CS 6240: Project Report

Team Members

Ameya Rane, Xuejie Guo, Saurav Shaw

Project Repository:-
https://github.com/CS-6240-2023-Summer-1/project-spark-frequent-item-set-in-data-set

Project Overview

The project focuses on developing a parallel implementation of the Apriori algorithm called R-Apriori on the Spark platform. The goal is to enhance the efficiency and performance of frequent itemset mining in large-scale transactional datasets. R-Apriori introduces optimizations and modifications to the traditional Apriori algorithm, leveraging the distributed processing capabilities of Spark to handle big data scenarios.

Through our experiments, we measured the execution times and resource utilization of R-Apriori under different settings. These results allowed us to analyze the algorithm's speedup potential and its ability to handle large-scale datasets efficiently. The outcomes of our project contribute to the understanding of parallel Apriori implementations and their suitability for big data analytics tasks.

Input Data

The dataset contains clickstream data from e-commerce.

- The number of sequence counts is 77512.
- The number of item count is 3340.
- The average sequence length is 4.62.

The description of the dataset can be found at:
https://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php

Each transaction is separated by "-2". Each item within the transaction is separated by "-1".

82475 -1 84211 -1 86919 -1 86927 -1 86943 -1 -2

Link to dataset:

https://www.philippe-fournier-viger.com/spmf/datasets/BMS2.txt

Apriori Frequent Itemset Mining Algorithm

Overview

The goal is to implement the apriori frequent itemset mining algorithm using spark using distributed computing and output the frequent itemsets with 2 to k elements. Conduct experiments to test the speedup and scalability of the algorithm.

Pseudo-Code

```
object AprioriMain {
  def candidateGeneration(frequentItemSet, k: Int)
  {
      val cartesian_product = frequentItemSet.cartesian(frequentItemSet)
        .filter(x => x._1.mkString(",") < x._2.mkString(",")) // for (a, b)
join (b, c) only considers (a,b)(b,c) not (b,c)(b,a)
        .map(x => x._1.union(x._2).distinct)// union
        .filter(x => x.length == k + 1) // only consider the set with size =
curSetSize + 1
        .map(x => x.sorted)
        .distinct()
      cartesian_product
   }


  def main(args: Array[String]) {

    var metrics = sc.emptyRDD[(Long,Long, Double)]
    val minSupportCount = 50
    val inputRDD = sc.textFile(args(0)).sample(false, 1) //Takes adjustable
sample of the dataset
    val singletonFrequentItems = inputRDD
      .flatMap(transaction => transaction.split(" -1"))
      .flatMap(transaction => transaction.split("-2")) // Split each
transaction into items
      .map(item => (item.trim(), 1)) // Convert each item into a key-value
pair (item, 1)
      .filter(item => item._1.length>0)
      .reduceByKey(_ + _) // Calculate the frequency of each item
      .filter(_._2 >= minSupportCount)
      .map(item => List(item._1)) // Prune items with frequency less than
minSupportCount

    // val candidates = candidateGeneration(singletonFrequentItems,sc, 1)
    // val tree = new HashTree.HashTree(candidates.collect().toList)
    // candidates.saveAsTextFile("output")
    var k = 1

    var lastRDD = singletonFrequentItems
    val transactions = inputRDD.flatMap(transaction => transaction.split("
-2")).map(transaction => transaction.split("-1").map(x =>
x.trim()).toList).map(x => x.sorted()) // Split along the delimiter
```

```
    while (k <= 15){
      val t1 = System.nanoTime //start time
      val candidatesRDDS = candidateGeneration(lastRDD,sc, k);
      val candidates = sc.broadcast(new
HashTree.HashTree(candidatesRDDS.collect().toList)) //broadcast hashtree to
all nodes
      val newCandidates  = transactions.flatMap(
        line => {
          val CT = candidates.findCandidatesForTransaction(line)
          CT.map(
            candidates => (candidates,1)
          )

        }
      )

      lastRDD = newCandidates.reduceByKey( _ + _).filter(row => row._2 >=
minSupportCount).map(row => row._1) //Prune items not appearing enough times
      if (lastRDD.count() < 2){ //If too few items remaining
        break;
      }
      k = k+1
      lastRDD.saveAsTextFile()
      val duration = (System.nanoTime - t1) / 1e9d //end time
      metrics = metrics.union(((k,lastRDD.count,duration)))
      lastRDD.persist();
    }
    metrics.coalesce(1).saveAsTextFile(args(1) + "/" + "metrics")
  }
}
```

Github Link:

https://github.com/orgs/CS-6240-2023-Summer-1/teams/frequent-item-set-in-data-set

Algorithm and Program Analysis

1st Iteration

In the first iteration we simply create a list of all the distinct items after preprocessing and removing delimiters and empty spaces. Then we sum together the count of each item to find the total number of times each item has appeared in the dataset. Finally, we prune out the items which do not appear at a frequency greater than the minimum threshold.

Kth Iteration

In the kth iteration, the goal of the algorithm is to generate a candidate set of size k-1, using candidates of size k. In the first step, we do a self-cartesian product on the dataset of the

candidates of length k. After pruning out the duplicate candidates we only keep the ones with k+1 unique items. This candidate set is used to create a hashtree. This hashtree is a read only data structure that is broadcast to all the tasks. This is effectively a broadcast and join algorithm.

In each task, we go through every transaction present there and find the matching transactions in the hashtree. For each transaction, we emit the matching candidates each with count 1. At the end, we group all the candidates by key and prune the candidates with count less than the minimum required count. This is persisted to memory to prevent an infinitely long lineage from being created. If the candidate set contains too few items (In our case < 2), the program stops there. Otherwise this is fed into the k+1th iteration.

Experiments

In our project, we implemented R-Apriori, a parallel Apriori algorithm on the Spark platform. We set the minimum support value to 50 and chose a maximum itemset length of 20 (k=15).

To evaluate the speedup capability of R-Apriori, we conducted experiments using a full dataset with different numbers of machines. Specifically, we tested the algorithm's performance using 4 and 8 machines. By leveraging the parallel processing capabilities of Spark and distributing the workload across multiple machines, we aimed to achieve faster execution times for frequent itemset mining.

Furthermore, we assessed the scalability of R-Apriori by varying the portion of data used for analysis. We experimented with data portions of 0.25, 0.5, and 1, representing different dataset sizes. This allowed us to examine how R-Apriori performed when handling varying data volumes, providing insights into its scalability characteristics.

Links to output and logs files :-

- 2 machines full dataset:

  output: output_2machines

  log: stderr_2machines.txt
- 4 machines full dataset:

  output: output_4machines

  log:stderr_4machines.txt
- 8 machines full dataset:

  output: output_8machines

  log: stderr_8machines.txt
- 4 machines 0.5 dataset:

  output: output_4machines_0.5dataset

log: stderr_4machines_0.5data.txt

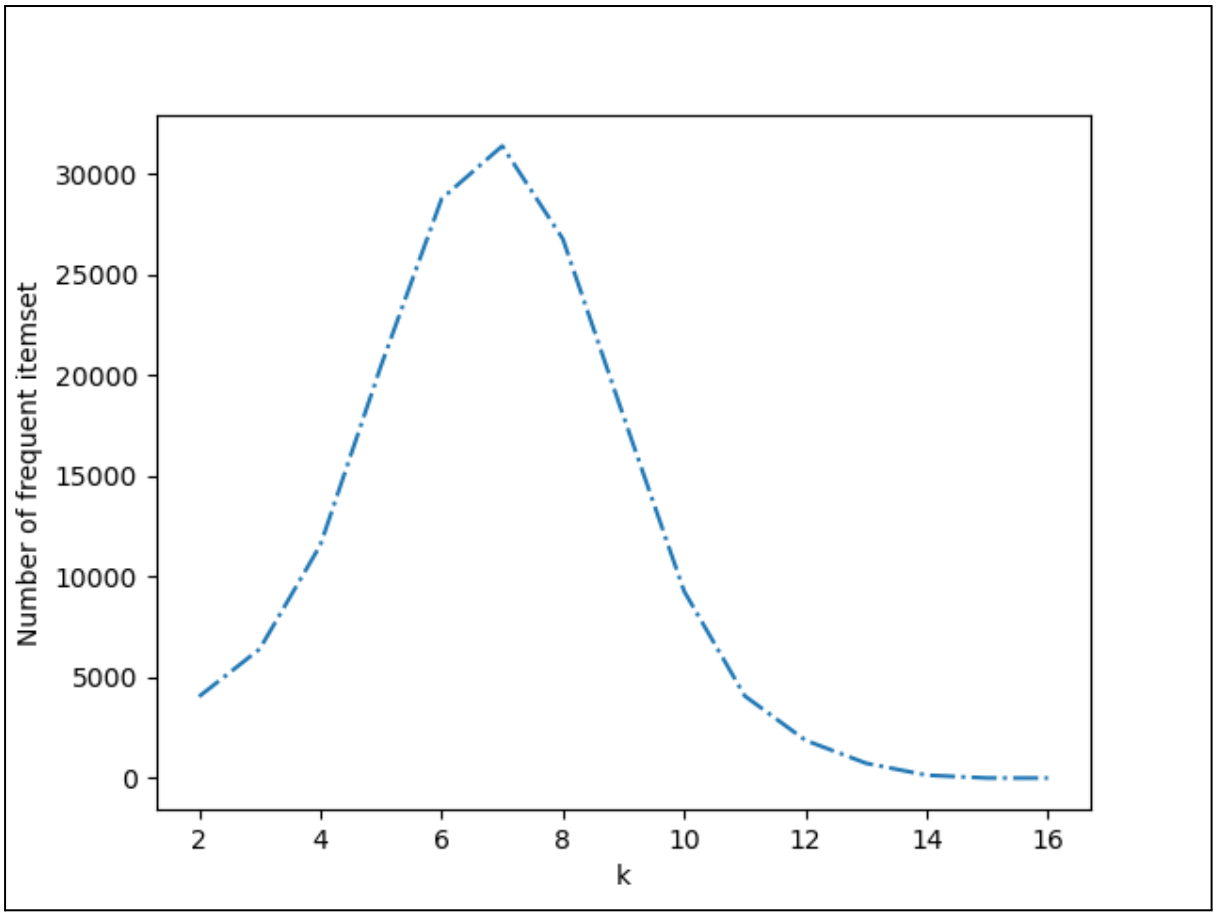
- 4 machines 0.25 dataset:

  output:output_4machines_0.25dataset

  log: stderr_4machines_0.25.txt

Results

In the experiments, we set the minSupportValue equal to 50.

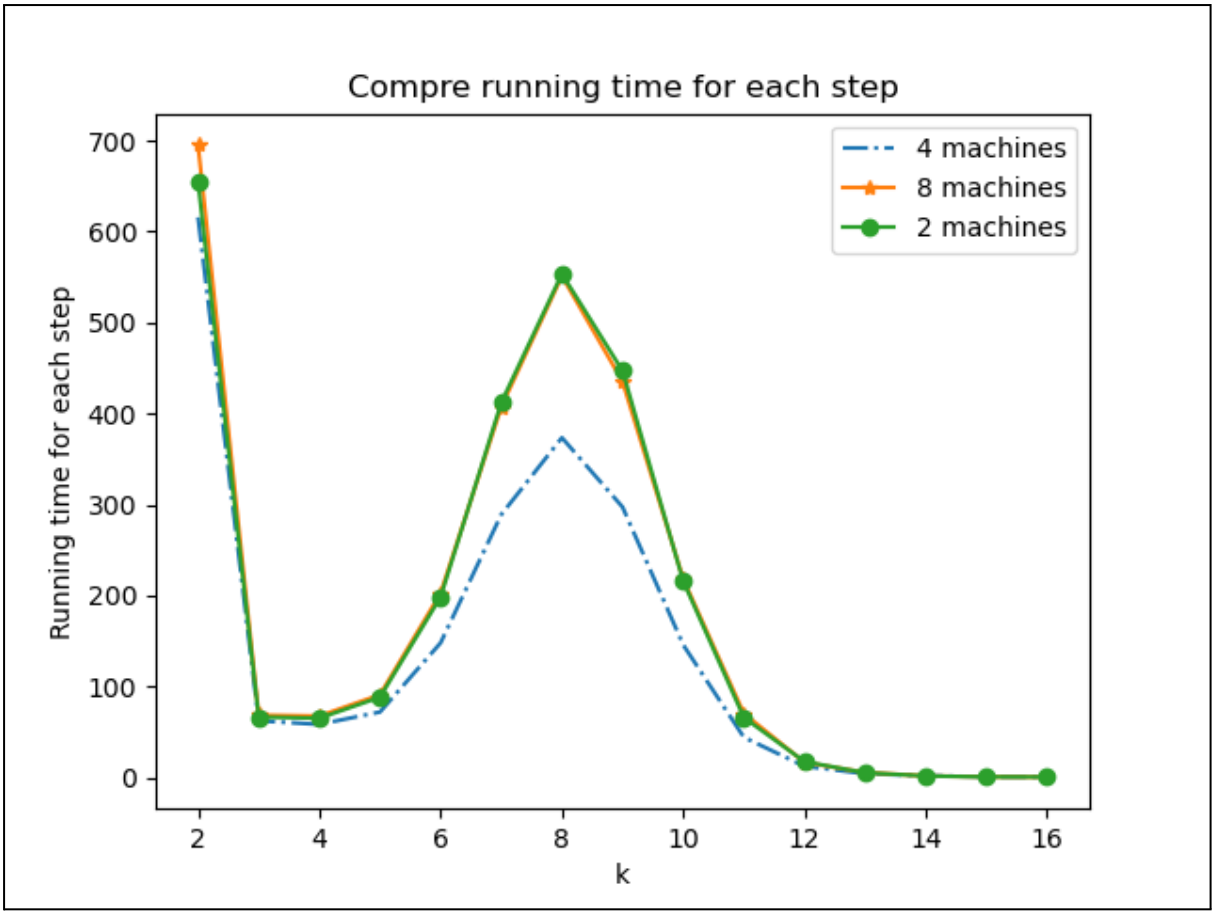


From the graph above, we can see that the number of frequent itemsets keeps increasing until k = 7 then decreasing. This follows the pattern of the pascal's triangle, where the number of unique combinations increases up to a point and then drops down as k approaches the limit of the largest few transactions.

Speedup

To analyze the speedup, we do trials with 2, 4, 8 m5.xlarge machines and the entire dataset.

- Running time at each step



- Total running time
  - 2 machines:

    Start time: 23/06/28 22:27:44

    End time: 23/06/28 23:14:37
  - 4 machines:

    Start time: 23/06/28 20:21:24

    End time: 23/06/28 20:57:08
  - 8 machines:

    Start time: 23/06/28 21:16:38

    End time: 23/06/28 22:04:08

Num of machines	2	4	8
Running time	2873s	2144s	2850s

Compared speed up using 2 machines and 4 machines:

2873/2144 = 1.34

Compared speed up using 4 machines and 8 machines:

2144/2850 = 0.75

From the graph and the table in this section, we can see the running time for 2 machines is almost the same as using 8 machines. Both of them have longer running time than using 4 machines. The possible reason that 4 machines achieve the best running time is that compared to using 2 machines, 4 machines achieves better speedup to using more machines to do pruning and in parallel. And for 8 machines, broadcasting the whole hash tree to all 8 machines becomes the new bottleneck, so it takes longer.

Scalability

We conduct experiments with sample fractions of 0.25, 0.5, and 1 to examine the scalability using AWS. In the experiment, four machines with the instance type m5.xlarge are used.

- Sample fraction: 0.25

  Start time: 23/06/28 23:58:21

  End time: 23/06/28 23:59:12

- Sample fraction: 0.5

  Start time: 23/06/28 23:43:20

  End time: 23/06/28 23:46:26

- Sample fraction: 1

  Start time: 23/06/28 20:21:24

  End time: 23/06/28 20:57:08

Sample fraction	0.25	0.5	1
Running time	51s	186s	2144s

The result shows that every time the input is doubled, the running time increase dramatically.

The quantity of amount of work does not, however, increase linearly with the magnitude of the input. Analyzing the scalability by the size of input may not be a good approach in this case.

Result Sample

- The file in the metrics folder contains one document. Each line of the file contains three elements. The first element indicates which step, the second element is the

number of frequent itemsets in kth step, and the last element indicates the running time to complete kth. The following is the snapshot of the metrics file.

```
(2,4046,615.767294836)
(3,6403,62.312397487)
(4,11560,58.406363941)
(5,20455,71.982717964)
(6,28744,148.03902148)
(7,31364,288.991414427)
(8,26745,373.492881646)
(9,17941,297.237671591)
(10,9264,145.817303112)
(11,4072,44.225857344)
(12,1874,12.335104541)
(13,734,4.199386738)
(14,143,1.99142193)
(15,2,0.766218665)
(16,0,0.539926724)
```

- The frequent itemsets generated at kth step are those with numerical values as their name. For example, all frequent itemsets with 2 elements are stored in the folder with name “2”, and for all frequent itemsets with 3 elements are stored in the folder with name “3”. Each line of the file is one frequent itemset with kth element.
  For lines in folder “3”, the output looks like:

```
List(55267, 55315, 55379)
List(55295, 55307, 55339)
List(55267, 55307, 55331)
```

Conclusions

The Apriori algorithm was used in this project to resolve the frequent itemset mining issue. With only four machines, the algorithm performs well, but as more machines are added, the speedup decreases. The hash tree broadcasting may be the bottleneck. Future attempts at splitting candidate itemsets to create numerous hashtrees and check whether certain candidate itemsets are a subset of a transaction using the bucket technique can be made during the pruning process.