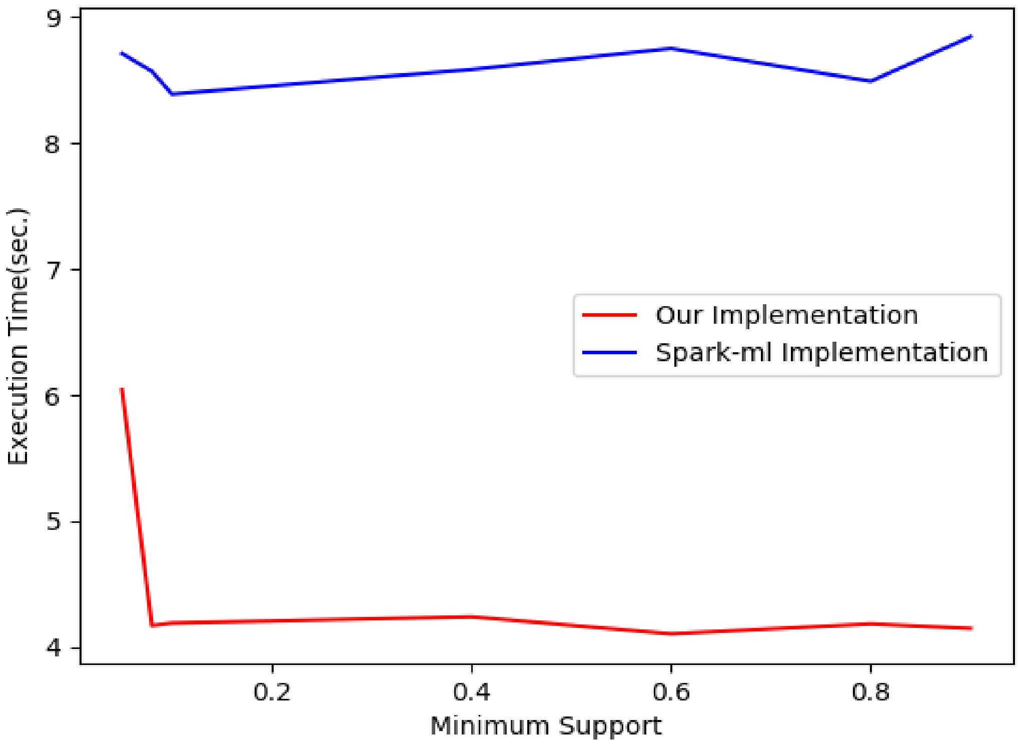

Figure 6: T10I4D100K Dataset Analysis

4.2. BRIEF DISCUSSION

We learn from the results of PFP implementation of spark-ml performs better than our implementation particularly with the datasets who's each transaction contain large number of frequent items and for low value of minimum support. We also observe that Our implementation outperforms spark-ml for higher value of minimum support.

In this paper we presented the Parallel Projection method of FP-Growth algorithm to mine frequent itemsets from transactional data. We demonstrated that the PFP implementation of spark-ml performs well for mining frequent itemsets when the support threshold is low. Spark's in-memory primitives provide performance up to 10 times faster for certain applications such as FIM and provide scalability and efficiency for processing large datasets.

5. FUTURE WORK

In Future we plan to improve our implementation so that it can perform better than spark-ml.

REFERENCES

[1] Sudhakar Singh, Pankaj Singh, Rakhi Garg and P K Mishra, “Big Data: Technologies, Trends and Applications”, In: International Journal of Computer Science and Information Technologies, Vol. 6(5), 2015

[2] Daniele Apiletti, Elena Baralis, Tania Cerquitelli, Paolo Garza, Fabio Pulvirenti and Luca Venturini, “Frequent Itemsets Mining for Big Data: A Comparative Analysis”, Elsevier Inc. All rights reserved, 2017

[3] Arkan A. G. AL-HAMODI and Song-feng LU, “MapReduce Frequent Itemsets for Mining Association Rules”, In: International Conference on Information System and Artificial Intelligence, 2016

[4] Aissatou Diaby dite Gassama, Fode Camara, Samba Ndiaye, “S-FPG: A Parallel Version of FP-Growth Algorithm under Apache Spark™”, In: The 2nd IEEE International Conference on Cloud Computing and Big Data Analysis,2017

[5] Daniele Apiletti, Paolo Garza and Fabio Pulvirenti, “A Review of Scalable Approaches for Frequent Itemset Mining”, T. Morzy et al. (Eds): ADBIS 2015, CCIS 539, pp. 243–247, Springer International Publishing Switzerland, 2015

[6] JIAWEI HAN, JIAN PEI†, YIWEN YIN and RUNYING MAO, “Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach∗”, In: Data Mining and Knowledge Discovery, 8, 53–87, 2004

[7] Jianwei Li, Ying Liu, Wei-keng Liao, Alok Choudhary, “Parallel Data Mining Algorithms for Association Rules and Clustering”, CRC Press, LLC, 2006

[8] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang and Edward Chang, “PFP: Parallel FP-Growth for Query Recommendation”, ACM New York, NY, USA, 2008

[9] Thanmayee and H R Manjunath Prasad, “Revamped Market-Basket Analysis Using In-Memory Computation Framework”, In: 11th International Conference on Intelligent Systems and Control (ISCO), 2017

[10] Bart Goethals (HIIT Basic Research Unit), “Survey on Frequent Pattern Mining”, 2003

[11] Apache Spark, https://spark.apache.org

[12] FIMI (Frequent Itemset Mining *Dataset*) Repository, *fimi.ua.ac.be/data*

[13] Rakesh Agrawal and Ramakrishnan Srikant Fast algorithms for mining association rules. Proceedings of the 20th International Conference on Very Large Data Bases, VLDB, pages 487-499, Santiago, Chile, September 1994

[14] Agrawal, R.; Imieliński, T.; Swami, A. (1993). "Mining association rules between sets of items in large databases". Proceedings of the 1993 ACM SIGMOD international conference on Management of data - SIGMOD '93. p. 207. doi:10.1145/170035.170072. ISBN 0897915925

[15] Apache Hadoop, http://hadoop.apache.org/

[16] Zhiyong Ma, Juncheng Yang, Taixia Zhang and Fan Liu, “An Improved Eclat Algorithm for Mining Association Rules Based on Increased Search Strategy”, In: International Journal of Database Theory and Application Vol.9, No.5 (2016)

[17] Ramesh C. Agrawal, Charu C. Aggrawal and V.V.V. Prasad, “A Tree Projection Algorithm For Generation of Frequent Itemsets”, IBM T. J. Watson research center, Yorktown Heights, NY 10598