MARKET BASKET ANALYSIS ALGORITHM WITH MAPREDUCE USING HDFS

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Aditya Nuthalapati

In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

May 2017

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Market Basket Analysis Algorithm with MapReduce using HDFS

By

Aditya Nuthalapati

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Simone Ludwig
Chair

Dr. Annie TangPong

Dr. Jun Kong

Approved:

May 12, 2017
Date

Dr. Kenneth Magel
Department Chair

ABSTRACT

Market basket analysis techniques are substantially important to every day's business decision. The traditional single processor and main memory based computing approach is not capable of handling ever increasing large transactional data. In today's world, the MapReduce approach has been popular to compute huge volumes of data, moreover existing sequential algorithms can be converted in to MapReduce framework for big data.

This paper presents a Market Basket Analysis (MBA) algorithm with MapReduce on Hadoop to generate the complete set of maximal frequent item sets. The algorithm is to sort data sets and to convert it to (key, value) pairs to fit with the MapReduce concept. The framework sorts the outputs of the maps, which are then input to the "reduce" tasks. The experimental results show that the code with MapReduce increases the performance as adding more nodes until it reaches saturation.

ACKNOWLEDGEMENTS

I would take this opportunity to thank my advisor, Dr. Simone Ludwig, who has given me valuable support, encouragement and advice without which this work would not have been completed. I am thankful to the members of the committee, Dr. Annie TangPong, Dr. Jun Kong, for their support. I would also like to thank my parents for their valuable support and constant encouragement.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES vii

LIST OF FIGURES viii

1. INTRODUCTION 1

2. MAP REDUCE IN HADOOP 4

2.1. Map/Reduce in Parallel Computing 4

2.2. Database for Big Data 5

2.3. Issues of Map/Reduce 6

3. MARKET BASKET ANALYSIS ALGORITHM 8

3.1. Data Structure and Conversion 9

3.2. The Algorithm 11

3.3. Market Basket Analysis Algorithm Implementation using HDFS 12

3.4. Code Execution 16

4. EXPERIMENTAL RESULTS 17

4.1. Data Distribution Between Mappers 17

4.1.1. Important Observations 18

4.2. Data Distribution Between Reducers 19

4.3. Varying Number of Mappers 19

4.3.1. Important Observations 22

4.4. Varying number of Reducers 24

4.4.1. Important Observations: 29

4.5. Other Factors affecting the Execution Time 30

4.5.1. Speculative execution 30

4.5.2. Nodes with poor performance 31

4.5.3. Data locality and distribution of data blocks 31

4.5.4. Parallelism control with input split size 32

5. CONCLUSION 33

6. REFERENCES 34

LIST OF TABLES

Table Page

1. Data blocks distribution for each Mapper ... 18

2. Variation of execution time due to increase in number of Mappers ... 22

3. Variation of execution time due to increase in number of reducers ... 29

LIST OF FIGURES

Figure Page

1. Map/Reduce Flows 4

2. Transaction data at a store 8

3. Top 5 set of three items that occurred frequently at a store 8

4. Data Set restructured for Map/Reduce 9

5. Data Set Restructured for the same list 10

6. Data Set Restructured with Sort 11

7. Map function 13

8. Calculate combination function 14

9. Reduce function 15

10. Reducer's clean-up code snippet 15

11. Variation of execution time with number of mappers when number of reducers=1 19

12. Variation of execution time with number of mappers when number of reducers=2 20

13. Variation of execution time with number of mappers when number of reducers=3 20

14. Variation of execution time with number of mappers when number of reducers=4 21

15. Variation of execution time with number of mappers when number of reducers=5 21

16. Variation of execution time with number of mappers when number of reducers=6 22

17. Variation of execution time with number of Reducers when number of Mappers=2 24

18. Variation of execution time with number of Reducers when number of Mappers=4 25

19. Variation of execution time with number of Reducers when number of Mappers=6 25

20. Variation of execution time with number of Reducers when number of Mappers=8 26

21. Variation of execution time with number of Reducers when number of Mappers=10 26

22. Variation of execution time with number of Reducers when number of Mappers=12 27

23. Variation of execution time with number of Reducers when number of Mappers=14 27

24. Variation of execution time with number of Reducers when number of Mappers=16 28

25. Variation of execution time with number of Reducers when number of Mappers=18 28

1. INTRODUCTION

Before the Internet and Web, we did not have enough data so that it was not easy to analyze people, society, and science, etc. with the limited volumes of data. Contradicting to the past, after the Internet and web, it has been more difficult to analyze data because of its huge volumes, that is, tera- or peta-bytes of data. Google faced the issue and collected big data and the existing file systems were not sufficient to handle the data efficiently. Besides, the legacy computing power and platforms were not useful for big data. Thus, Google implemented the Google File Systems (GFS) and the MapReduce parallel computing platform, which the Apache Hadoop project is motivated from. Hadoop is the parallel programming platform built on the Hadoop Distributed File Systems (HDFS) for MapReduce computations that process data as (key, value) pairs. Hadoop has been receiving interest applied to enterprise computing because the business world always has big data such as log files for web transactions. Hadoop is useful to process such big data for business intelligence and thus has been used in data mining for the past few years. The era of Hadoop means that the legacy algorithms for sequential computing need to be redesigned or converted to MapReduce algorithms. Therefore, in this paper, a Market Basket Analysis algorithm in data mining with MapReduce is implemented and experimental results are shown using a 22-node Hadoop cluster.

Market basket analysis is one of data mining approaches to analyze the association of items for daily buying/selling. The basic idea is to find the associated pairs of items in a store from the transaction dataset. Therefore, to raise the probability of purchasing, to control the stocks more intelligently, and to promoting certain items together, the corporate manager of a shop can place associated items at the neighboring shelf. Thus, he/she can have much better chance to make profits

by controlling the order of goods and marketing. Online shopping is another form of market basket whereby consumers directly buy goods from a seller in real-time over the Internet. Similarly, an online shop, e-shop, web-shop, or virtual stores are the physical analogy of purchasing products in a shopping center. For extracting and identifying useful information from these large amounts of data, a data mining technique is crucial. The earlier proposed design based on single processor and main memory based computing and is not capable of handling peta/tera bytes of transaction data. To remove this limitation, MapReduce and Hadoop based techniques have been introduced. Hadoop is the parallel programming platform built on the Hadoop Distributed File System for MapReduce computation that processes data as <key, value> pairs. Map and Reduce are two important primitives in the MapReduce framework. In a Hadoop cluster, a master node controls a group of slave nodes on which the Map and Reduce functions run in parallel. An input file is passed to Map functions that resides on the HDFS on a cluster. Hadoop's HDFS splits the input file into fragments, which are distributed to a pool of slaves for Map/Reduce processing.

Association Rule or Affinity Analysis is the fundamental data mining analysis to find the co-occurrence relationships like the purchase behavior of customers. The analysis is fundamental in sequential computation, and many data mining books illustrate it. Aster Data has an SQL MapReduce framework as a product [1]. Aster provides nPath SQL to process big data stored in the DB. Market Basket Analysis is executed on the framework but it is based on its SQL API with MapReduce Database. As far as we understand, except in [2], there is only one way to present Market Basket Analysis algorithms with MapReduce. The experimental results that were provided in the paper are rather limited to prove the point. The approach in the paper is to propose the algorithm and to convert data to (key, value) pairs, and execute the code on Map/Reduce platform with varying numbers of mappers and reducers.

In this paper, Section 2 describes Map/Reduce and Hadoop as well as other related projects. Section 3 presents the proposed Map/Reduce algorithm for Market Basket Analysis. Section 4 shows the experimental result, Finally, Section 5 is the conclusion.

2. MAP REDUCE IN HADOOP

Map/Reduce is an algorithm is based on functional programming. It has been receiving attention since Google re-introduced it to solve the problem to analyze huge volumes of data in a distributed computing environment. It is composed of two functions to specify: “Map” and “Reduce”. They are both defined to process data structured in (key, value) pairs.

2.1. Map/Reduce in Parallel Computing

The MapReduce programming platform is implemented by the Apache Hadoop project that develops open-source software for reliable, scalable, and distributed computing. Hadoop can compose hundreds of nodes that process and compute peta- or tera-bytes of data working together. Hadoop was inspired by Google's MapReduce and GFS, as Google has had needs to process huge data for information retrieval and analysis [3]. Hadoop is used by a global community of contributors such as Yahoo, Facebook, and Twitters. Hadoop’s subprojects include Hadoop Common, HDFS, MapReduce, Avro, Chukwa, HBase, Hive, Mahout, Pig, ZooKeeper, etc. [4].

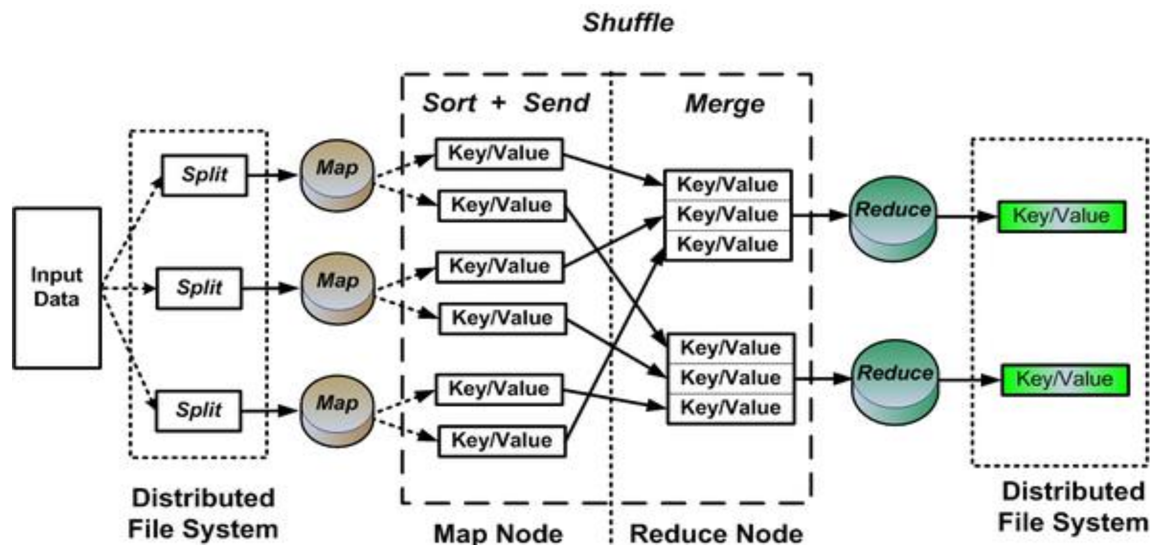

Figure 1. Map/reduce flows

The map and reduce functions run on distributed nodes in parallel. Each map operation can be processed independently on each node and all the operations can be performed in parallel. But in practice, it is limited by the data source and/or the number of CPUs near that data. The reduce functions are similar since they are from all the output of the map operations. However, MapReduce can handle significantly huge data sets since data are distributed on HDFS and operations move close to the data for better performance [5].

Hadoop is a restricted or partial parallel programming platform because it needs to collect data as (key, value) pairs as input and computes and generates the list of (key, value) as output in parallel on the mapreduce functions. In the map function, the master node parts the input into smaller sub-problems, and distributes those to worker nodes. Those worker nodes process smaller problems, and pass the answers back to their master node. That is, the map function takes inputs (k1, v1) and generates <k2, v2> where < > represents a list or set. Between the map and reduce, there is a combiner that resides on the map node, which takes inputs (k2, <v2>) and generates <k2, v2>. In the reduce function, the master node takes the answers to all sub-problems and combines them in some way to get the output, which is the answer to the problem [3, 4]. The reduce function takes inputs (k2, <v2>) and generates <k3, v3>. Figure1 illustrates the MapReduce control flow where each $value_{mn}$ is simply 1 and gets accumulated for the occurrence of items found together in the proposed Market Basket Analysis Algorithm.

2.2. Database for Big Data

Input/output files are processed on HDFS instead of using HBase DB in the paper. However, as HBase is interesting and will be integrated with the algorithm in the future, the section briefly introduces HBase.

There are some drawbacks when we use RDBMS to handle huge volumes of data, like impossible deleting, slow inserting, and random failing. HBase on HDFS is a distributed database that supports structured data storage for horizontally scalable tables. It is a column oriented semi-structured data store.

It is relatively easy to integrate with Hadoop MapReduce because HBase consists of a core map that is composed of keys and values - each key is associated with a value. Users store data rows in labeled tables. A data row has a sort able key and an arbitrary number of columns. The table is stored sparsely, so that rows in the same table can have different columns.

Using legacy programming languages such as Java, PHP, and Ruby, we can put data in the map as Java JDBC does for RDBMS. The file storage of HBase can be distributed over an array of independent nodes because it is built on HDFS. Data is replicated across some participating nodes. When the table is created, the table's column families are generated at the same time. We can retrieve data from HBase with the full column name in a certain form, and then HBase returns the result according to the given queries as SQL does in RDBMS.

2.3. Issues of Map/Reduce

The issues of Map/Reduce, although there are advantages of Map/Reduce, for some researchers and educators [6] they are:

i. A giant step backward in the programming paradigm for large-scale data intensive applications.
ii. Not new at all - it represents a specific implementation of well-known techniques developed tens of years ago, especially in Artificial Intelligence.
iii. Data should be converted to the format of (key, value) pair for MapReduce, which misses most of the features that are routinely included in current DBMS.

iv. Incompatible with all the tools or algorithms that have been built [6].

However, the issues clearly show us not only the problems but also the opportunity to implement algorithms with the MapReduce approach, especially for big data. It will give us the chance to develop new systems and evolve IT for parallel computing environment. It started a few years ago and many IT departments of companies have been moving to the MapReduce approach.

3. MARKET BASKET ANALYSIS ALGORITHM

Market Basket Analysis is one of the Data Mining approaches to analyze the association of a data set. The basic idea is to find the associated pairs of items in a store when there are transaction data as given in Figure 2 [7]. If storeowners list a pair of items that frequently occurred, s/he could control the stocks more intelligently, to arrange items on shelves and to promote items together etc. Thus, s/he should have a much better opportunity to make a profit by controlling the order of products and marketing.

```
Transaction 1: cracker, icecream, beer
Transaction 2: chicken, pizza, coke, bread
Transaction 3: baguette, soda, hering, cracker, beer
Transaction 4: bourbon, coke, turkey
Transaction 5: sardines, beer, chicken, coke
Transaction 6: apples, peppers, avocado, steak
Transaction 7: sardines, apples, peppers, avocado,
steak
...
```

Figure 2. Transaction data at a store

For example, people have built and run Market Basket Analysis codes – sequential codes – that compute the top 5 frequently occurred 3 items of transactions as in Figure 3 [7]. At the store, when customers buy a cracker and Heineken, they purchase soda as well, which happens 234 times, and hering as well 165 times. Thus, the owner can refer to the data to run the store.

```
total number of itemcount:   7008
most frequent itemcount list for:    5
cracker,heineken,soda    234
baguette,heineken,hering     214
corned_b,hering,olives   202
artichok,avocado,heineken    199
cracker,heineken,hering 165
```

Figure 3. Top 5 set of three items that occurred frequently at a store

3.1. Data Structure and Conversion

The data in Figure 2 [7] is composed of the list of transactions with its transaction number and the list of products. For a MapReduce operation, the data set should be structured with (key, value) pairs. The simplest way used in the paper is to pair the items as a key and the number of key occurrences as its value in the basket, especially for all transactions, without the transaction numbers. Thus, Figure 2 [7] can be restructured as Figure 4 [7] assuming collecting pairs of items in order 2 – two items as a key.

< (cracker, icecream), (cracker, beer) >
< (chicken, pizza), (chicken, coke), (chicken, bread) >
< (baguette, soda), (baguette, hering), (baguette, cracker), (baguette, beer) >
< (bourbon, coke), (bourbon, turkey) >
< (sardines, beer), (sardines, chicken), (sardines, coke) >
...

Figure 4. Data Set restructured for Map/Reduce

However, if we select the two items in a basket as a key, there should be incorrect counting for the occurrence of the items in the pairs. As shown in Figure 5 [7] , transactions n and m have the items (cracker, icecream, beer) and (icecream, beer, cracker), which have the same items but in a different order.

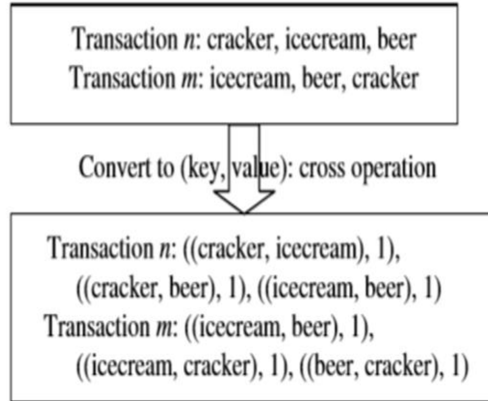

Figure 5. Data Set Restructured for the same list

That is, for (cracker, icecream, beer), the possible pair of items in (key, value) are ((cracker, icecream), 1), ((cracker, beer), 1), ((icecream, beer), 1). And, for (icecream, beer, cracker), the possible pair of items are ((icecream, beer), 1), ((icecream, cracker), 1), ((beer, cracker), 1). Therefore, we have a total of SIX different pair of items that occur only once respectively, which should be THREE different pairs. That is, keys (cracker, icecream) and (icecream, cracker) are not the same even though they are, which is not correct. We can avoid this issue if we sort the transaction in alphabetical order before generating (key, value) as shown in Figure 6 [1]. Now each transaction has the following THREE pair of items ((beer, cracker), 1), ((beer, icecream), 1), ((cracker, icecream), 1). That is TWO different pair of items that occur twice respectively so that we accumulate the value of the occurrence for these two transactions as follows: ((beer, cracker), 2), ((beer, icecream), 2), ((cracker, icecream), 2), which is correct to count the total number of occurrences.

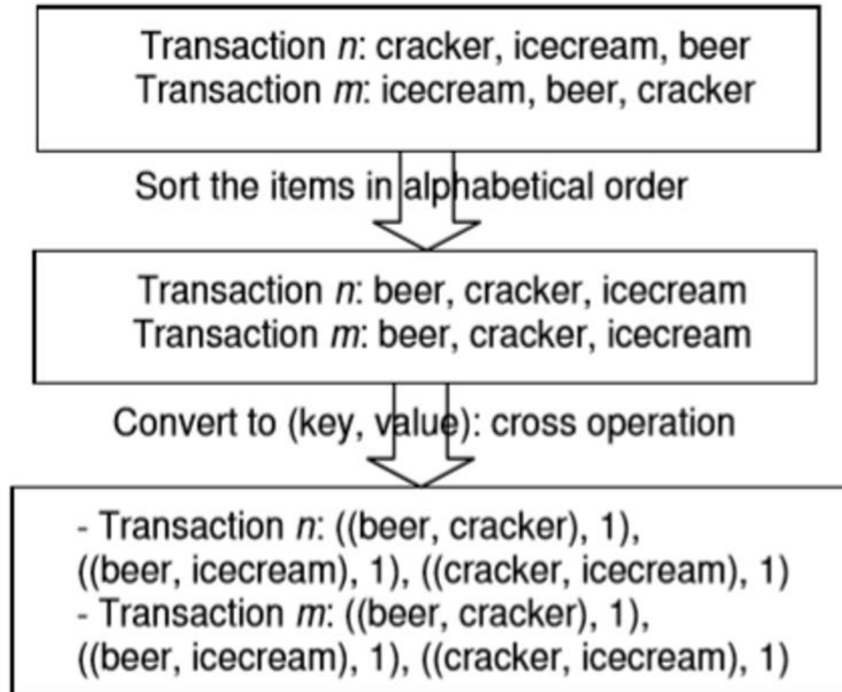

Figure 6. Data Set Restructured with Sort

3.2. The Algorithm

The Market Basket Analysis (MBA) algorithm [9] for mapper and reducer are illustrated below. The mapper reads the input data and creates a list of items for each transaction. As a mapper of a node reads each transaction on Hadoop, it assigns mappers to number of nodes, where the assigning operation in Hadoop is hidden to us. For each transaction, its time complexity is O(n) where n is the number of items for a transaction. Then, the items in the list are sorted to avoid duplicated keys as shown in Figures 5 and 6 [7]. Its time complexity is O(n log n) on merge sort. Then, the sorted items should be converted to pairs of items as keys, which is a cross-operation in order to generate cross pairs of the items in the list as shown in Figures 5 and 6 [7]. Its time complexity is O(n x m) where m is the number of pairs that occurs together in the transaction. Thus, the time complexity of each mapper is O(n + n log n + n x m).

The steps of the mapper are as follows:

a) Read each transaction of input file and generate the data set of the items: (<V1>, <V2>, …, <Vn>) where < Vn>: (vn1, vn2,.. vnm)

b) Sort all data set <Vn> and generates sorted data set <Un>: (<U1>, <U2>, …, <Un>) where < Un>: (un1, un2,.. unm).

c) Loop While < Un> has the next element;   note: each list Un is handled individually.

    i. Loop For each item from un1 to unm of < Un> with NUM_OF_PAIRS

        1. Generate the data set <Yn>: (yn1, yn2,.. ynl);   ynl: (unx, uny) is the list of self-crossed pairs of (un1, un2,.. unm) where unx != uny

        2. Increment the occurrence of ynl; note: (key, value) = (ynl, number of occurrences)

    ii. End Loop For

d) End Loop While

e) Data set is created as input of Reducer: (key, <value>) = (ynl, <number of occurrences>)

The reducer is to accumulate the number of values per key. Thus, its time complexity is O(v) where v is the number of values per key.

i. Read (ynl, <number of occurrences>) data from multiple nodes

    i. Add the values for ynl to have (ynl, total number of occurrences)

3.3. Market Basket Analysis Algorithm Implementation using HDFS

The transaction input data need to be loaded into the Hadoop Distributed File System (HDFS). HDFS divides the data into different blocks (based on the block size, default 128MB)

and stores on the data nodes. The mapper uses the data in each block as an input. The input split is calculated based on the InputFormat specified. And for each input split one mapper is launched. While launching the mappers, the framework takes care of data locality. The default InputFormat is TextInputFormat. It reads each transaction data in the input split and passes the offset and transaction data as the key value pair to the "map" function of the mapper. The "map" function performs the following operations:

a. Split transaction data received as value in "map" function and generates the data set of the items.

b. Sort the generated data set of items and generates sorted data set. This step is performed to remove the duplication of items (e.g (coke,pepsi) and (pepsi,coke) should be considered same).

c. Now, on the sorted data set of items, we calculate all the combinations of user defined length. And output each combination as key and value 1 as the mapper output. The mapper output will be given as input to reducer after sorting and shuffling all mappers output.

In our code MBAMapper.java is the mapper class. Figure 7 is the code for the "map" function.

```
public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException{

    String tranList[] = (value.toString()).toLowerCase().split(" ");
    Arrays.sort(tranList);
    int arraySize = tranList.length;
    itemCount = itemCount + arraySize;
    //System.out.println("starting the calculation of combination" + value.toString());
    calculateCombination(tranList, arraySize, combinationValue , context);
    /*for (int itemListNo = 0; itemListNo < arraySize; itemListNo++) {
        int nextItemNum = itemListNo+1;
        while(nextItemNum < arraySize){
            context.write(new Text(tranList[itemListNo]+","+tranList[nextItemNum]), new IntWritable(1));
            nextItemNum++;
        }
    }*/
}
```

Figure 7. Map function

The “calculateCombination” function writes the output of the mapper to the context using the code in Figure 8:

```
context.write(new Text("total number of itemcount:"), new IntWritable(itemCount));
```

Figure 8. Calculate combination function

Once the mappers finish the execution, the framework starts the shuffling and sorting phase. After shuffling and sorting, all the values related to one key will be given to one reducer. The “reduce” method of the reducer class will receive the key and iterable values. The “reduce” function performs the following operations:

a. Get all values related to one key as iterable. It counts the values for each combination by iterating over the Iterable values.

b. After calculating the value related to the key, the reducer output the user-defined number of top items.

In our code MBAReducer.java is the Reducer class. Figure 9 shows the code for 'reduce' function:

```java
public void reduce (Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException{

    int count = 0;
    Iterator<IntWritable> it =values.iterator();
    while (it.hasNext()) {
        IntWritable intWritable = (IntWritable) it.next();
        count = count+intWritable.get();
    }
    if(key.toString().equals("total number of itemcount:")){
        totalItemCount = totalItemCount + count;
    }else {
        if(topNItems.size()<noOfTopItems){
            topNItems.put(key.toString(), count);
        }
        else{
            if(flag){
            sortedMapItems = sortByValues(topNItems);
            minKey = sortedMapItems.keySet().iterator().next();
            minValue = sortedMapItems.get(minKey);
            flag = false;
            }
            if(count > minValue){
                sortedMapItems.remove(minKey);
                sortedMapItems.put(key.toString(), count);
                sortedMapItems = sortByValues(sortedMapItems);
                minKey = sortedMapItems.keySet().iterator().next();
                minValue = sortedMapItems.get(minKey);
            }
        }
    }
}
```

Figure 9. Reduce function

After all the calculations, the "cleanup" method writes the final result to the context, i.e the final output is written to the HDFS as shown in Figure 10.

```java
protected void cleanup(Context context) throws IOException, InterruptedException {

    context.write(new Text("total number of itemcount:"), new IntWritable(totalItemCount));
    //Map<String, Integer> sortedMap = sortByValues(topNItems);
    int counter = 0;
    context.write(new Text("most frequent itemcount list for: "), new IntWritable(noOfTopItems) );
    for(String key : sortedMapItems.keySet()) {
        if(counter ++ == noOfTopItems) {
            break;
        }
        context.write(new Text(key), new IntWritable(sortedMapItems.get(key)));
    }

}
```

Figure 10. Reducer's clean-up code snippet

Also, we have the MBADriver.java class, which takes care of the entire configuration related things. And it also launches the job and starts the execution.

3.4. Code Execution

The Untitled.jar code is run on the Hadoop 2.7.1 cluster and executed in the stand-alone and clustered modes. The code generates the top 5 associated items that customers purchased together as shown in Figure 3. A total of four different types of inputs were used known to be input.txt, input1.txt, input2.txt with data size of 5.1GB, 298MB, and 0.59GB, respectively.

4. EXPERIMENTAL RESULTS

There are many variables, which can affect the execution time of a MapReduce algorithm. The most important factors that affect the execution time are:

1. Number of mappers used during the mapping phase.
2. Number of reducers used during the reducing phase.
3. Amount of data for which the algorithm is being executed on.

To get a clear understanding how these parameters affect the execution time, I have taken three different datasets 5.1GB, 298MB, 0.59GB and executed the MBA algorithm on these data sets by varying the number of mappers from 2 to 18, and the number of reducers from 1 to 6. After executing the code on the three different data sets, we can analyze the results by calculating the execution time obtained by varying the number of mappers and reducers. The total execution time here is the summation of the time taken by the mappers and the reducers. Here I am analyzing all results by keeping one of the variables constant.

Before going to the evaluation of the experimental results we need to talk about how the data is being distributed among these mappers and reducers.

4.1. Data Distribution Between Mappers

The amount of data sent to each mapper vary and we can control how much of input data needs to go through each mapper. There are three factors, which control the amount of data space assigned for each mapper or we can say the number of blocks assigned to each mapper. They are:

i. Block size in the Hadoop cluster: 128MB (since the Hadoop version we are using on the Hadoop cluster is 2.7.1).
ii. Number of Mappers: 2 to 18.

iii. Input split size: varying with the amount of data.

We can calculate the number of blocks assigned to each mapper in two steps:

i. Amount of input split data assigned to each mapper.

ii. Number of blocks per mapper.

#Amount of Input split data assigned to Mapper = amount of split-able input data/number of input splits.

#Number of blocks assigned to each mapper = amount of input split data assigned to mapper/128MB.

Table 1. Data blocks distribution for each Mapper

Number of Mappers / Input Data (MB)	**297.47**	**580.36**	**5105.45**
2	2	3	20
4	1	2	10
6	1	1	7
8	1	1	5
10	1	1	4
12	1	1	4
14	1	1	3
16	1	1	3
18	1	1	3

4.1.1. Important Observations

1. Table 1 clearly shows why the Hadoop framework is mainly useful for huge amounts of data. We can see that even though the amount of data available is less (less than 128MB), for each mapper there is a dedicated block for the data. This causes inefficient utilization

of memory, delay in total execution time, unavailability of resources to other MapReduce jobs, performance issues, ineffective parallel processing, etc.

2. Whereas when it comes to datasets with large quantities of data (in this specific case 5.1GB), each mapper is being provided with at least 3 blocks of data keeping every mapper busy by effective parallel processing.

4.2. Data Distribution Between Reducers

In this case, we cannot exactly determine how much amount of data is being send to each reducer unlike here the input for each reducer comes from the mappers as output. But what we know for a fact is, all the <key,value> pairs with the same key value goes to the same reducer.

4.3. Varying Number of Mappers

We observe how the execution time varies, when we vary the number of mappers, we should keep the number of reducers constant.

Figure 11 shows the experimental results obtained when the reducer=1.

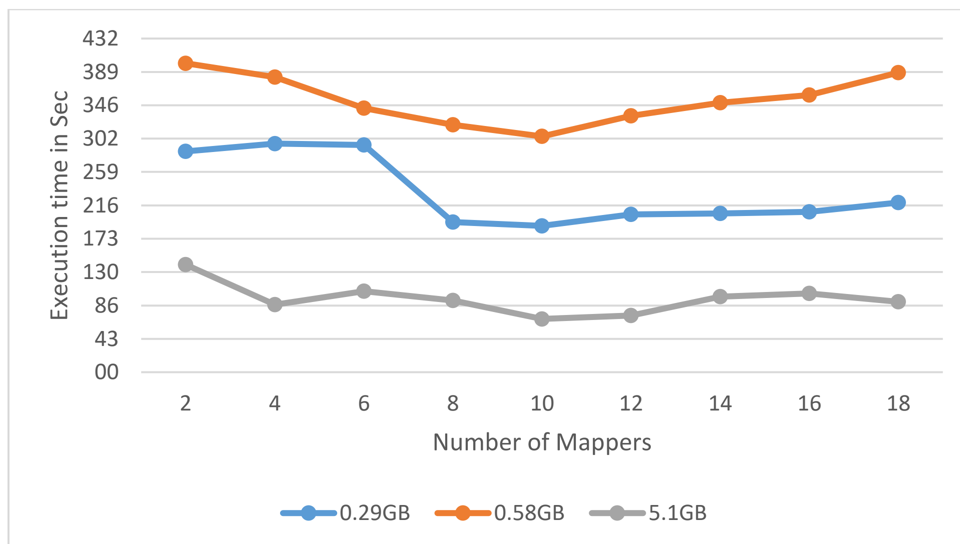

Figure 11. Variation of execution time with number of mappers when number of reducers=1

Figure 12 shows the experimental results obtained when the reducers=2.

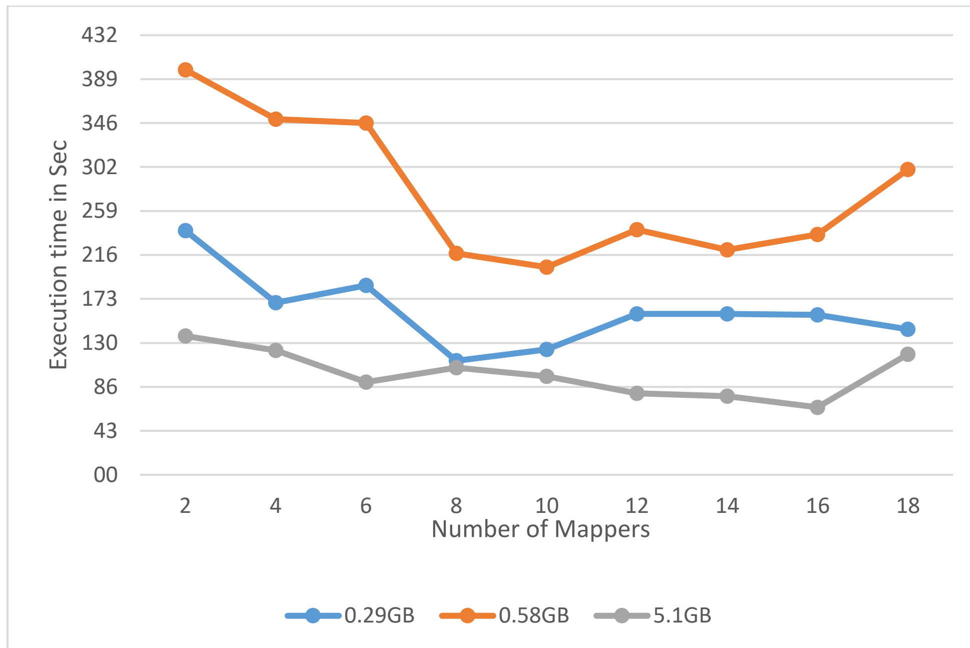

Figure 12. Variation of execution time with number of mappers when number of reducers=2

Figure 13 shows the experimental results obtained when the reducers=3.

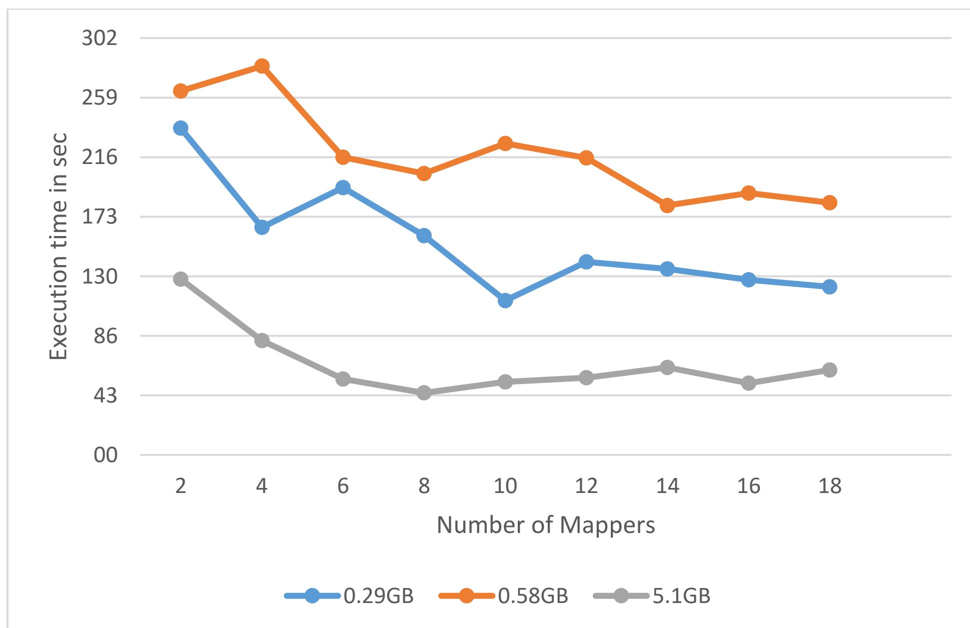

Figure 13. Variation of execution time with number of mappers when number of reducers=3

Figure 14 shows the experimental results obtained when the reducers=4.

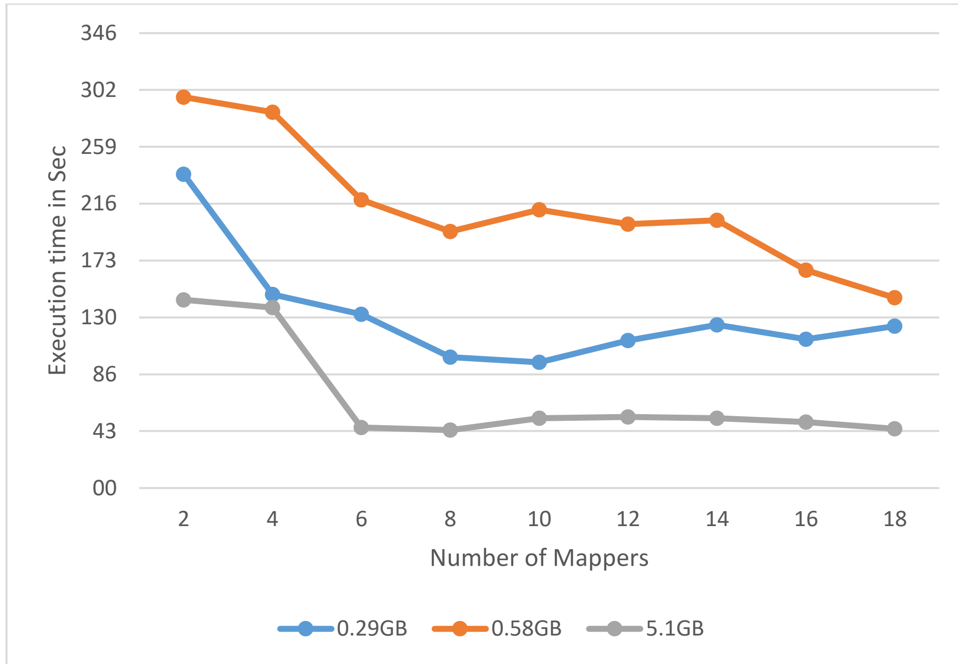

Figure 14. Variation of execution time with number of mappers when number of reducers=4

Figure 15 shows the experimental results obtained when the reducers=5.

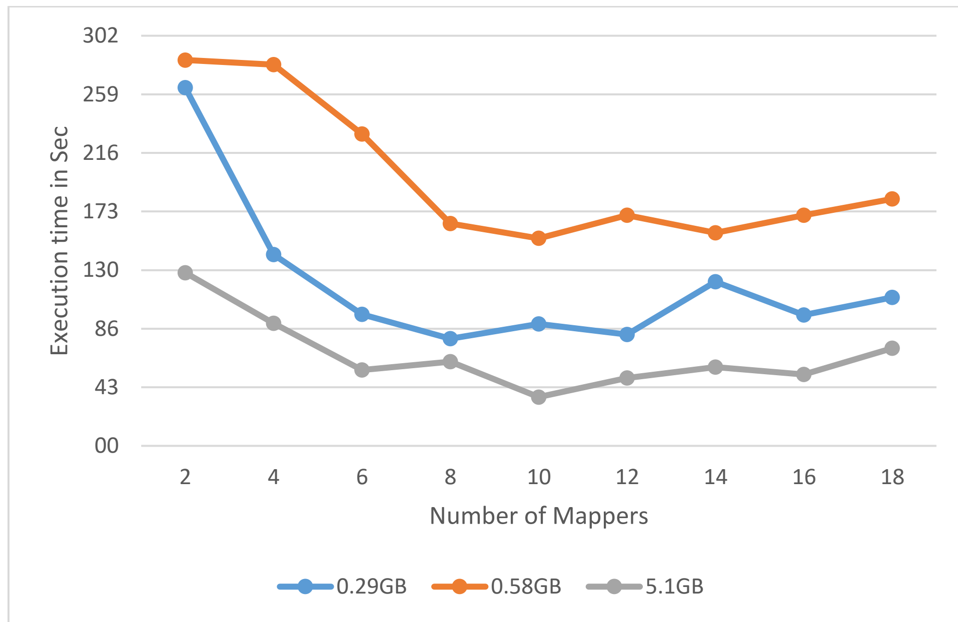

Figure 15. Variation of execution time with number of mappers when number of reducers=5

Figure 16 shows the experimental results obtained when the reducers=6.

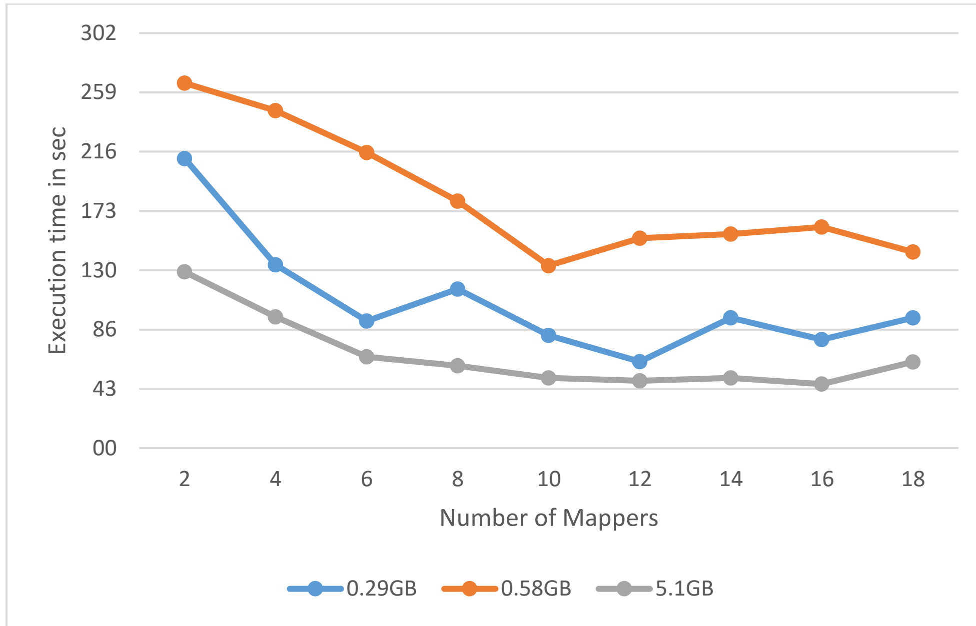

Figure 16. Variation of execution time with number of mappers when number of reducers=6

4.3.1. Important Observations

- As we increase the number of mappers to process the datasets, the execution time significantly dropped until a certain point and then it tends to increase but the difference is not as significant as we have seen previously [Table 2].

Table 2. Variation of execution time due to increase in number of Mappers

Number of reducers is 2	0.29GB	difference	0.58GB	difference	5.1GB	difference
mappers =2	240sec		398sec		136sec	
mappers =10	123sec	117sec	204sec	194sec	97sec	40sec
mappers=18	143sec	20sec	300sec	96sec	118sec	21sec

From all figures, we can observe that the execution time taken by the algorithm when we are using 2 mappers is much higher than when it is running on 10 mappers. If we take a particular

case with number of reducers is 2, We can clearly observe from the difference column of Table 2, the difference in execution time like for 5.1GB dataset, when the number of mappers increased from 2 to 10 and then from 10 to 18, the difference in execution time is 40 sec and 21 sec.

- No matter what the number of nodes or number of mappers, the execution time taken at any point is always less than the time taken by the algorithm while executing on 2 mappers (considering that we are running our algorithm on a multi-node Hadoop cluster).

  When the number of reducers is 6, the execution time for the 5.1GB data set is 128 sec (no. of mappers=2) > 63 sec (no. of mappers=18).

- We cannot neglect the fact that the execution time taken by the 5.1GB data set is a very low when compared to the other two data sets even though the amount of data is very high.

  We can observe that for almost all nodes, the execution time taken by the 5.1GB data set is significantly lower than the other two data sets.

- We should discuss about the execution time taken by the 0.58GB data set since it is the one which showed peculiar behavior. As we have observed the execution time taken by this data set is significantly higher than the 0.29GB data set and it is even higher than the 5.1GB data set.

  Here the execution time taken by the 0.29GB, 0.58GB, 5.1GB data set when the number of reducers are 5 and the mappers are 6 are 194 sec, 230 sec, 56 sec, respectively. When the same algorithm is run on 1.54GB, it took a total of approximately 836 sec. From these results, we can say that the execution time also has a specific pattern when we start increasing the amount of data.

The execution time tends to increase when we increase the data on which we are about to run the code and reaches a threshold limit. After reaching the threshold limit, it drops exponentially (from 836 sec to 56 sec).

4.4. Varying number of Reducers

We can observe how the execution time varies by varying the number of reducers after keeping the number of mappers constant.

Figure 17 shows the experimental results obtained when the Mappers=2.

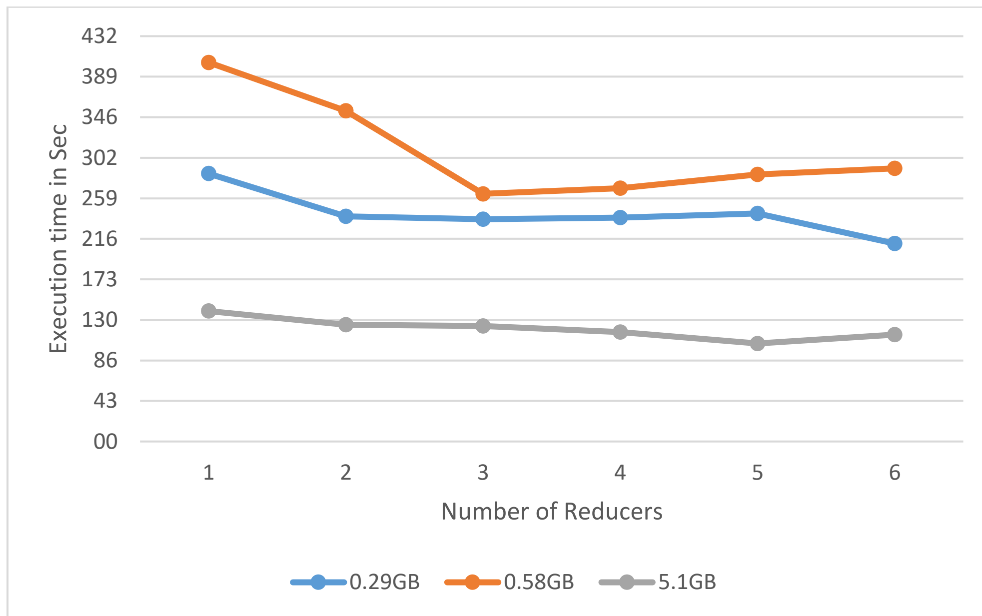

Figure 17. Variation of execution time with number of Reducers when number of Mappers=2

Figure 18 shows the experimental results obtained when the Mappers=4.

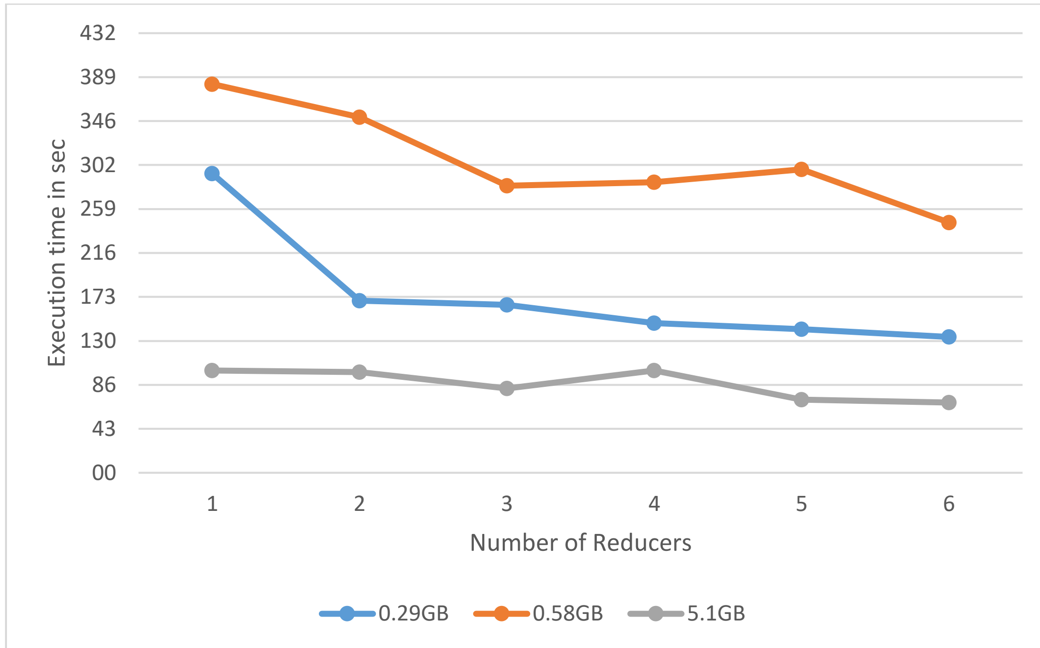

Figure 18. Variation of execution time with number of Reducers when number of Mappers=4

Figure 19 shows the experimental results obtained when the Mappers=6.

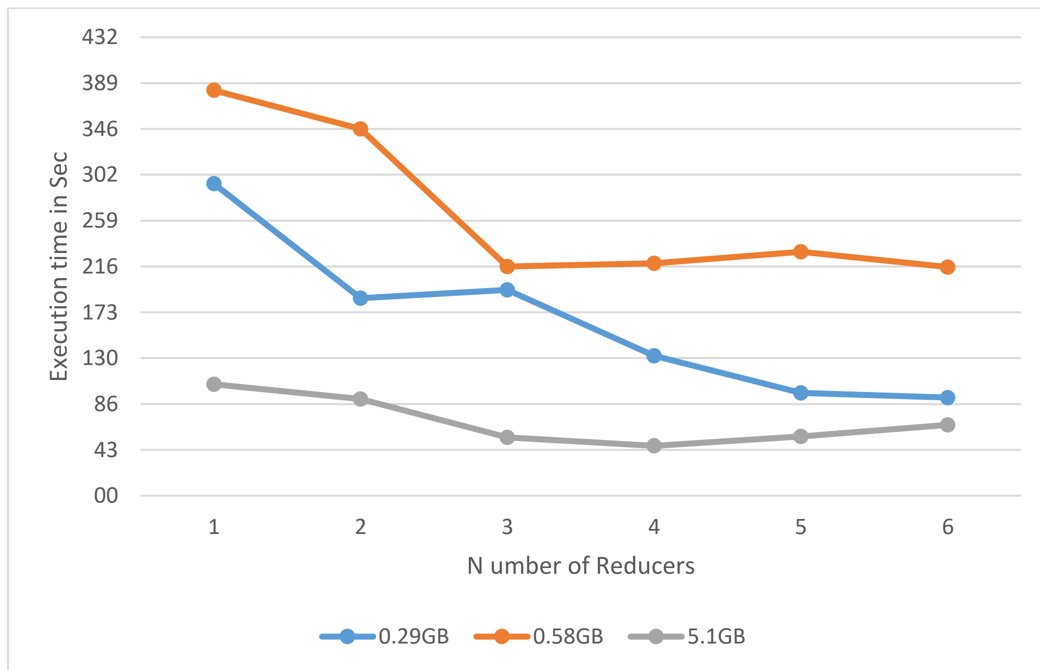

Figure 19. Variation of execution time with number of Reducers when number of Mappers=6

Figure 20 shows the experimental results obtained when the Mappers=8.

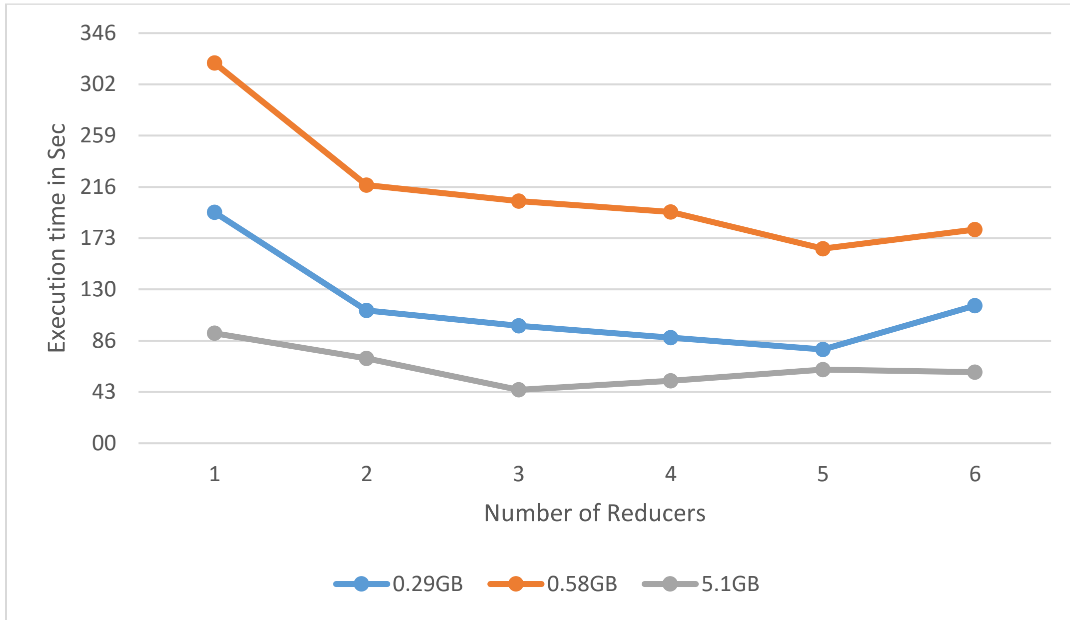

Figure 20. Variation of execution time with number of Reducers when number of Mappers=8

Figure 21 shows the experimental results obtained when the Mappers=10.

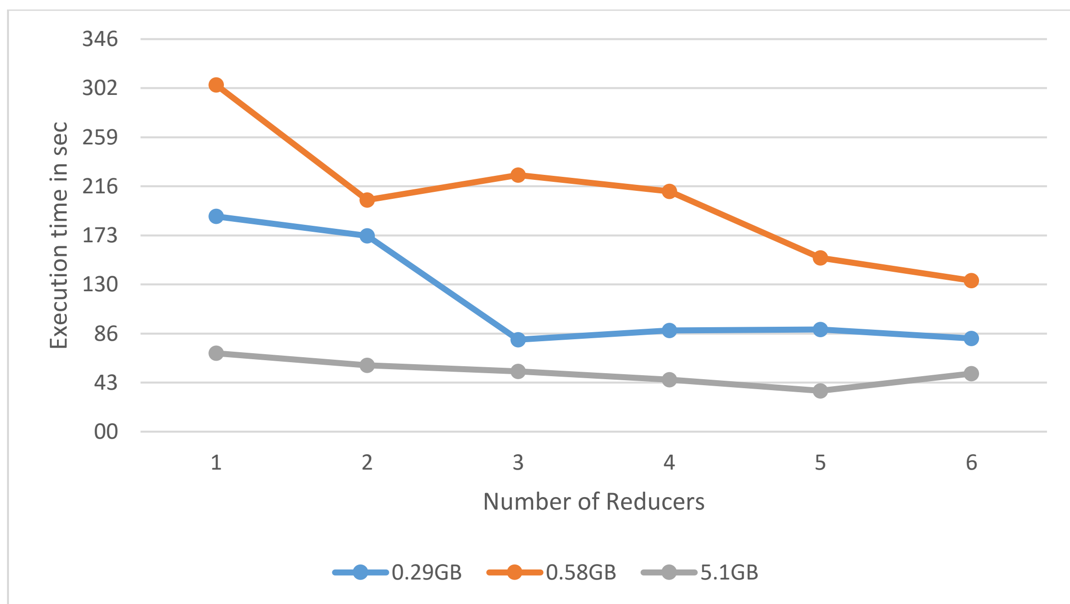

Figure 21. Variation of execution time with number of Reducers when number of Mappers=10

Figure 22 shows the experimental results obtained when the Mappers=12.

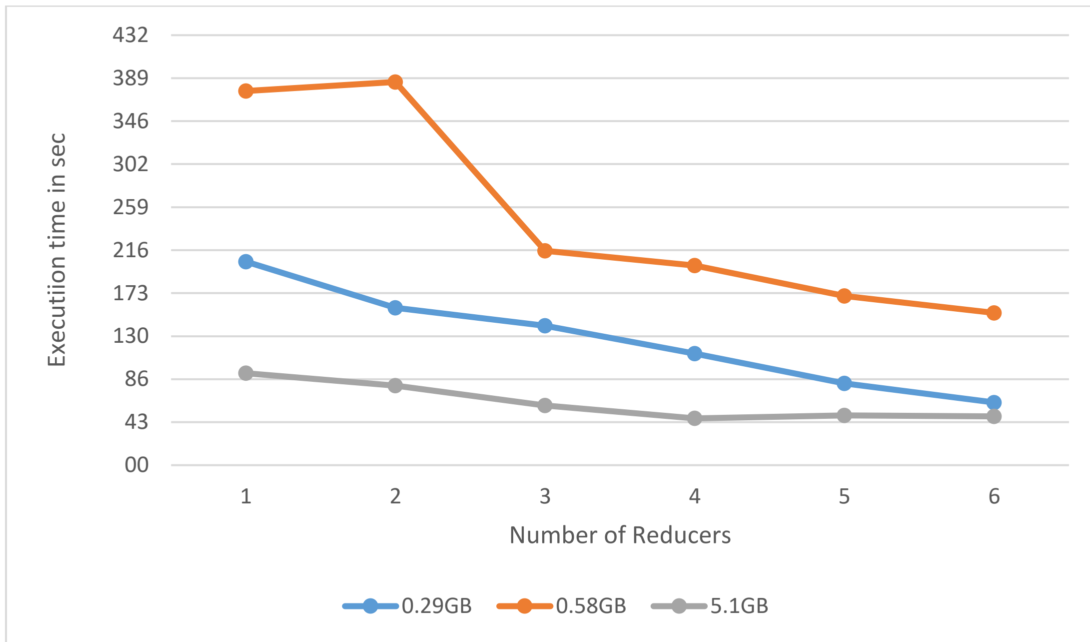

Figure 22. Variation of execution time with number of Reducers when number of Mappers=12

Figure 23 shows the experimental results obtained when the Mappers=14.

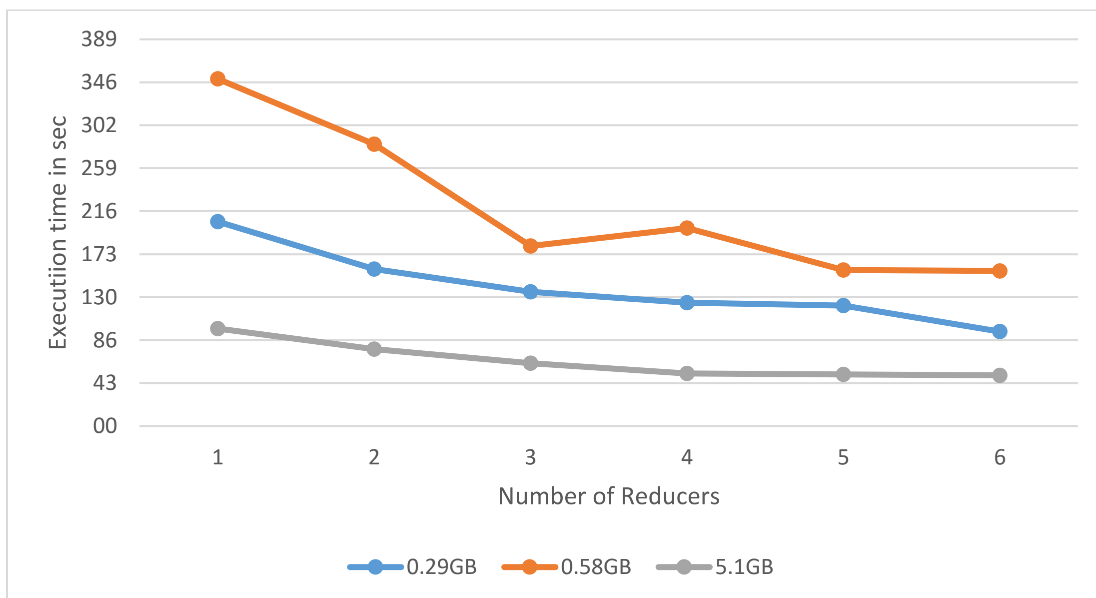

Figure 23. Variation of execution time with number of Reducers when number of Mappers=14

Figure 24 shows the experimental results obtained when the Mappers=16.

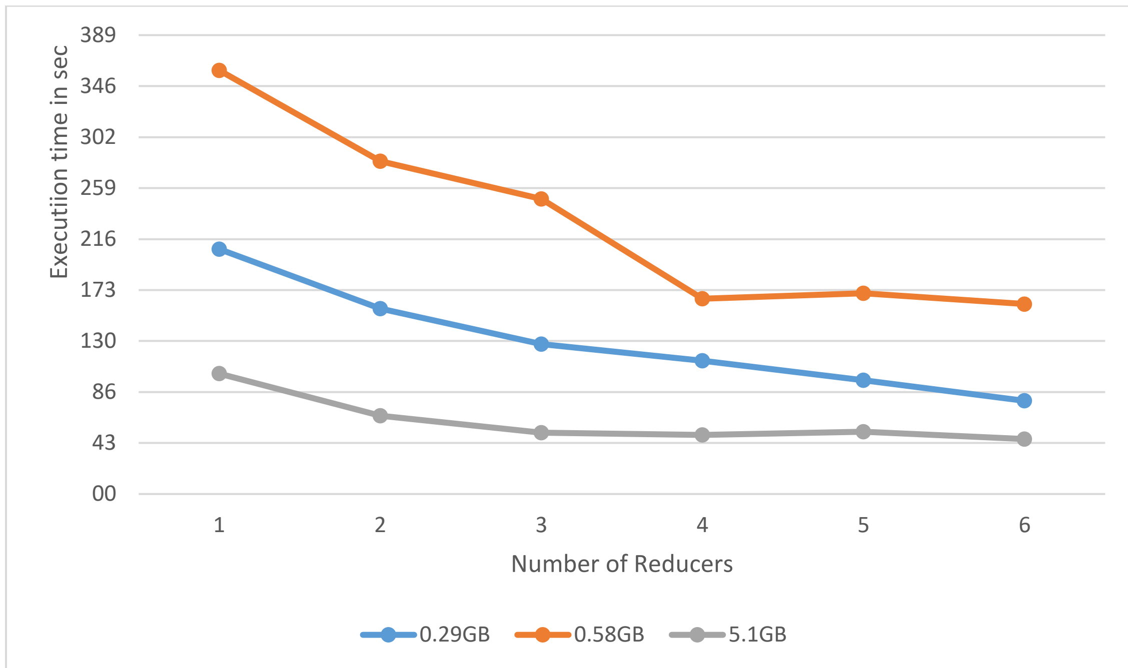

Figure 24. Variation of execution time with number of Reducers when number of Mappers=16

Figure 25 shows the experimental results obtained when the Mappers=18.

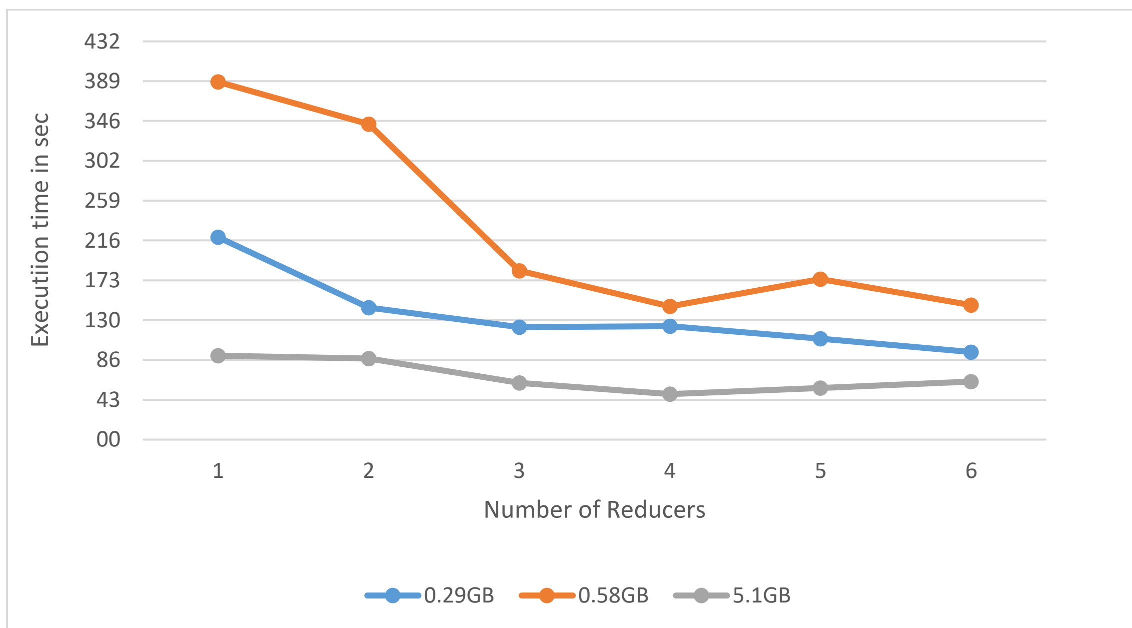

Figure 25. Variation of execution time with number of Reducers when number of Mappers=18

4.4.1. Important Observations:

- As we increase the number of reducers to process the datasets, the execution time significantly dropped until a certain point and then it tends to increase but the difference is not as significant as we have seen previously [Table 3].

Table 3. Variation of execution time due to increase in number of reducers

number of mappers is 8	0.29GB	difference	0.58GB	difference	5.1GB	difference
Reducers=1	195sec		320sec		93sec	
Reducers=3	99sec	96sec	204sec	116sec	45sec	48sec
Reducers=6	116sec	17sec	180sec	-24sec	60sec	15sec

From all figures, we can observe that the execution time taken by the algorithm on the 5.1GB dataset when we are using 1 Reducer is relatively higher than when it is running with 3 reducers. However, it is not the same since the data distribution between the reducers is not the same as compared to the mappers. Here all the <key,Value> pairs that have the same key value go to the same reducer. So, we can make an observation that when the data set is relatively small then the execution time decreases.

- No matter what the number of nodes or number of reducers, the execution time taken at any point is always less than the time taken by the algorithm while executing on 1 reducer (considering that we are running our algorithm on a multi-node Hadoop cluster).

  When the number of mappers is 6, the execution time for the 5.1GB data set is 105 sec (no. of reducers=1) > 67 sec (no. of reducers=6).

- We cannot even neglect the fact that the execution time taken by the 5.1GB data set is a very low when compared to the other two data sets even though the amount of data is very high.

We can observe that for almost all nodes, the execution time taken by the 5.1GB data set is significantly lower than the other two data sets.

- We should discuss about the execution time taken by the 0.58GB data set since it is the one that shows a peculiar behavior. As we have observed the execution time taken by this data set is significantly higher than the 0.29GB data set and it is even higher than the 5.1GB data set. Here the execution time taken by the 0.29GB, 0.58GB, 5.1GB data set when the number of reducers are 5 and the mappers are 6 are 97sec, 230 sec, 56 sec, respectively. When the same algorithm is run on 1.54GB, it took a total of approximately 836 sec. From these results, we can say that the execution time also has a specific pattern when we start increasing the amount of data.

  The execution time tends to increase when we increase the data on which we are about to run the code and reaches a threshold limit. After reaching the threshold limit, it drops exponentially (from 836sec to 56sec).

4.5. Other Factors affecting the Execution Time

Some of the factors that do impact the execution time while implementing an algorithm on the MapReduce framework [10] are: speculative execution, nodes with poor performance, data locality and distribution of data blocks, and parallelism control with input split size.

4.5.1. Speculative execution

Speculative execution is a strategy of Hadoop that provides fault tolerance and reduces job execution time. When a TaskTracker (a daemon process running on slave nodes that executes map and reduce tasks) performs poorly or crashes, the JobTracker (a demon process running on a master node that accepts the job and submit tasks to TaskTrackers) launches another backup task on

another nodes to accomplish the task faster and successfully. This process of redundant execution is called speculative execution and the backup copy of task is called speculative task. Among the original and speculative tasks which one completes first is kept while other is killed since it's no longer needed. Speculative execution is good at most of the time but it affects cluster efficiency by duplicating tasks.

4.5.2. Nodes with poor performance

Nodes in a heterogeneous cluster are of different capability. Heterogeneity may arise due to differences in hardware as well as using virtualization technology. Virtualization facilitates efficient utilization of resources and environments for different operating systems. There are many benefits of VM-hosted (virtual machine hosted) Hadoop such as lower cost of installation, on demand cluster setup, reuse of remaining physical infrastructure, on demand expansion and contraction of cluster size. However, virtualization provides efficient re-use and management of resources but on the cost of performance. Virtual machines are slower in comparison to physical machines. Even when we execute the same algorithm on the same amount of data using the same number of mappers and reducers, taking two nodes might not give the same execution time.

4.5.3. Data locality and distribution of data blocks

HDFS breaks a large file into smaller data blocks of 128 MB and stores 3 replicated copies on data nodes (DNs) of the cluster. Data blocks are the physical division of data. Data files can be logically divided using input split. The number of mappers to be run for a job is equal to the number of logical splits. If input split is not defined, then the default block size is the split size. MapReduce processes a data block locally on the DataNode where the block is present. For block distribution, all Mappers are running on the same DataNode since all blocks are locally available to each node.

In different attempt of running a job, DataNode may be different each time but all the Mappers are being run on a same DataNode. All Mappers running on the same DataNode does not make use of available resources, which leads to increased execution time. Here it can be seen that due to higher replication factor data locality may be a hurdle that slow down the execution.

4.5.4. Parallelism control with input split size

Hadoop is designed to process big datasets that does not mean one cannot benefit from small datasets. To reduce the execution time we need more than one task running in parallel. Split is used to control the number of map tasks for a MapReduce job. A split may consist of multiple blocks and there may be multiple splits for a single block. So, without changing the block size, the user can control the number of mappers to be run for a particular job.

Smaller split size launches more number of mappers, which consequently increase the parallelism. Increasing the number of mappers beyond a particular point starts to degrade the performance due to unnecessary overheads and shortage of resources. To achieve the right level of parallelism it must be taken care that the map task is CPU-intensive or CPU-light as well as the size of dataset to be processed.

5. CONCLUSION

Hadoop with MapReduce motivates the needs to propose new algorithms for the existing applications for which the algorithms were implemented for sequential computation. Besides, it is a (key, value) based restricted parallel computing paradigm that allow sequential implementations to be redesigned with MapReduce to parallelize these algorithms.

In this paper, the Market Basket Analysis algorithm on MapReduce is presented, which is association based data mining analysis to find the most frequently occurred group of products in the baskets at a store. The user is given the capability of choosing how many top transactions and how many items needed to be in that group. The data set shows that associated items can be grouped with the MapReduce approach.

This algorithm has been executed on the 22-Node Hadoop cluster at North Dakota State University varying the number of mappers from 2 to 18, and the number of reducers from 1 to 6 on three different sizes of data. The execution times of the three different data sets show that processing large amount of data is way faster when compared to smaller amount of data. The execution times of the test cases show that the proposed algorithm gets better performance while running on large number of mappers and reducers until certain point. However, from a certain point, MapReduce does not guarantee to increase the performance even though we add more mappers or reducers because there is a bottle-neck for distributing, aggregating, and reducing the data set among the nodes against the computing powers of additional nodes.

6. REFERENCES

1. Lam, C. (2010). *Hadoop in action*. Manning Publications Co.
2. Woo, J., & Xu, Y. (2011, July). Market basket analysis algorithm with Map/Reduce of cloud computing. *In The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2011), Las Vegas.*
3. Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113
4. Apache Hadoop. (n.d.). In *Wikipedia.* Retrieved September 10, 2014, from https://en.wikipedia.org/wiki/Apache_Hadoop
5. Lin, J., & Dyer, C. (2010). Data-intensive text processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, *3*(1), 1-177.
6. Kim, W. (2010). MapReduce Debates and Schema-Free. *Coord, March*, *3*.
7. Woo, J., & Xu, Y. (2011, July). Market basket analysis algorithm with Map/Reduce of cloud computing. In *The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2011), Las Vegas*.
8. Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval* (Vol. 1, No. 1, p. 496). Cambridge: Cambridge university press.
9. Woo, J. (2013). Market basket analysis algorithms with mapreduce. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, *3*(6), 445-452
10. Singh, S., Garg, R., & Mishra, P. K. (2016, April). Observations on factors affecting performance of MapReduce based Apriori on Hadoop cluster. In *Computing, Communication and Automation (ICCCA), 2016 International Conference on* (pp. 87-94). IEEE.

11. Stephens, B. (2009). Building a business on an open source distributed computing. In *Oreilly Open Source Convention (OSCON)* (Vol. 2009, pp. 20-24).

12. Woo, J., Basopia, S., Xu, Y., & Kim, S. H. (2011, August). Market Basket Analysis Algorithm with NoSQL DB HBase and Hadoop. In *The Third International Conference on Emerging Databases (EDB 2011), Songdo Park Hotel, Incheon, Korea*.

13. Woo, J. (2011). *The Technical Demand of Cloud Computing*. Korean Technical Report of KISTI (Korea Institute of Science and Technical Information).

14. Woo, J. (2012, January). Apriori-Map/Reduce Algorithm. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)* (p. 1). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

15. Lin, J., & Dyer, C. (2010). Data-intensive text processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, *3*(1), 1-177.

16. Muppidi, S., & Murty, M. R. (2015, January) Document Clustering with Map Reduce using Hadoop Framework. *International Journal on Recent and Innovation Trends in Computing and Communication (Volume: 3 Issue: 1).* Available @ http://www.ijritcc.org

17. P. Sharma, V. Garg, R. Kaur, S. Sonare , (2013). Big Data in Cloud Environment. International Journal of Computer Sciences and Engineering, 1(3), 15-17.