CS 6240: Project Report

Team Members

Shashank Shekhar, Barkha Saxena, and Nikhil Anand

Github Spark Repository: https://github.com/CS6240/project-spark-mavericks.git

Project Overview

To increase your sales, do you want to know which itemset is more likely to sell together? Do you want to leverage the patterns in the transactions of your business? Is the transactions data too large to process? Well, we are proposing a solution to all these questions. Apriori algorithm is one of the most widely used techniques for mining frequent itemset from item transactions. Since apriori is very data-intensive and computation-intensive, using parallel algorithms to implement this idea has become quite popular. For the parallel implementation of Apriori, we are using Spark. The reason for selecting spark is to further optimize the iterative nature of the Apriori algorithm of reading and searching pattern in the transactions.

To summarize, we aim to experiment and optimize the implementation of Apriori algorithm and its flavors with pruning to come up with the analysis of our work. We implemented YAFIM (Yet Another Frequent Itemset Mining) [*version 1 of our Apriori code*] in Spark and ran a lot of experiments. According to our observations from these experiments, in version 1 of Apriori we found the bottleneck steps as

1. Broadcasting the exponentially increased candidate itemset generated in intermediate iteration (solved with bloom filter) [A]
2. Searching the entire transactions to calculate the support for each itemset for pruning (solved with hashtree)

To resolve these bottlenecks, we implemented the R-Apriori algorithm [*version 2 of our Apriori code*]. Not only did we resolve the bottlenecks in our first implementation but also tested the scalability and speedup of both the implementations.

Input Data

Describe the data you are working with. Include a line of input, if feasible.

We are using processed transaction data from:

1. Dataset-1: A small dataset for development purpose. This file contains around 8,124 transactions for a catalog of 119 distinct items. Link: http://fimi.uantwerpen.be/data/mushroom.dat
2. Dataset-2:A large dataset for processing. This dataset contains 49,046 transactions for a catalog of 2113 distinct items. Link: http://fimi.uantwerpen.be/data/pumsb_star.dat

Each line represents a transaction in the file. The format of each transaction looks likes: item id separated by space

```
1 3 9 13 23 25 34 36 38 40 52 54 59 63 67 76 85 86 90 93 98 107 113
```

Task Name: Frequent-itemset mining (Spark)

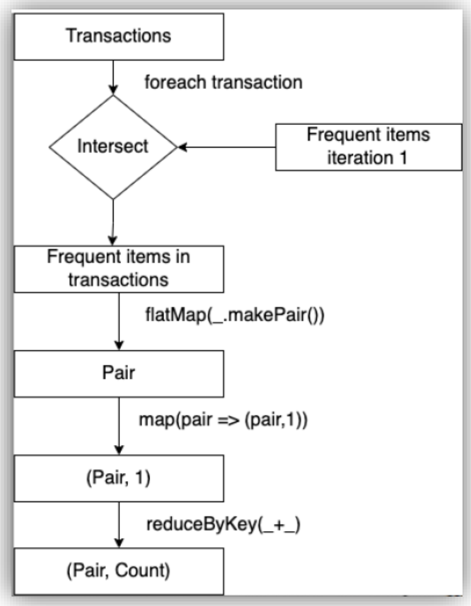
Overview

We are implementing a parallel version of the Apriori frequent itemset mining algorithm and its flavor from scratch. We are using a minimum percentage threshold for support which will be used to remove itemsets with less support than this threshold (pruning step). With this selected subset of itemset, next batch of itemset set is calculated. Each iteration contains several items from range 1 to max items possible in a frequent itemset.

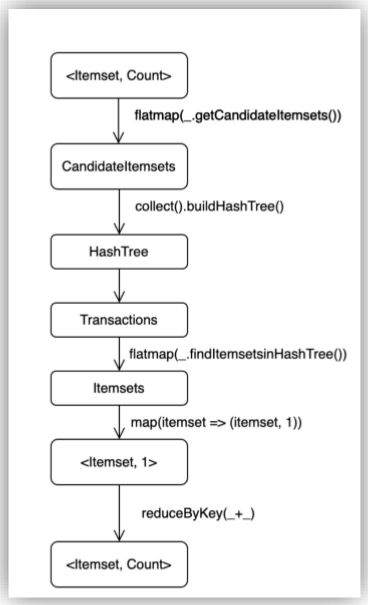
The two important aspects while developing these versions were:

1. **Correctness of the results from each of the two implementations**: We tackled this by initially developing on a small dataset and comparing the resultant itemset generated through the sequential execution of the brute force implementation. As the resultant dataset increased, we used md5 hash and file size to check the correctness of our results.
2. **To improve the scalability of the program, optimizing the space complexity of the algorithm dynamically across iterations**: Our version 1 code broadcasts the candidate itemset to the next iteration. During the processing of a large dataset, it was identified that the candidate itemset in second iteration becomes very expensive as compared to further iterations. We use bloom filters (resolving the space complexity while broadcasting to next iteration) instead of hash trees to eliminate the generation of candidate itemsets in this iteration, streamlining the search for singletons during processing. In the further iterations the number of candidate itemsets is already reduced hence the working of our version 1 and version 2 codes remain the same.

Pseudo-Code



Flowchart-1: Showing the algorithm flow of the vanilla Apriori implementation in spark r



<Itemset, Count>
flatmap(_.getCandidateItemsets())
CandidateItemsets
collect().buildHashTree()
HashTree
Transactions
flatmap(_.findItemsetsinHashTree())
Itemsets
map(itemset => (itemset, 1))
<Itemset, 1>
reduceByKey(_+_)
<Itemset, Count>

Flowchart-2: Showing schema in an iteration

```
object Apriori {
        def main(args: Array[String]) {
        val apriori = new Apriori
        //calling run method to begin execution of apriori algorithm with minSupport “0.35”
        Class Apriori {
        // method to start execution of apriori i.e. first iteration of apriori
        def findFrequentItemsets {
                fileRDD = sc.textFile(filename)
                //getting transasctionsRDD from fileRDD
                transactionsRDD = fileRDD.filter(!_.trim.isEmpty) // filtering out empty lines
                  .map(_.split(separator + "+")) // converting each transaction line into an array
                  .map(l => l.map(_.trim).toList) // converting each transaction array to list
                // Generate single item(singleton) RDD i.e. first candidate set
                // by filtering out items with frequency less than support
                val singleItemRDD = transactionsRDD
                  .flatMap(identity)
                  .map(item => (item, 1))
                  .reduceByKey(_ + _) // filtering duplicate value
                  .filter(_._2 >= supportCount) // filtering out items with lesser supportCount value
                frequentItemsets = findFrequentItemsets(transactionsRDD, singleItemRDD, supportCount, sc)
                return frequentItemsets
        }
        // method to run multiple iterations of apriori algorithm
        def findFrequentItemsets(transactions, singletons) {
         //Creating a mutable map where key is size of itemset and value is list of itemsets of that size
         //Initially, mapping singleton itemsets with key '1'
         val frequentItemsets = mutable.Map(1 -> singletons.map(_._1).map(List(_)).collect().toList)
         var k = 1
        // running iterations of apriori algorithm until we get empty set of itemsets
                while (frequentItemsets.get(k).nonEmpty) {
                        k += 1
                        // Generating candidate set for next iteration and performing pruning
                        val candidates = generateCandidates(frequentItemsets(k - 1), sc)
                        // Comparing frequent itemsets with transactionsRDD, and filtering out itemsets
                        // with frequency less than minimum support threshold
                        val kFrequentItemsets = filterFrequentItemsets(candidates, transactions, minSupport, sc)
                        if (kFrequentItemsets.nonEmpty) {
                        // adding new itemsets to mutable map ‘frequentItemsets’
                        frequentItemsets.update(k, kFrequentItemsets)
                }
        }
        return “list of frequentItemsets”
    }
        // method to generate candidate itemsets
        def generateCandidates(frequentItemSets) {
            val previousFrequentSets = sc.parallelize(frequentItemSets)
              val cartesian = previousFrequentSets.cartesian(previousFrequentSets)
               .filter { case (a, b) =>
                 a.mkString("") > b.mkString("")
            }
              cartesian
               .flatMap({ case (a, b) =>
                 var result: List[Itemset] = null
                 //only one element among two itemsets should be different
                 if (a.size == 1 || allElementsEqualButLast(a, b)) {
                   val newItemset = (a :+ b.last).sorted
                 if (pruneItemsets(newItemset, frequentItemSets))
                     result = List(newItemset)
                 }
                 if (result == null) List.empty[Itemset] else result
               })
               .collect().toList
            }
            // method to prune itemsets
        def pruneItemsets(itemset: List[String], previousItemsets: List[Itemset]): Boolean = {
          for (i <- itemset.indices) {
            val subset = itemset.diff(List(itemset(i)))
            // checking if itemset is present in any subset of previous iteration’s itemset
            val found = previousItemsets.contains(subset)
            if (!found) {
              return false
            }
          }
```

```
  true
}

def saveFinalItemsetRDD(itemsets: List[Itemset], output: String, sc: SparkContext) = {
  val rdd = sc.parallelize(itemsets).map(x => (x.size, x))
  val rdd2 = rdd.map(x => x._1).map((x) => (x, 1)).reduceByKey(_+_)
// saving final rdd
  rdd2.saveAsTextFile(output)
}
```

Optimization for RApriori:

```
// Broadcasting bloomfilter in second iteration of Rapriori for streamlining the search for singletons during processing.
def findPairsBloomFilter(transactions, singletons, minSupport, bc){
  // iterating over the singletons
  for set in singletons:
     bloomfilter.add(set)
  // broadcasting the bloom filter
  val bfBC = sc.broadcast(bf)
transactions.map(t=>t.filter(bfBC.value.mightContains(_)))
  .flatMap(_.combinations(2)) // generate combinations of size 2
  .map((_,1))
 .reduceByKey(_+_)
 .filter(_._2 >= minSupport) // filtering out based on minsupport
 .map(_._1.sorted)
 .collect().toList
}
// performing filtering using HashTree
def filterFrequentItemsets(candidates, transactionsRDD, minSupport) = {
    val items = candidates.flatten.distinct
    val hashTree = new HashTree(candidates, items) // creating hashtree for all candidate itemsets
    val hashTreeBC = sc.broadcast(hashTree) // broadcasting hashtree to optimally search candidates
    val r = transactionsRDD.flatMap(t =>
      hashTreeBC.value.findCandidatesForTransaction(t.filter(i => items.contains(i)).sorted))
      .map(candidate => (candidate, 1))
      .reduceByKey(_ + _)
      .filter(_._2 >= minSupport) // filtering on the basis of minimum support
      .map(_._1)
      .collect().toList
}
```

Algorithm and Program Analysis

Given below are steps for Apriori algorithm implementation:

Step 1: We read input file from the input directory and split it by spaces to fetch separate transactions from it. While creating an RDD for transactions, we make sure that it has a minimum of 8 partitions(parameter).

Step 2: We find individual items present among all transactions using flatMap and reduceByKey operations. For these individual items, we create candidate itemsets of size 1. At this stage, we ignore the itemsets that appear less than the minimum support value.

Step 3: Now, we generate new itemsets by performing cartesian product i.e. self join (shuffling takes place). At this step, we also make sure that we do not generate duplicate itemsets by ensuring an order inside an itemset.

Step 4: We now broadcast the itemset in current iteration and use it to prune the newly generated candidate itemset. If any subset of the candidate itemset is not present in the broadcasted itemset, then we prune it and generate the frequency itemset for the next iteration.

Step 5: We repeat steps 3 and 4 until we reach an iteration where there are no elements in the frequency itemset. The last non-empty frequency itemset generated is the frequent itemset that we need. We save this itemset in the output directory.

Experiments

Github Spark Repository: https://github.com/CS6240/project-spark-mavericks.git

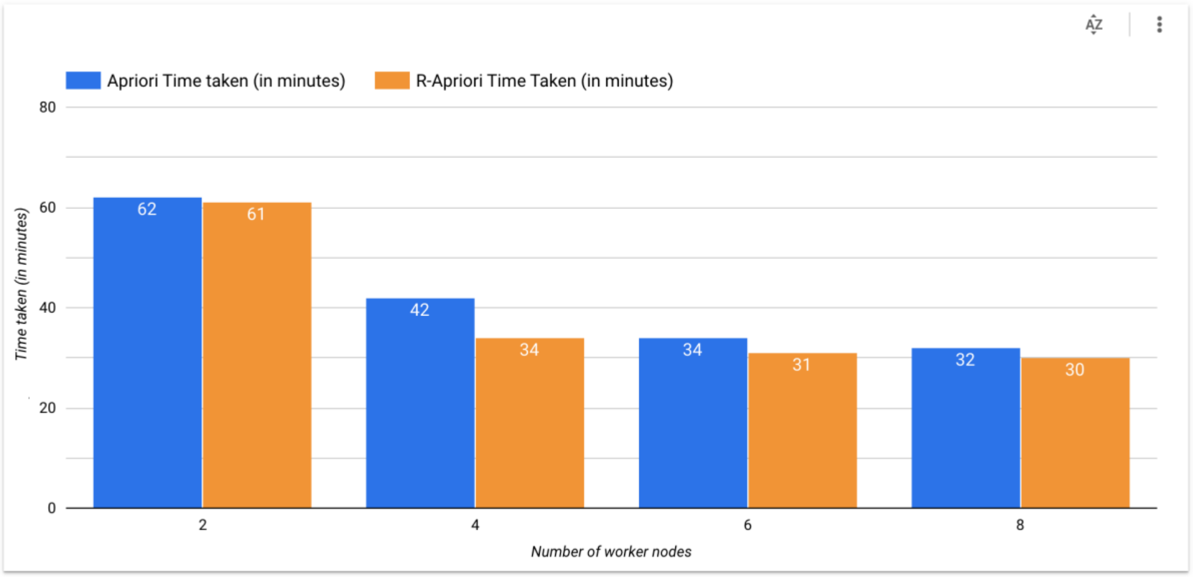
Speedup

For speedup, we fixated the work to be done as d (size of the dataset-2) and ran our code implementation of R-Apriori with multiple cluster sizes. The configuration for these set of experiments were:

Workload: d (size of dataset-2) (**constant**)
Cluster: emr-5.17.0, 1 master and "x" m3.xlarge EC2 workers (x values as on X-axis)
Parameter: minimum support = 0.35 (**constant**)

[Table 1.] Showing the time and output (includes stderr as well) for Apriori and Rapriori implementations on AWS EMR for varying cluster size

Num of Nodes (X-axis)	Apriori Time in minutes (Y-axis)	R-Apriori Time in minutes (Y-axis)	Apriori output	R-apriori output
2	62	61	output	output
4	42	34	output	output
6	35	31	output	output
8	32	30	output	output



[Fig 1.] Showing the speedup for Apriori and R-Apriori implementations on AWS EMR

Observation: The difference in run-time between Apriori and R-Apriori increased when we increased the number of nodes from 2 to 4. The difference in run-time started to decrease when we increased the number of nodes from 4 to 6 and it decreased further upon increasing worker nodes from 6 to 8.

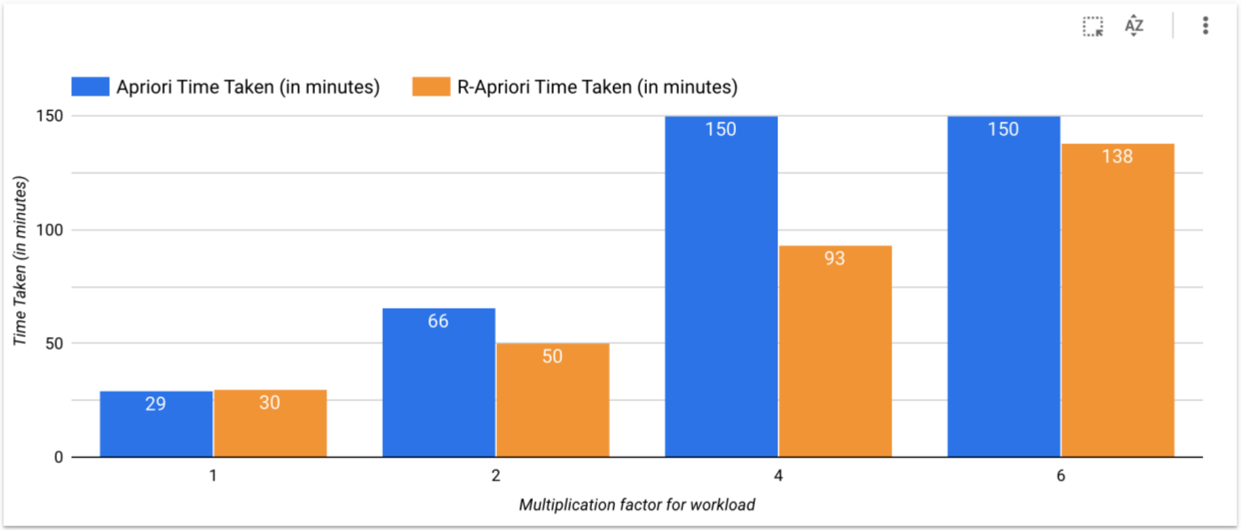
Scalability

For scalability, we fixated the cluster size and ran our code implementation of R-Apriori with multiple input sizes. The configuration for these set of experiments were:

Workload: “x” times d (size of dataset-2) (x values as on X-axis)
Cluster: emr-5.17.0, 1 master and 8 m5.xlarge EC2 workers (**constant**)
Parameter: minimum support = 0.35 (**constant**)

[Table 2.] Showing the input size and output (includes stderr as well) for Apriori and Rapriori implementations on AWS EMR for varying cluster size

Input Dataset size (X-axis)	Apriori Time in minutes (Y-axis)	R-apriori Time in minutes (Y-axis)		R-apriori output
d	29	30		output
2d	66	50		output
4d	150+	93		output
6d	150+	138		output



[Fig 2.] Showing the scalibility for Apriori and Rapriori implementations on AWS EMR with varying the workload size

Observation: On smaller dataset (with no replication), both Apriori and R-Apriori took almost the same amount of time. When we replicated our dataset(2x), there was significant improvement in the runtime of R-Apriori (22.7%). When we replicated our datasets further i.e. 4 times and 6 times, the Apriori algorithm timed out (> 180 minutes) whereas R-Apriori ran significantly fast giving runtime of 93 and 138 minutes respectively.

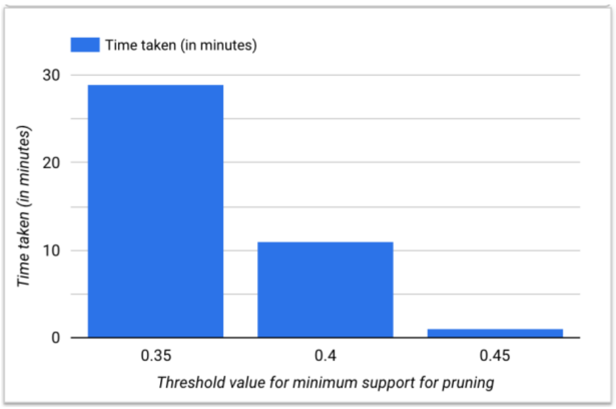
Minimum Support Variations

We wanted to observe the effect of changing the minimum support threshold set in our code implementation of R-Apriori for pruning. A higher threshold implies we get fewer itemset after pruning and vice-versa.

Workload: d (size of dataset-2) (**constant**)
Cluster: emr-5.17.0, 1 master and 8 m5.xlarge EC2 workers (**constant**)
Parameter: minimum support = *"x"*

[Table 3.] Showing the the minimum support threshold for pruning and time taken by RApriori algorithm *on AWS EMR*

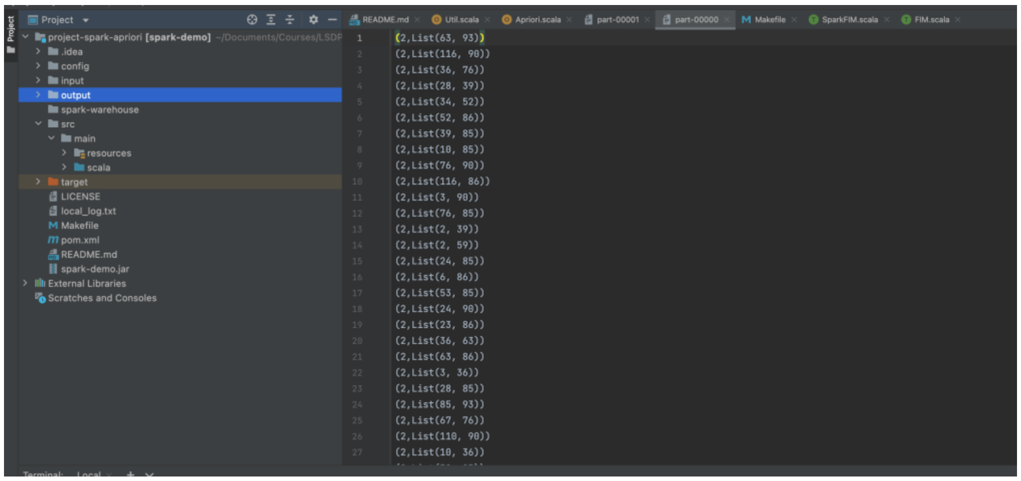
Minimum support (X-axis)	Time taken (in mins) by RApriori (Y-axis)	Output links
0.35	29	output
0.4	11	output
0.45	1	output



[Fig 3.] Showing the effect of variation of threshold minimum support for pruning on the processing time of R-Apriori on AWS EMR

Observation: For higher values of minimum support such as 0.4 and 0.45, the R-Apriori algorithm ran very fast i.e. in 11 minutes and 1 minute respectively. This was because most itemsets were pruned because of higher support value. For minimum support value of 0.35, the algorithm took 29 minutes. This was because there were less itemsets pruned in each iteration and more candidate sets got generated taking more time. As expected, on increasing the minimum support threshold value, the processing time decreased.

Result Sample



```
(2,List(63, 93))
(2,List(116, 90))
(2,List(36, 76))
(2,List(28, 39))
(2,List(34, 52))
(2,List(52, 86))
(2,List(39, 85))
(2,List(10, 85))
(2,List(76, 90))
(2,List(116, 86))
(2,List(3, 90))
(2,List(76, 85))
(2,List(2, 39))
(2,List(2, 59))
(2,List(24, 85))
(2,List(6, 86))
(2,List(53, 85))
(2,List(24, 90))
(2,List(23, 86))
(2,List(36, 63))
(2,List(63, 86))
(2,List(3, 36))
(2,List(28, 85))
(2,List(85, 93))
(2,List(67, 76))
(2,List(110, 90))
(2,List(10, 36))
```

Conclusion

We ran a total of 33 experiments (including success and failures) as a team of three members. Only the relevant experiments are presented in the report. From the implementation of Apriori and Rapriori algorithm in parallel, we conclude that unlike the apriori algorithm, our implementation of the RApriori algorithm is highly scalable, as can be clearly inferred from the experiment results and visualizations. In the RApriori, we have used Hashtree and Bloom Filters to optimize the bottlenecks of the Apriori algorithm. In future we intend to extend the current algorithm to derive association rules in parallel which can then derive value for business decision making process.

Contributions:
All the members of the team discussed and understood the Apriori algorithm, bottleneck and its resolution. Shashank worked on the Apriori(YAFIM) algorithm, R-Apriori algorithm, running majority experiments on aws, flowcharts and report(pseudo code), Barkha worked on the Apriori(YAFIM) algorithm, R-Apriori algorithm, running experiments on aws and creating the report and visualization for the experiments, and Nikhil worked on the Apriori(YAFIM) algorithm, R-Apriori algorithm, running experiments on aws and report. In future, we intend to work on association rule mining building up on the current codebase and experiment to improve the performance using different data structures or techniques of optimization. Some of the areas of improvements could be the iterative nature, in-memory computations with shuffling, creation of candidate itemsets.

References

- **[A.]** Work on feedback from intermediate report: “Alternatively, find a solution for when the set of frequent itemsets from the previous iteration cannot be broadcast, e.g., when it is too big for the memory”
- **[B.]** Work on feedback from intermediate report “Code is often difficult to read. In the final report, show the "schema" of each RDD/DataSet (what data fields does it contain). Also explain the intuition how you implemented pruning: e.g., "we are using ... join to do ..." State assumptions you are making, e.g., "we assume the set of frequent (k-1)-itemsets fits in memory... etc"

- **Report of Experiment**: https://datastudio.google.com/reporting/141271f1-d59b-4e00-b5ad-34661a3f5bd1

- **Flowcharts drawn** on draw.io: Flowchart-1: chart link, Flowchart-2: chart link

- **RApriori**: RATHEE, S.; KAUL, M.; KASHYAP, A. R-Apriori: An Efficient Apriori based Algorithm on Spark. In Proceedings of the 8th Workshop on Ph. D. Workshop in Information and Knowledge Management ACM, p. 27–34, 2015

- **YAFIM**: H. Qiu, R. Gu, C. Yuan and Y. Huang, "YAFIM: A Parallel Frequent Itemset Mining Algorithm with Spark," 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, 2014, pp. 1664-1671, doi: 10.1109/IPDPSW.2014.185.