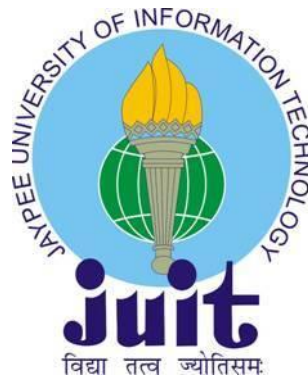


Minimum Spanning Tree Algorithms in MapReduce Framework on Hadoop

Enrollment Number - 122201

Name of Student - Mohit Sharma

Name of supervisor -Mr. Suman Saha



May 2014

Submitted in partial fulfillment of the Degree of
Master of Technology

DEPARTMENT OF COMPUTER SCIENCE,
JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY,
WAKNAGHAT, SOLAN, H.P., INDIA

CERTIFICATE

This is to certify that the work titled “**Minimum Spanning Tree in MapReduce Framework on Hadoop**” submitted by “**Mohit Sharma**” in partial fulfillment for the award of degree of **M.Tech** of Jaypee University of Information Technology, Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor *Suman Saha*
Name of Supervisor Mr. Suman Saha
Designation Assistant Professor
Date *23-05-14*

ACKNOWLEDGEMENT

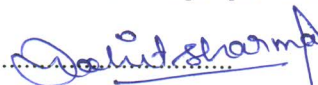
This may seem long but the task of my thesis work both theoretically and practically may not have been completed without the help, guidance and mental support of the following persons.

Firstly, I would like to thank my guide Assistant Professor, Department of Computer Science & Engineering, Jaypee University of Information Technology, Waknaghat, **Mr. Suman Saha** sir who provided me the idea and related material for the project proposal. He indeed guided me to do the task for my thesis in such a way that it seems to be research work. His continuous monitoring to support me and my research work encouraged me a lot for doing my thesis in very smooth manner.

Secondly, I would like to thank my friends who have always been with me for inspiring me that I can do the good thesis with the hard work. Their instigation always helped me to grow my mind focused towards the hard work for thesis with having the research work in the mind.

Thirdly, I would like to thank God for keeping me energetic, healthy and enthusiastic every time due to which I could complete my thesis work successfully.

Once again thank a ton to all mentioned people in my life.

Signature of the Student: 

Name of the Student: Mohit Sharma

Date: 25/05/14

TABLE OF CONTENTS

Chapter Number	Topic Name	Page Number
	Certificate	IV
	Acknowledgement.....	V
	Abstract.....	VI
	List of Figures.....	VII
Chapter 1: INTRODUCTION.....		1
1.1	Introduction.....	1
1.2	Motivation.....	3
1.3	Problem Statement.....	4
1.4	Organization of the Thesis.....	5
Chapter 2: LITERATURE REVIEW.....		6
2.1	Literature Review on Cloud Computing.....	6
2.1.1	Cloud Computing.....	6
2.1.2	Cloud Computing services model.....	6
2.2	Literature Review on MapReduce Framework.....	7
2.2.1	Bounds in MapReduce.....	10
2.2.2	MapReduce class.....	11
2.2.3	Deterministic MapReduce class.....	12

2.2.4 MapReduce example.....	12
2.2.5 Hadoop.....	14
2.2.6 HDFS.....	16
2.2.7 Related projects.....	20
2.3 Literature Review on Graph algorithms in MapReduce Framework.....	21
2.4 Literature Review on Minimum Spanning Tree algorithms.....	22
2.4.1 Sequential minimum spanning tree algorithms.....	23
2.4.2 Parallel minimum spanning tree algorithms.....	26
2.5 Literature Review on MST in MapReduce Framework.....	26
Chapter 3: PROPOSED WORK.....	31
3.1 Proposed Algorithm.....	33
3.2 Example.....	36
3.3 Storage and performance complexities	39
3.3.1 Rounds.....	40
3.3.2 Memory	41
3.3.3 Communication cost.....	42
3.3.4 Input/Output cost.....	42
Chapter 4: IMPLEMENTATION.....	43
4.1 Installation of Hadoop.....	43
4.2 Running the MST.....	54

Chapter 5: CONCLUSION AND FUTURE WORK.....	57
5.1 Conclusion.....	57
5.2 Future work.....	57
References.....	58
Publication.....	62
Appendix.....	63

CERTIFICATE

This is to certify that the work titled “**Minimum Spanning Tree in MapReduce Framework on Hadoop**” submitted by “**Mohit Sharma**” in partial fulfillment for the award of degree of **M.Tech** of Jaypee University of Information Technology, Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor

Name of Supervisor Mr. Suman Saha

Designation Assistant Professor

Date

ACKNOWLEDGEMENT

This may seem long but the task of my thesis work both theoretically and practically may not have been completed without the help, guidance and mental support of the following persons.

Firstly, I would like to thank my guide Assistant Professor, Department of Computer Science & Engineering, Jaypee University of Information Technology, Waknaghat, **Mr. Suman Saha** sir who provided me the idea and related material for the project proposal. He indeed guided me to do the task for my thesis in such a way that it seems to be research work. His continuous monitoring to support me and my research work encouraged me a lot for doing my thesis in very smooth manner.

Secondly, I would like to thank my friends who have always been with me for inspiring me that I can do the good thesis with the hard work. Their instigation always helped me to grow my mind focused towards the hard work for thesis with having the research work in the mind.

Thirdly, I would like to thank God for keeping me energetic, healthy and enthusiastic every time due to which I could complete my thesis work successfully.

Once again thank a ton to all mentioned people in my life.

Signature of the Student:

Name of the Student: Mohit Sharma

Date:

ABSTRACT

We live in a data age, it means that today there is big problem for processing the data. Data processing is required everywhere like industry, educational centers or research center. So handling of these types of huge data is big problem. Graphs are analyzed in many important contexts like page rank, protein-protein interaction networks, and analysis of social networks. Many graphs of interest are difficult to analyze because of their large size, often spanning millions of vertices and billions of edges. We believe that MapReduce has emerged as an enabling technology for large-scale distributed graph processing. Its functional abstraction provides an easy-to-understand model for designing scalable, distributed algorithms. The open-source Hadoop implementation of MapReduce has provided researchers with a powerful tool for tackling large-data problems.

In this work, we analyze the programmability and efficiency of minimum spanning tree graph algorithms on Hadoop using the MapReduce model. MapReduce provide parallel execution of the program that can be written in any language java, C++, Perl, Ruby etc. I have run a WordCount algorithm and design a minimum spanning tree (MST) algorithm called Round Robin minimum Spanning tree, in the MapReduce programming model. We have taken the problem of finding the minimum spanning tree by the Round Robin algorithm of a large graph, which is an important building block for many graph algorithms. Minimum Spanning Tree algorithm, comprising of three stages i.e. Find-Min, Connect-Component and Merge.

LIST OF FIGURE

2.1 Execution of MapReduce model.....	9
2.2 Detailed Hadoop MapReduce data flow.....	15
2.3 HDFS Architecture.....	17
2.4 Adjacency matrix representation.....	27
2.5 Partitioned adjacency matrix.....	28
2.6 Removing duplicate edges.....	30
3.1 Working of a single Mapper and Reducer System.....	32
3.2 Working of a multiple Mapper and Reducer system.....	33
3.3 Workflow diagram of proposed algorithm.....	35
3.4 A simple graph $G (V, E)$	36
3.5 After the completion of first round.....	37
3.6 After the completion of second round, generated MST.....	38
3.7 Graph vertices v/s rounds (initialization round is excluded).....	40
3.8 Graph rounds v/s vertices (when initialization round is included).....	41
4.1 Checking java and javac version.....	44
4.2 Starting and stopping of hadoop cluster.....	48
4.3 Compiling a MapReduce program code.....	51

4.4 Making a jar file of classes and execution of a program.....	51
4.5 Output of the WordCount MapReduce job.....	52
4.6 Retrieving the output from the HDFS.....	52
4.7 Execution of pi MapReduce example.....	53
4.8 Compiling and making a jar file.....	54
4.9 Insertion of the data files into HDFS and Running of the job.....	55
4.10 Data in HDFS and execution of the MST.	55
4.10 Output of the MST algorithm.....	56

Chapter 1

INTRODUCTION

1.1 Introduction

In the recent years, Data-intensive-computing and Cloud Computing [6] becomes more and more popular and growing rapidly. Almost all of the well-known enterprises in IT field were involved in this cloud computing carnival, including Google, IBM, Amazon, Sun, Oracle, and Microsoft and more.

The industry is focusing on cloud computing not only because of its growing, but also for its outstanding flexibility, scalability, mobility, automaticity, and the most important point is that it helps organizations to reduce cost. Lots of companies have published their products which were claimed using cloud technology. The product line covered from low level abstraction such as Amazon's EC2 [13] to higher levels like Google's App Engine etc.

The amount of data available and requiring analysis has grown at an astonishing rate in recent years. For example, Yahoo! Processes over 100 billion events, amounting to over 120 terabytes daily [7, 9]. Similarly, Facebook processes over 80 terabytes of data per day [9, 10, 11]. Although the amount of memory in commercially available servers has also grown at a remarkable pace in the past decade, and now it exceeds from hundreds of GB, it remains woefully inadequate to process such huge amounts of data.

Modern computers provide high processing, but the manipulation of multiple gigabytes of data is still not feasible on a single PC. If one machine takes ten hours to solve a problem, one could expect two cooperating machines to do it in five hours. Such an ideal is called linear speedup. Additionally, if those two machines could solve a twice as large problem at the same time as one machine, there would be so-called linear scale-up. The computation has to be split between several machines. But these ideals are hard to attain in practice because: (1) in order to divide a problem into suitable sub-problems and prepare all participants, there will be some start-up and communication overhead, which eventually grows dominant,(2) as the number of participants increases, it will be increasingly hard to devise evenly sized sub-problems for all of them, causing bottlenecks, and (3) there may be

contention for shared resources, causing interference between participants and overall slowdowns.

A distribution of work among a network of computers is more complicated than developing an application which runs on a single computer [28, 29]. The programmer has to find a way to divide the work - a task that might not be straightforward. Afterwards he has to implement different programs and coordinate their communication. The risk for the occurrence of failures and the time for their correction will increase significantly in such a system. Even if all programs contain no functional errors, it is not assured that they can interact properly. All of these problems lead to the idea of encapsulating the mechanisms for splitting the work between several computers in a library. With such a framework, this problem has to be solved only once and can be used for different data processing tasks. When building a system for manipulating huge amounts of data, the programmer only has to provide his own application code and does not have to mind the pitfalls of a distributed system. Rapid improvement and availability of cheap, commodity high-performance components were the driving force for a new era in computing to use networks of computers to handle large-scale computations [1, 6, 7, 11]. A very powerful but simple approach for implementing such a framework is provided by the MapReduce framework [1].

MapReduce was introduced by Jeffrey Dean and Sanjay Ghemawat; Google's engineers [1, 2] made a great impact by demonstrating a simple, flexible and generic way of processing and generating large distributed datasets. It was designed for and is still used at Google for processing large amounts of raw data to produce various kinds of derived data. Due to its simple programming interface, a novice programmer can also make effective use of large computing clusters [1]. It is more scalable and it works on commodity machines' cluster with integrated mechanisms for fault tolerance. MapReduce programs are written in a particular functional style and may be executed within a framework that automatically enables distributed and highly parallel execution. The programmer is only required to write specialized map and reduce functions as part of the Map/Reduce job and the Map/Reduce framework takes care of the rest. It distributes the data across the cluster, instantiates multiple copies of the map and reduce functions in parallel, and takes care of any system failures that might occur during the execution.

Since its inception at Google, MapReduce has found many adopters. Among them, the prominent one is the Apache Software Foundation, which has developed an Open-Source version of the MapReduce framework called Hadoop [7, 9, 11]. Hadoop is used for a number of large web-based corporates like Yahoo, Facebook, Twitter, Flipcart, Amazon, [10, 11] etc., that use it for various kinds of data-warehousing purposes. Facebook, for instance, uses it to store copies of internal logs and uses it as a source for reporting and machine learning. A lot of companies are owed to its ease of use, installation and implementation.

Hadoop has found many uses among programmers. One of them is a minimum spanning tree over large scale graphs in social network sites. Efficient solution techniques had been known for many years. However, in the last two decades asymptotically faster algorithms have been invented. Each new algorithm brought the time bound one step closer to linearity and finally Karger et al. proposed the only known expected linear-time method. Modern algorithms make use of more advanced data structures and appear to be more complicated to implement. Most authors and practitioners refer to these, but still use the classical ones, which are considerably simpler but asymptotically slower.

This thesis explores the existing solutions and the more recent algorithmic developments. A particular algorithm, Round Robin Minimum Spanning Tree is chosen to be designed using Map-Reduce model, and implemented on Hadoop. Scalability and performance of this new designed algorithm are evaluated.

1.2 Motivation

In recent years, Social networks are getting popular and growing rapidly. They became consumers of cloud computation, because when the size of social networks growing larger, it is impossible to process a huge graph on a single machine in a “real time” level execution time. Graph-based algorithms are becoming important not only on social networks, but also on IP networks, semantic mining and etc. To handle such large graphs data is not possible by the single machine. It can be handled by a distributed or the parallel computing way. But here, the programmer has to face a lot of problems like division of task, handling the task, integrate the results and the main thing is fault tolerance. The solution of such problem is given by MapReduce Framework.

Map-Reduce [1] is a distributed computing model proposed by Google. The main purpose of the Map-Reduce is to process large data sets paralleled and distributed. It provides a programming model in which users can specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Map-Reduce has become a popular model for developments in cloud computing.

The minimum-weight spanning tree problem, one of the most typical and well-known problems of combinatorial optimization, is that of finding a spanning tree of an undirected, connected graph, such that the sum of the weights of the selected edges is minimum [38, 39]. The minimum spanning tree problem is an important and very popular for the following reasons:

1. It is useful in many applications that uses it directly like, the design of computer and communication networks, power lines, leased-line telephone networks, wiring connections, links in a transportation network, piping in a flow network, etc.

2. It is also used to provide a method of solution to other problems to which it applies indirectly, such as network reliability, clustering and classification problems.

3. It is useful as a sub problem in the solution of other problems. E.g., minimum spanning tree algorithms are used in several exact and approximation algorithms for the travelling salesman problem, and in object matching problem.

The algorithm to determine the minimum spanning tree used in the Hadoop tool kit uses a single mapper and single reducer. We are focused to implement the Round Robin algorithm in the Map-Reduce framework with a single mapper and reducer and then extend it with multiple mappers and reducers.

1.3 Problem Statement

Firstly, Map-Reduce model has its own weakness because during the runtime of map or reduce function no sharing of information among the different machine is possible, not all of graph based algorithms can be mapped onto it. And, even though some graph related problems can be solved by Map-Reduce, they may not be the best solutions in a cloud environment. Finding out which kinds of graph based algorithms are the most suitable for the Map-Reduce Model is another problem of the thesis.

Secondly, the term “Minimum spanning tree” covered a large range of algorithms. So, how to categorize these algorithms and try to find patterns among them will be discussed. This problem also potentially indicates what class of algorithms can be implemented using Map-Reduce model.

Our main goal of this thesis is to design a **Round Robin Minimum Spanning Tree algorithm in the MapReduce Framework and analyze the efficiency of the algorithm.** It leads to the following sub questions:

Q1 What are the characteristics of Graph?

Q2 What are the characteristics of the MapReduce Framework?

Q3 How can we represent a graph in MapReduce model?

Q4 What will be the (key, value) pairs for Map and Reduce function?

Q5 What will be the Partition Function?

Q6 How reducer will communicate with HDFS in the middle to share information?

1.4 Organization of the Thesis

In Chapter 2, we give the overview on the basics of the MapReduce framework, cloud computing and Hadoop with the help of some graph-processing problem in MapReduce Model. It explains the fundamentals of the Minimum Spanning Tree algorithm and literature review on sequential and parallel algorithm. It also discusses the previous works done in areas related to Minimum Spanning Tree in MapReduce framework on Hadoop.

In Chapter 3, Some MapReduce version of the Round Robin Minimum Spanning Tree algorithm in detail are discussed that we have design. It is explained with an example and the storage and performance analysis of the algorithm.

In Chapter 4, the installation of Hadoop on a system is described. We take a WordCount example and run it on Hadoop. Then we discussed the MST algorithm implementation in MapReduce framework on Hadoop and its results. In Chapter 5, conclusion and future scope of the project is given which is followed by the References.

Chapter 2

LITERATURE REVIEW

The first section gives a general overview about cloud computing. In the second section we discuss about MapReduce framework and in third section some graph problem and minimum spanning tree algorithms are discussed. The last section gives the implementation of some distributed minimum spanning tree algorithms in MapReduce framework.

2.1 Literature Review on Cloud Computing

2.1.1 Cloud Computing

For maintaining data and applications, Cloud computing uses the internet and central remote servers. It allows businesses and consumers to use applications without installation and access their personal files on any computer with internet access [5, 6].

2.1.2 Cloud computing Service Model

The cloud computing service model is based on the three primary models:

1. Infrastructure as a service (IaaS),
2. Platform as a service (PaaS), and
3. Software as a service (SaaS)

Where IaaS is the most essential and each top model abstracts from the details of the lower models.

Infrastructure as a service (IaaS): This is the most basic service model of cloud computing, cloud providers provide computers – as physical or further often as firewall, virtual machines, networks and load balancers. These are provided by IaaS providers on demand from their large pools installed in data centers. Local area networks (LAN) containing IP addresses are part of the offer. The Internet can be used or - in carrier clouds - dedicated, virtual private networks can be configured, for the wide area connectivity.

Cloud users install operating system images on the machines as well as their application software, to manage their applications. In IaaS model, it is the cloud user who takes responsibility for patching and maintaining the operating systems and application software. Cloud providers in the general bill IaaS services on an efficacy computing basis,

that is, the cost will reflect the amount of resources allocated and consumed.

Platform as a service (PaaS): Cloud providers deliver a computing platform and or solution stack usually including operating system, software, web-servers and, programming language execution environment In the PaaS model. Application developers are able to develop and run their software solutions on a cloud platform not including the cost and complexity of buying and supervision the underlying hardware and software layers. Through some PaaS offers, the fundamental compute and storage resources to scale automatically to match application demand such that the cloud user doesn't have to assign resources manually [6].

Software as a service (SaaS): In this service model, cloud users can access the software from cloud clients that application software install and operate by cloud providers. Cloud infrastructure and platform are not managed by the cloud users on which the application is running. This provides a facility to cloud users to do, not install and run applications on own computers it simplifies maintenance and support. Elasticity property makes cloud application different from the other application. We can achieve this by cloning tasks on multiple virtual machines at run-time to meet the varying work demand. Over the set of virtual machine the work is distributed by the load balancer. The Distribution process is transparent for the cloud user who sees only a single access point. To put up a large number of cloud users, a single instance of cloud applications can be multiple users at a same time, it means, any machine in the cloud serves more than one cloud user organization. It is frequent to refer to particular types of cloud based application software with a common naming caucus called: business process as a service, desktop as a service, communication as a service, and Test environment as a Service.

2.2 Literature Review on MapReduce framework

Although parallelism in a distributed system is always possible, this does not mean that it can speed up all single computations. In client-server architecture for example the work on the server might still be executed in sequence and therefore does not benefit from the parallelism between client and server. In the general reduction of processing time in such an environment can be achieved by giving the server more resources than the clients. For computations on very large data sets it might be more feasible to split the work among

multiple servers. So the algorithm performed by the server should be executable in parallel. For this the sequential algorithm has to be transformed into a parallel algorithm. But finding such a parallel algorithm is not always an easy task. In fact, it is still a topic for research if it is possible to find a parallel counterpart for every sequential algorithm [28, 30]. Parallelism can be reached for many problems by splitting the input data and run the sequential algorithm on multiple processors at the same time. After each processor finished its work all computation results need to be merged together to get an overall result. With this method there is no need to modify the original algorithm to be applicable to parallel computation.

In most cases the creation of distributed systems is more complex than building a single application. Therefore the code for splitting the work, controlling the progress and merging the output is placed inside a framework. The application programmer only adds his own algorithm to manipulate the data and does not have to care about parallel programming issues. It is obvious that the framework needs a simple but powerful interface to link the application code. One of the approaches for building such a library leads to the MapReduce concept which is used in commercial computer systems today.

Map-Reduce [1] provides a programming model in which users need to define map and reduce functions to specify what kind of calculation should be performed on the partition of the input. The names and the general functionality of these two steps are borrowed from functional programming languages like LISP and Haskell. Map and reduce are standard functions in those languages and are used for list processing.

Here are some of the simple and interesting examples that can be easily expressed in Map-Reduce framework [1, 2].

Distributed Grep: It takes a pattern and then finds that pattern in the given file.

Count of URL Access Frequency: It emits the total count of any URL.

Reverse Web-Link Graph: it gives the number of sources with names in which target URL is present.

Inverted Index: it gives the list of documents in which a particular word falls.

Distributed Sort: it returns the records in a sorted manner.

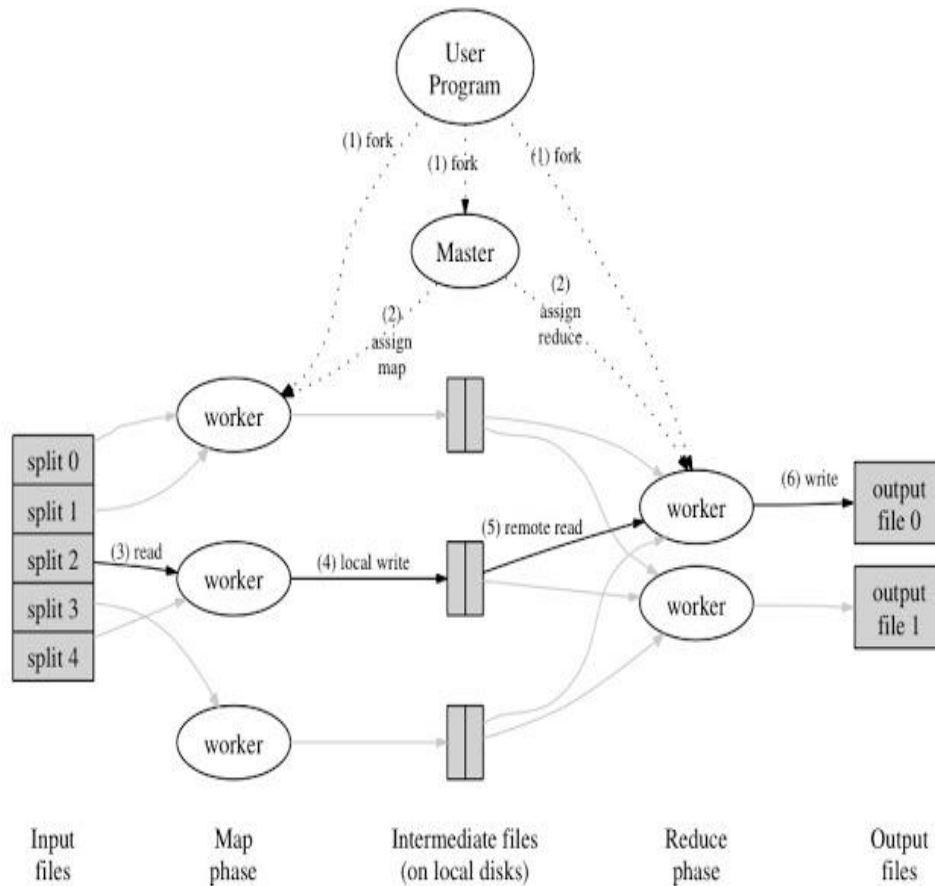


Figure 2.1: Execution of MapReduce model

One iteration of the map and reduce functions is called Map-Reduce Job. Job will be submitted to the master node of a machine cluster. According to the definition of Map-Reduce Job, master machine will divide the input data into several parts and arrange a number of slave machines to process these input data partitions in map functions. The output of map function will be intermediate result in the form of key-value pairs. The result will be sorted and shuffled, then routed to the proper reducer according to the rule defined by the partition function. The intermediate result will be processed again in the reducers, and turned into the final result. Because of the programs are written in a functional style and scheduling of all works and fault tolerance are automatically done by the Map-Reduce system itself, those programmers who do not have any parallel and distributed programming experiences can study easily and use Map-Reduce to model their own problem and process data using the resources on a cloud.

The user defines the map function and reduce functions in the form of (keys, value) pairs; like,

Map : $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$

Reduce : $(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)$

A **mapper** is a function (which may or may not be randomized) that receives one (key; value) pair as input. The mapper outputs a finite multiset of (key, value) pairs. A **reducer** is a function (which may or may not be randomized) that receives a key k , and a sequence of values v_1, v_2, \dots all of which are binary strings. The reducer outputs a multiset of pairs of binary strings $(k, v_{k1}), (k, v_{k2}) \dots$. The key in the output pairs is the same as the key received by the receiver as input [37].

A MapReduce program consists of a finite sequence of MapReduce rounds $(\alpha_1, \beta_1, \alpha_2, \beta_2, \alpha_3, \beta_3, \dots)$, where each α_i is a mapper, each β_i is a reducer, and the subsequence (α_i, β_i) denotes a MapReduce round. The input is a multiset of (key, value) pairs, denoted by U_0 , and U_i is the multiset of (key, value) pairs output by round i .

2.2.1 Bounds in MapReduce

The metrics typically used for efficiency in a MapReduce algorithm are the number of rounds required, and the amount of communication per round. There currently exist no lower bound techniques which can give lower bounds on the number of rounds for problems in the MapReduce model. However, research has been done on bounding the communication cost of problems with the MapReduce model, which require one or two rounds. This is done by modeling the tradeoff between parallelism and communication; more parallelization requires more communication [37].

The problems are viewed as sets of inputs, outputs, and a mapping of outputs to inputs. For example, finding the triangles in a graph: the inputs are sets of two nodes (edges), the outputs are sets of three nodes (the triangles), and the mapping of outputs to inputs is the set of three inputs representing the edges making up a given triangle. Here q is defined as the maximum number of inputs a reducer can receive and r is the replication rate, or the number of key-value pairs that a mapper can create from each input. The parallelism/communication tradeoff can be seen here as smaller values of q require more machines to solve the problem, which leads to more communication.

The replication rate is used as a measure of the communication cost for an instance of the problem, and is defined in terms of q and the size of the input. Among other results, the upper and lower bound of r for finding the number of triangles in a graph of n nodes is $\frac{n}{\sqrt{2q}}$. Similarly, the upper and lower bound of r for finding paths of length two in a n node graph is $\frac{2n}{q}$. The upper and lower bounds on r for matrix multiplication of an $n \times n$ matrix is $\frac{2n^2}{q}$ however the upper bound only holds for $q \geq 2n^2$ [1].

2.2.2 MapReduce Class (MRC)

The definition for the MapReduce paradigm provides a good framework for parallelization. However, it does not lie any restrictions on the program, or provide any notion of efficiency. Thus, a MapReduce Class (MRC) must be defined to help classify problems and algorithms. Without a restriction on the amount of memory any machine is allowed, any problem with a polynomial time classical algorithm could be solved in one round. However, the reason to use MapReduce is that the problem can't fit into the memory of one machine. Similarly, if any numbers of machines are allowed, the implementation becomes impractical [37]. Lastly, some restriction must be placed on the amount of time that can be taken. For example, allowing any reducer to run in exponential time would not make practical sense. Similarly, shuffling is time consuming because communication is orders of magnitude slower than processor speeds. Thus the number of MapReduce rounds should be bound in some way. These restrictions lead to the following definitions [10]:

Definition 2.2.2.1. A random access machine (RAM) consists of a finite program operating on an infinite sequence of registers, referred to as words [5].

Definition 2.2.2.2. Fix an $\epsilon > 0$. Let π be some arbitrary problem. We say $\pi \in MRC^\epsilon$ if there exists an algorithm that takes in a finite sequence of (key; value) pairs, (k_j, v_j) such that $n = \sum_j (|k_j| + |v_j|)$, and consists of a sequence of $(\alpha_1, \beta_1, \alpha_2, \beta_2, \alpha_3, \beta_3, \dots)$, operations which outputs the correct answer with probability at least $\frac{3}{4}$, Where:

- Each μ_r is a randomized mapper implemented by a RAM with $O(\log n)$ -length words, that uses $O(n^{1-\epsilon})$ space and polynomial time, with respect to n .

- Each μ_r is a randomized reducer implemented by a RAM with $O(\log n)$ -length words, that uses $O(n^{1-\epsilon})$ space and polynomial time, with respect to n .
- The total space, $\sum_{(k,v) \in U_{rr}} (|k| + |v|)$ used by the (key; value) pairs output by μ_r is $O(n^{2-2\epsilon})$.
- The number of rounds $R = O(\log^i n)$.

It is important to note that the space used by a RAM is measured by the number of words used. So, the definition above specifies that each mapper and reducer may use $O(n^{1-\epsilon})$ words each of size $O(\log n)$.

2.2.3 Deterministic MapReduce Class (DMRC)

MRC is defined for randomized reducers and mappers. We can similarly define a deterministic MapReduce Class, DMRC as follows [37]:

Definition 2.2.3.1. Fix an $\epsilon > 0$. Let π be some arbitrary problem. We say $\pi \in DMRC^i$ if there exists an algorithm which takes in a finite sequence of (key, value) pairs, (k_j, v_j) such that $n = \sum_j (|k_j| + |v_j|)$, and consists of a sequence $(\alpha_1, \beta_1, \alpha_2, \beta_2, \alpha_3, \beta_3, \dots)$ of operations which outputs the correct answer where:

- Each μ_r is a randomized mapper implemented by a RAM with $O(\log n)$ -length words, that uses $O(n^{1-\epsilon})$ space and polynomial time, with respect to n .
- Each μ_r is a randomized reducer implemented by a RAM with $O(\log n)$ -length words, that uses $O(n^{1-\epsilon})$ space and polynomial time, with respect to n .
- The total space, $\sum_{(k,v) \in U_{rr}} (|k| + |v|)$ used by the (key; value) pairs output by μ_r is $O(n^{2-2\epsilon})$.
- The number of rounds $R = O(\log^i n)$.

Because the shuffle phase is so time consuming, the goal when designing MapReduce algorithms is $O(1)$ rounds, typically a small constant. Even $O(\log n)$ rounds are often impractical.

2.2.4 Map-Reduce Example

We are taking two simple examples to better understand the MapReduce Framework.

Example 1: Lets, the problem to count the number of occurrences of every word in a large collection of documents. Here, the user has to write the following pseudo-code:

```
Map(String key, String value):
    // key: document name
    // value: document contents
    For each word w in value
    EmitIntermediate(w, "1");
Reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
    result = result +ParseInt(v);
    Emit(AsString(result));
```

So if a user gives input “We are here.” Then the output of the reducer is like,

[(a, 1), (e, 4), (h, 1), (r, 2), (w, 1)]

It is a very simple example. MapReduce model is used for large datasets.

Example 2:

Following pseudo-code to the map and reduce functions for categorizing a set of numbers as even or odd.

```
Map(String key, Integer values)
{
    //key :File Name
    //values :list of numbers
    for each v in values:
    if(v%2==0)
    EmitIntermediate("Even", v)
    else
    EmitIntermediate("Odd", v)
}
Reduce(String key, Iterator values)
```



```

{
//key: Even or Odd
//values : Iterator over list of numbers
//(categorized as odd or even)
String val = ""verbatim
While (values.hasNext())
{
Val=val+", "+values.toString()
}
Emit(key, val)
}

```

So if we give a list [3, 45, 12, 56, 4, 9, 90, 13, 32], then the final result look like,

Even 12, 56, 4, 90, 32

Odd 3, 45, 9, 13

2.2.5 Hadoop

Hadoop [8, 10] is projected by the Apache Software Foundation. Hadoop is an open source and Java based implementation of the MapReduce framework. Hadoop is scalable and reliable. It is a framework that supports distributed processing of large datasets across clusters of computers with the help of a simple programming model. [1] The advent of Hadoop was inspired by Google's MapReduce and GFS [4]. Due to the capability and simplicity, Hadoop has become a popular infrastructure for cloud computing. However, the Hadoop project is still young and immature. The weaknesses of Hadoop have manifested themselves mainly in the areas of Resource Scheduling [7, 9], Single Point Failure [10] and etc.

Not only the team in Apache, but also the Hadoop developers from all over the world are continually making their best effort to improve and perfect the Hadoop system and relevant projects of Apache. As an excellent large scale data mining platform, Hadoop is regarded as a good framework for graph related processing.

Hadoop provides the tools for processing vast amounts of data using the Map/Reduce framework and, additionally, implements the Hadoop Distributed File System

(HDFS). Figure 2.2 shows that how the data is flowed in MapReduce model. It can be used to process vast amounts of data in-parallel on large clusters in a reliable and fault-tolerant fashion.

Hadoop has two features:

1. Hadoop Distributed File System (HDFS) and
2. MapReduce processing engine.

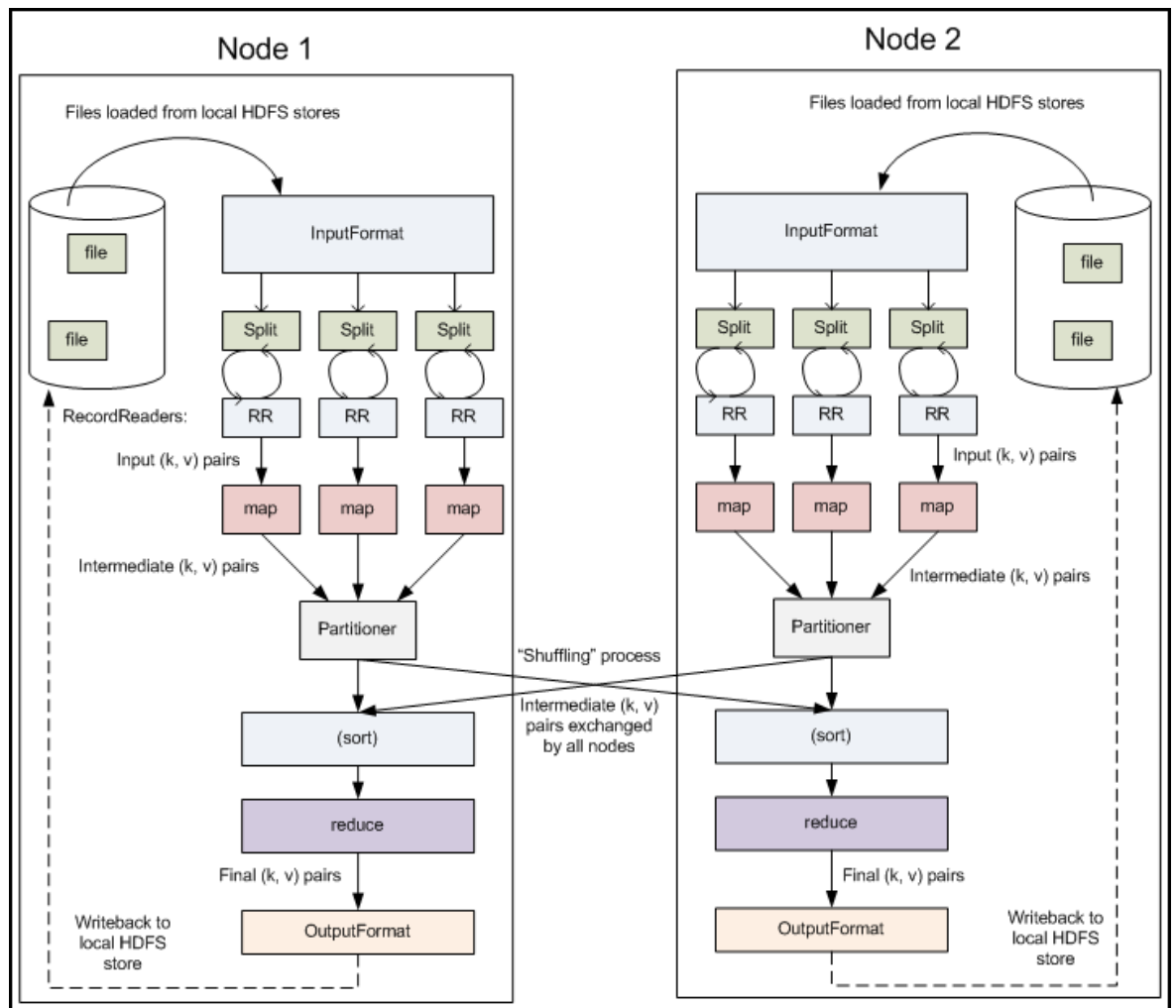


Figure 2.2: Detailed Hadoop MapReduce data flow

Hadoop uses master-slave architecture for both distributed computation and storage. In the distributed storage, the NameNode is the master and the DataNodes are the slaves. In the distributed computation, the JobTracker is the master and the TaskTrackers are the

slaves. An entire Hadoop execution of a client request is called a job. Users can submit a job request to the Hadoop framework, and the framework processes the jobs. Hadoop uses a Remote Procedure Call to communicate among the nodes.

Hadoop uses the four entities to process a job:

1. The **User**: submits the job and specifies the configuration.
2. The **JobTracker**: a program that coordinates and manages the jobs. It accepts the jobs from users, provides job monitoring and control, and manages the distribution of tasks in a job to the TaskTracker nodes. Normally there is one JobTracker per cluster.
3. The **TaskTrackers**: are the actual node that runs the Hadoop job. It processes the map and reduce tasks of a process. One or more TaskTracker processes per node in a cluster exist.
4. The **Distributed File System**: such as HDFS.

2.2.6 HDFS

HDFS is a distributed file system designed for storing very large files with streaming data access patterns, running on clusters on commodity hardware [12]. HDFS was designed keeping in mind the ideas behind Map-Reduce and Hadoop. What this implies is that it is capable of handling datasets of much bigger size than conventional file systems (even petabytes). These datasets are divided into blocks and stored across a cluster of machines which run the Map/Reduce or Hadoop jobs. This helps the Hadoop framework to partition the work in such a way that data access is locally as much as possible.

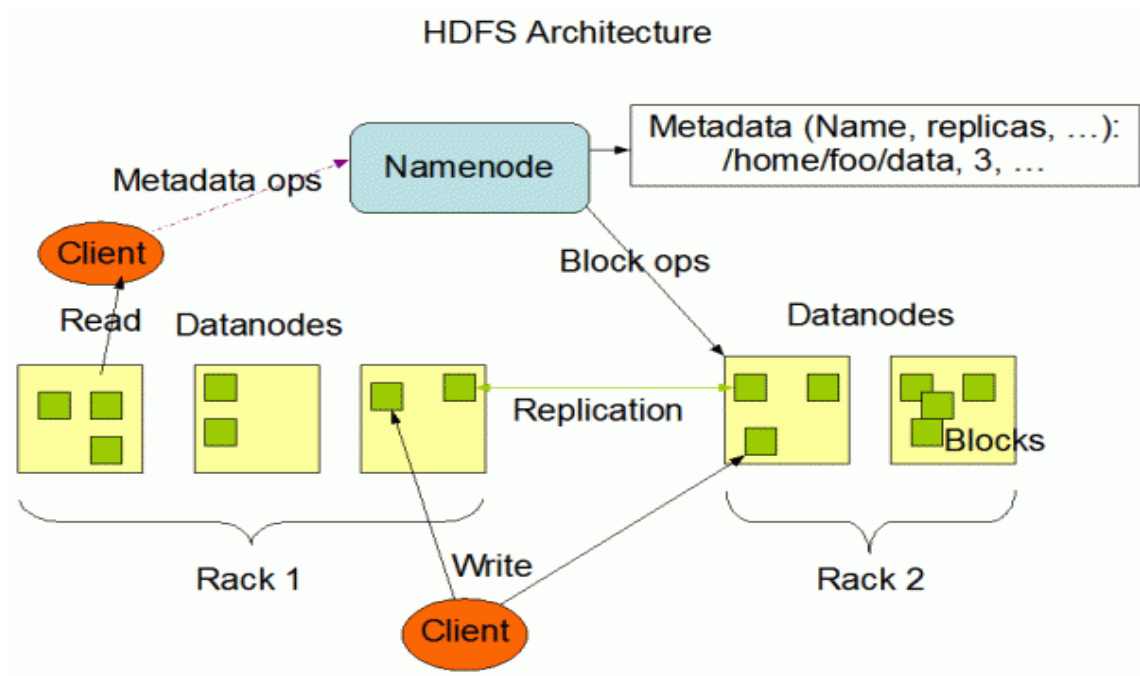


Figure 2.3: HDFS Architecture

Design of HDFS

HDFS is running on a cluster of commodity hardware for storing very huge files with streaming data access patterns [8, 10]. Now, we know the meaning of the statement.

Very large Files: We live in data age, and now a day in industry or education system everywhere data is so huge or files are very large it means that the file contain hundreds of megabytes (MB), gigabytes (GB), or terabytes (TB) in size. Today Hadoop clusters running with petabytes of data storage.

Streaming data access: HDFS is planned that the mainly well-organized data processing, pattern is a write-once and read-many-times pattern. We typically generated a data set or copied it from source, and after that various analyses are performed on that dataset over time.

Commodity hardware: Designing of Hadoop is to run on clusters of commodity hardware (commonly accessible hardware available from multiple vendors) therefore for large clusters the chance of node failure is high. It doesn't have need of expensive, highly consistent hardware to run on. Failure notification in HDFS is not known by the user and

the working is carried on.

Low-latency data access: Applications that have need of low-latency access to data, in the tens of milliseconds range, did not work fine with HDFS. Keep in mind, HDFS is optimized for delivering a high throughput of data, and this may be at the any cost of latency.

Hadoop is designed to run on clusters of machines. HDFS fulfills this requirement. HDFS has also some specific features, these are following:

1. A very important feature of HDFS is “streaming access”.
2. HDFS can handle large data sets.
3. HDFS supports a cluster of machines.
4. HDFS provides a write-once-read-many access model.

HDFS is written in Java language which provides it portability across various platforms.

Blocks in HDFS

The block size of the disk is the minimum amount of data that it can read or write. Blocks of filesystem are generally a few kilobytes (KB) in size, although disk blocks are normally 512 bytes. The filesystem user, who is simply reading or writing a file of any length, is generally known about this. On the other hand, there are tools like `df` and `fsck`, to perform filesystem maintenance, that operate on the filesystem block level. HDFS, also, has the concept of a block, but its blocks are much larger unit—64 MB (Megabytes) by default. HDFS files are broken into block-sized chunks, which are stored as independent units; same as a single disk file system. A filesystem for a single disk occupies a whole block, but in HDFS files that are smaller than a single block doesn't occupy a whole block's worth of underlying storage. Blocks of HDFS are larger than disk blocks, for minimizing the cost of seek. The time to transfer the data from the disk can be made to be significantly larger than the time to seek to the start of the block, by making a block size large enough. Therefore, the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

NameNode and DataNodes in HDFS

There are two types of nodes in an HDFS cluster, operating in a master-worker prototype: a NameNode (the master) and a number of DataNodes (workers). An HDFS cluster has a single NameNode, works as a master server that manage the file system namespace with controls right to use of files through clients. On the other hand, there are a number (more than one) of DataNodes; typically each cluster has one node, for managing the storage attached to the nodes so as to they run on. HDFS exposes a file system namespace as well as allowing user data to be stored in files. Within, a file is divided into single or extra blocks and these blocks are stored in a set of DataNodes. File system namespace operations like renaming files, closing, and opening and directories are executed through the NameNode. Mapping of blocks to DataNodes is also managed by the NameNode. The DataNodes also performs the block design, removal, and duplication ahead instruction from NameNode as well as DataNode are also liable for serving read as well as write desires from the file system's clients. [8] The NameNode as well as DataNode be pieces of software conscious to run on commodity apparatus. This machinery is naturally run a GNU Linux operating system (OS). HDFS is constructed using Java language; whichever machine that supports Java be capable to run the NameNode and DataNodes software. HDFS is able to be deployed on an extensive variety of machines using the advantage of portability of the Java language.

A usual operation has a devoted machine to runs merely the NameNode software. Every other machine within the cluster runs a single instance of the DataNode software. The architecture doesn't prevent running multiple DataNodes on the identical machine, however in an actual operation that is not often the case. The presence of a single NameNode into a cluster really simplifies the architecture of the system. The NameNode is the arbitrator with store for all HDFS metadata. The system is considered in such a way so as to user data never flow through the NameNode.

Data Replication in HDFS

HDFS efficiently store very huge files across machines in a huge cluster. Hadoop distributed file system stores all files as a series of blocks and all the blocks in a file, not including the last block are the same size. HDFS provides a good fault tolerance; therefore

the blocks of a file are replicated across all the machines. The duplication aspect and block size are configurable for each file. Replicas of a file are dependent on the application. The duplication factor can be notified at file formation time also can be altered later. HDFS files are writing-once and mutually exclusive it means they contain exactly one writer at every instant. The NameNode is one who only takes a decision about duplication of blocks. It occasionally receives a Block Report and a Heartbeat from every of the DataNodes in the cluster from time to time gives the block report and heartbeat to the NameNode. DataNode is functioning properly is known by reception of Heartbeat, and Block Report contain a record of all blocks on DataNode.

2.2.7 Related Projects

Although the idea of implementing map and reduce functions in functional programming languages is not new, the transfer of this concept into a distributed framework is still under research. There are three mentionable projects: Google MapReduce [1], Apache Hadoop [10, 11] and Nokia's Disco [12].

Google MapReduce

Google MapReduce was developed in 2004 at Google Inc. [1, 2]. It was designed to be able to run processes on the large input data of the company's search engine. Although the system itself is proprietary and only little detail knowledge is published, it is regarded as the prototype for a distributed MapReduce system. The core programming of software is written in C and it runs on thousands of machines in the company's data centers around the world.

Apache Hadoop

The Apache Hadoop system is a distributed MapReduce project of the Apache Foundation [9, 10]. As it is published under the Apache License, its source code is available for download. One aim of the project is the ability to be platform independent; therefore it is implemented in Java.

It is used in many commercial applications. In 2008, Yahoo! started to build parts of search engine's index with the help of Hadoop [10, 11]. It is also used in the Amazon Elastic Compute Cloud [13].

Nokia's Disco

The distributed MapReduce system Disco [12] was introduced in 2008 by the Nokia Research Center in Palo Alto. Disco's source code can be downloaded and is published under the revised BSD license.

In the Disco framework the map and reduce functions are specified in Python, but the core of the system is written in the functional programming language Erlang. Therefore, this software might be a good example for the development of a similar framework in Haskell. Currently there are no reports that Disco is used in bigger commercial applications, but like Hadoop it can easily be used in Amazon Elastic Compute Cloud [13], too.

2.3 Literature review on Graph algorithms in MapReduce Framework

A graph is a type of mathematical structures. The graph is an ordered pair $G = (V, E)$. V is the set of vertices or nodes which indicate the objects, people need to study, and the elements in set E are edges or lines which are abstraction of relations between different objects. If each pair of vertices is connected by edges which do not have incoming or outgoing states, the edges are undirected. Otherwise, the edges are directed from one vertex to another.

Vertices and edges could have values or states on them. Some graph theory problems focus on those states (e.g. Single Source Shortest Path Problem, Minimal Spanning Tree Problem, etc.). Most of graph-based algorithms can be categorized into two classes, vertex-oriented and edge-oriented. But if the main part of an algorithm is to compute the states of edges or message transmitting, (e.g. PageRank [34, 35]), the algorithm will be considered edge-oriented.

MapReduce is very well suited for naïve parallelization for example; counting how many times a word participated in a data set. **Jinn et al. [34]** an example, graph is split into blocks and taken as input of map function. In map function, the value of each vertex is divided by the edge number of that vertex, and the result is stored as a key / value pair {neighbor ID, result}. Before reduce function, each machine will fetch a certain range of key/value pairs onto its local storage, and performs reduce function on each key value. In this example, reduce function reads all values under the same key (vertex ID), sum them

up, and write the result back as the new value of this vertex. Some graph algorithms are here that have converted in the MapReduce model.

Parallel breath first search in MapReduce: [1] [26] the mappers emit distances to reachable nodes, while the reducers select the minimum of those distances for each destination node. Each iteration (one MapReduce job) of the algorithm expands the search frontier by one hop. Breadth first search is an important searching technique in the graph. Basically, this algorithm works sequentially, but with the help of the MapReduce we can make it parallel breath search. In the Data intensive text processing [1] given by the Jimmy Lin and Chris Dyer has proposed this breath search algorithm in the map reduce, and they found that it is faster than the sequential breath first algorithm.

Page Rank with MapReduce [26]: PageRank [21] [26] is used to measure the web page quality based on the structure of the hyperlink graph. Although only one of thousands of features is taken into account in Google's search algorithm, it is one of the best most studied and known. [10] First all the nodes are given to the map function and then processing in map phase the intermediate key-value pair is generated, this key-value pair is as input to the reducer. After this whole process the page rank is calculated parallel and the result set generated.

2.4 Literature review on Minimum Spanning Tree Algorithms

Computing a minimum spanning tree is one of the most studied problems in combinatorial optimization [17, 29]. Formally, an MST of a given undirected connected Graph $G = (V, E)$ with vertices $V = \{v_0, v_1, v_2, \dots, v_{n-1}\}$ and weighted edges E , $|E|=m$, can be defined as an acyclic subgraph of G which connects all vertices in V with the least total weight.

The two theorems on the optimality conditions for a MST are stated below without proofs. Two elegant proofs can be found in [16, 28, 29, 30].

Theorem 1: (*Fundamental cut-set optimality*). A spanning tree T in a weighted graph is a MST if and only if every edge in the tree is a minimum-weight edge in the fundamental cutset defined by that edge.

Theorem 2: (*Fundamental cycle optimality*). A spanning tree T in a graph G is a MST if and only if every edge $e \in (E - T)$ is a maximum weight edge in the unique fundamental cycle defined by that edge.

The MST problem is solved by a simple incremental, *greedy* method in which the MST is built by taking small edge and excluding other large ones. The technique is greedy in the sense that at every stage, the best possible edge is chosen for inclusion in the MST without producing a cycle in the subgraph constructed so far or for exclusion from the MST without disconnecting the graph. **Tarjan** [15, 21] represents the construction process as one of edge coloring. Starting from an initial uncoloured edge set, we colour one edge at a time either blue (included) or red (excluded) according to the following two rules:

Blue rule: Select a cutset that does not contain a blue edge. Select a minimum weight from the cutset among the uncoloured edges and colour it blue.

Red rule: Select a simple cycle containing no red edges. Select a minimum weight on the cycle among the uncoloured edges and colour it red.

These two rules have a lot of direct and indirect applications of the fundamental cutset and fundamental cycle optimality theorems.

The most important property of the greedy method is that it colours all the edges of any connected graph and maintains a MST containing all of the blue edges and none of the red ones.

2.4.1 Sequential Minimum Spanning Tree Algorithms

Classical algorithms

The three classical algorithms differ in their starting states and coloring steps. In other words, they differ in the criterion used to select the next edge or edges to be added in each iteration. **The Boruvka algorithm** [18] starts with the initial set of n blue trees. Each blue tree initially, consisting only one vertex and no edge. And then repeat the following step until there is only one blue tree.

Coloring step: For every blue tree T , select a minimum weight edge incident to T . Colour all selected edges blue. The algorithm builds the trees uniformly throughout the graph. It is, therefore, suitable for use in parallel computations. It runs in $O(m \log n)$.

The Kruskal algorithm [19]: Sort the edges in the order of increasing weights and apply the following step to the edges in the sorted list until the number of blue edges is $n-1$.

Colouring step: If the edge considered has both endpoints in the same blue tree, colour it red; otherwise colour it blue. The algorithm builds up blue trees in an irregular fashion dictated only by edge weights. It is worth noting that the algorithm is best in situations where the edges are given in sorted order or can be sorted fast (like when the weights are small integers, making it possible to employ radix sort) or where the graph is sparse. If edges are in disorder with respect to their weight, the Kruskal algorithm requires $O(m \log n)$ time.

Given a sorted edge list, finding a MST requires $O(m \alpha(m, n))$ time assuming the use of the disjoint set union algorithm of Tarjan[27, 26], Where $\alpha(m, n)$ is the inverse Ackermann's function $\alpha(m, n)$ is defined as

$$A(m, n) = \min \{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n\}, \text{ for } m \geq n \geq 1$$

Where Ackermann's function $A(i, j)$ for $i, j \geq 1$ is given by

$$A(1, j) = 2^j \text{ for } j \geq 1,$$

$$A(i, 1) = A(i-1, 2) \text{ for } i \geq 2,$$

$$A(i, j) = A(i-1, A(i, j-1)) \text{ for } i, j \geq 2.$$

The function $A(i, j)$ grows exponentially fast and hence $\alpha(m, n)$ is a very slowly growing function, $\alpha(m, n) \leq 3$ for $n < 2^{16}$. Hence, for all practical purposes $\alpha(m, n)$ is assumed to be a constant not larger than four.

The Prim algorithm [20] Use an arbitrary starting vertex s and apply the following step $n-1$ times.

Colouring step: Let T be the blue tree containing s vertices. Now, Select a minimum weight edge that is incident to T and colour it blue.

The algorithm generates only one nontrivial blue tree from a single root. The Prim algorithm with an implementation using d -heaps, as proposed by Johnson, runs in $n \log_d n + m \log_d n$. In the case of binary heaps (i.e., $d=2$), the time complexity is $O(m \log n)$. Thus, the algorithm with Johnson's implementation is well suited for dense graphs as well as sparse ones and the method is asymptotically worse than that of Kruskal only if the edges are presorted.

Modern algorithms

Yao [22] was the first to discover an implementation of Boruvka which had a faster asymptotic running time. It has the complexity $O(m \log \log n)$. **Cheriton and Tarjan [23]** followed by proposing a similar but slightly more practical algorithm. It runs in $O(m \log \log n)$ time. Both algorithms are based on the general blue rule algorithm given above. They both use heaps to implement efficient Boruvka-like colouring steps. With the invention of an efficient form of heap, called the Fibonacci heap (F-heap), **Fredman and Tarjan [23]** developed an implementation of the minimum spanning tree algorithm which runs in $O(m \beta(m, n))$.

Where $\beta(m, n) = \min\{i / \log^{(i)} n \mid i \leq m/n\}$ with $\log^{(0)} n = n$.

In addition, they pointed out that the use of an F-heap in its simplest form in the implementation of the Prim algorithm, i.e., replacing the binary or d-heap with an F-heap solves the MSTP in $O(n \log n + m)$ time. **Gabow et al. [24]** introduced the use of packets in conjunction with F-heaps and modified the Fredman and Tarjan algorithm to achieve the slightly better time complexity of $O(m \log \beta(m, n))$. The method of Gabow et al. was the fastest theoretically until recently. **Karger et al. [25]** proposed a randomized minimum spanning tree which gives the linear time complexity. It has $O(n \log n + m)$ time complexity. Karger et al. [25] stands as theoretically the fastest linear expected-time algorithm of till now. It is a randomized and recursive algorithm which also requires the solution of a related problem, that of verifying whether a given spanning tree is minimum. It has $O(m)$ time complexity. **Cheriton and Tarjan Algorithm [23]:** For each blue tree, a heap is maintained and that holds the edges with at least one endpoint in the tree and which are candidates for becoming blue; the cost of an edge being its key in the heap. It differs from the Boruvka algorithm because here in each iteration we pick one blue tree in the forest for the merge operation. In conjunction with the heap, the selection rule for picking a blue tree for merging plays an important role in beating the $O(m \log n)$ time bound. Two alternatives are [23]:

1. Pick uniformly the first in a queue of candidate blue trees (implementation with a doubly linked list is sufficient to have an $O(1)$ time operation including deletion from the queue),
2. Pick the smallest candidate tree (less efficient with an $O(n)$ implementation).

Cheriton and Tarjan's algorithm is asymptotically faster than any of the three classical algorithms for sparse graphs, but is slower by a factor of $O(\log \log n)$ than the Prim algorithm for dense graphs.

2.4.2 Parallel Minimum Spanning Tree Algorithms

Numerous models of parallel computation have been proposed in the literature, while the most prevalent model in theoretical computer science is the PRAM. In a PRAM, an arbitrary number of processors shared a large memory space. Same shared input is given to produce some output. There are different types of PRAMs algorithms, based on issues of reading and writing in shared memory. In this model, if we have a problem of size n , then polynomial in n number of processors is required. It is a necessary, but hardly sufficient, condition to ensure efficiency.

There are two general strands of PRAM research [38]. The first asks, with a polynomial number of processors what problems can be solved in polylog time on a PRAM? Polylog time serves as a standard for parallel running time, and a polynomial number of processors provide a necessary condition for efficiency. The second strand of research asks which algorithms can be efficiently parallelized. While theoretically appealing, the PRAM model suffers from the practical drawback that fully shared memory machines with large numbers of processors do not exist to date (though they may in the future) and simulations are slow. It seems difficult to build a large computer with a large robust shared memory. Prim's and Kruskal's algorithms are inherently sequential, while Boruvka's algorithm has natural parallelism. There are a lot of parallel algorithms that have been proposed [28, 29].

Therefore, most parallel MST algorithms are based on Boruvka's approach. Even though these algorithms achieve parallel speedup for relatively regular graphs; none of these Boruvka's algorithm-based implementations runs significantly faster than the best sequential algorithm for a wide range of irregular and sparse graphs.

2.5 Literature review on Minimum Spanning Trees algorithms in MapReduce Framework

With the increasing of graph size, there have been a lot of studies on the distributed algorithms for finding MST with the goal of reducing both the running time and the number of messages exchanged among computing nodes.

H. Karloff et al. [37] demonstrate how algorithms can take advantage of MapReduce model to compute an MST of a dense graph in only two rounds, as opposed to $(\log(n))$ rounds needed in the standard PRAM model. He proposed an easily parallelized algorithm for MST [16]. Denote the graph, vertex set; edge set by G, V, E , the procedure of this algorithm is as follows:

Step 1: Vertex set V is partitioned into k equally sized subsets,

$V=V_1 \cup V_2 \cup \dots \cup V_k$ with $V_i \cap V_j = \text{NULL}$ for $i \neq j$ and $|V_i| = N/k$

For every pair $\{i, j\}$, let $E_{i,j}$ is subset of E be the edge set induced by $V_i \cup V_j$, i.e. $E_{i,j} = \{(u, v) \in E \mid u, v \in V_i \cup V_j\}$, denote the resulting sub graph by $G_{i,j} = (V_i \cup V_j, E_{i,j})$

Step 2: for each of $\binom{k}{2}$ the sub graphs $G_{i,j}$, compute the unique minimum spanning forest $M_{i,j}$; then let H be the graph consisting all of the edges present in $M_{i,j}$, $H = (V, \cup_{i,j} M_{i,j})$

Step 3: Now, compute the minimum spanning tree M of H .

The algorithm is not so good as best sequential algorithm, but easy parallelization is there. More importantly, it has been demonstrated that it can be implemented with MapReduce framework. Here MapReduce MST algorithm works in two rounds:

-Adjacency matrix M is used to denote the graph G as shown in figure

$$\left[\begin{array}{c|cccc} & 0 & 1 & \dots & (n-1) \\ \hline 0 & d_{00} & d_{01} & \dots & d_{0(n-1)} \\ 1 & d_{10} & d_{11} & \dots & d_{1(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (n-1) & d_{(n-1)0} & d_{(n-1)1} & \dots & d_{(n-1)(n-1)} \end{array} \right]$$

Figure 2.4: Adjacency matrix representation [30]

-Now the vertex set is partitioned into k equally sized sub sets. This can be achieved by putting vertex I in sub set $(I(n/k))$, shown in following figure;

	$0 \dots \frac{n}{k} - 1$	$\frac{n}{k} \dots 2\frac{n}{k} - 1$...	$(k-1)\frac{n}{k} \dots n-1$
0	$[0][0]$	$[0][1]$...	$[0][k-1]$
\vdots				
$\frac{n}{k} - 1$				
$\frac{n}{k}$	$[1][0]$	$[1][1]$...	$[1][k-1]$
\vdots				
$2\frac{n}{k} - 1$				
\vdots	\vdots	\vdots	\ddots	\vdots
$(k-1)\frac{n}{k}$				
\vdots	$[k-1][0]$	$[k-1][1]$...	$[k-1][k-1]$
$n-1$				

Figure 2.5: Partitioned adjacency matrix

In the first round;

-Matrix M is partitioned to $\binom{k}{2}$ blocks, each block with a unique partition ID.

- Denote partition ID by **PID**, which is in the form of [Row Element] [Column Element]

-Since G is undirected, it'll suffice only considering the upper triangular part of the matrix.

This partition process is accomplished by the Mapper. Input for the mapper is matrix M and output of mapper is partitioned vertices with neighbor vertex and weight.

Map1:

Input (Key, Value) pair for first Mapper is

$\langle \text{Line Number}, (\text{firstVertex}, \text{secondVertex}, \text{weight}) \rangle$

And Output (Key, Value) pair or the intermediate keys are like

$\langle [\text{first}] [\text{Second}], (\text{firstVertex}, \text{secondVertex}, \text{weight}) \rangle$

Reduce1:

Input (Key, Value) pair for first Reducer is PID and an edge list of $E_{i,j}$.

$\langle \text{PID}, \text{List} (\text{firstVertex}, \text{secondVertex}, \text{weight}) \rangle$

And the output of reducer is like

$\langle \text{PID}, (\text{firstVertex}, \text{secondVertex}, \text{weight}) \rangle$

In the Second Round:

The work of second round is to collect edges of every M_{ij} and compute the global MST. The map of this round is different than Map1 and reduce job is same as the first round.

Map2: Input- $\langle \text{PID}, (\text{firstVertex}, \text{secondVertex}, \text{weight}) \rangle$

Output- $\langle \text{MST}, (\text{firstVertex}, \text{secondVertex}, \text{weight}) \rangle$

Where MST: Select all the MST edges into one list and setting the key “MST”.

S. Chung et al. [28, 36] made an improvement in this algorithm, he use Kruskal's algorithm, when computing the MST of each sub graph. The reason is as follows:

1. Kruskal outperforms Boruvka's algorithm on a general basis.
2. When the graph is disconnected, Kruskal finds the minimum spanning forest of the graph. However, Prim can only find MST in connected graph.

They apply the following two operations between step 2 and step 3 of the basic algorithm:

1. Removing the duplicated edges;
2. Let $G=H$, and $k=k-1$. Then go to step 1.

Repeat the above operations until the edge number of H is small enough.

In this paper he suggests that we can reduce the edge number of H by setting a smaller k . So the edge number of H can be greatly reduced by repeating the above two operations. The reason why we don't set k to a very small value initially is to allow more parallelization on the original graph. By analyzing the procedure of the basic algorithm, we decompose the implementation into two rounds of MapReduce job. Step 1 and step 2 are completed in the first round and step 3 in the second round. Assume without loss of generality that the input file comprises edge records, with the form of (first vertex, second vertex, weight) and each record per line. The file of a graph G with n vertexes and m edges will have m records, and the vertexes in G are labeled by 0 to $(n-1)$. (Here we discuss the MST of an undirected graph, (first vertex, second vertex, weight) and (second vertex, (first vertex, weight) vertex is smaller than the second vertex, just for unity).

Based on the flow of basic algorithm's task's there is one new MapReduce job which needs to be designed. That is the job of removing duplicated edges. This can be achieved as follows.

The map sets edge record as the output key and integer “1” as output value. The framework will group the same keys together (since the key is edge record, actually the same edges are grouped together). The reduce outputs the key (i.e. edge record) received from map line by line.

The job of removing duplicated edges is added between the first round and the second round.

Map:

Input (key, value) pair- (lineNumber, <firstVertex, secondVertex, weight>)

Input value is set to the output key.

Output (key, value) pair- (< firstVertex, secondVertex, weight>, 1)

Reduce:

Input (key, value) pair-(<firstVertex, secondVertex, weight>, List<> (1))

Duplicated edges are group together because of they have the same key. Reduce set edge as output value.

Output (key, value) pair-(NULL, <firstVertex, secondVertex, weight>)

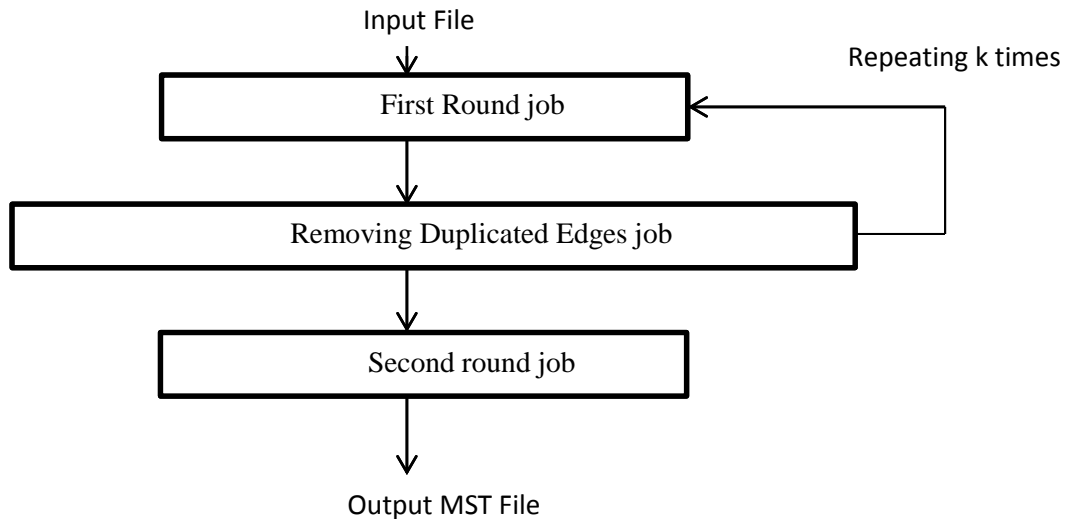


Figure 2.6: Removing duplicate edges

Chapter 3

PROPOSED WORK

Graph based algorithms covered a wide range of problems. To make the research significant, the algorithms chosen here must be representative. That algorithm should fulfill the following factors:

1. Chosen algorithms should be useful and have been widely used.
2. That would not be considered easy to map-reduce.
3. That should be comparable, by which it means, that should have a certain level of similarity of data inputs, outputs and purposes of the algorithm themselves.
4. That algorithm which has not been applied on the Map-Reduce model would be considered first.

Here, we select Round robin MST algorithm which follows all the points defined above. First, we will discuss the basic concept of Round Robin MST algorithm and then focused to generate a MapReduce version of the Round Robin MST algorithm. We know that in a sequential way it has the complexity $O(m \log \log n)$ which is better than Boruvka's minimum spanning tree and all others in practical aspects. A sequential algorithm of RRA MST can be understood from the Robert Tarzan's book [46]. He used the Leftist Heap data structure to implement the queue.

Step 1: Set- data structure is used for storing the vertex set, a mapping function is used for providing the mapping between the canonical vertex set and their vertices heap.

Step 2: Now we perform the MakeSet (x) operation on all vertices to form make sets and creates the leftist heap for these vertex sets.

Step 3: Now the first element from the queue is taken out and FindMin(x) operation is executed in the leftist heap for that vertex set. For which it comes minimum that vertex or the edge is considered and Merge (H1, H2) operation is performed. And we take the union of these two vertex sets; represented by a canonical vertex and then it is inserted into the rear of the queue. Step 3 is repeated until (queue size > 1). In the final we got a minimum spanning tree of the input graph.

Now we will perform all these steps (defined above) in a parallel way so that it can be used in MapReduce framework. We know that the map operation is stateless (because it operates on one pair at a time). It allows for easy parallelization to our algorithm. So the different inputs are processed by different machines. Each mapper emits the intermediate keys. In the shuffle phase intermediate keys are read by the reducer with the help of the master. The partition function is the base for this shuffle phase. If the same key is sent to the reducer than this framework gives the best results. Partition function affects a lot for this MapReduce model. Here we use partition function $(w \bmod R)$ i. e. based on the weight. Mod R is applied to the weight, so the same edge is passed to same reducer. At the reducer side, we use another function, i.e. the Compact function. It reduces the redundancy and creates the forest by merging the edges of the vertex sets. Every forest is represented by a canonical vertex.

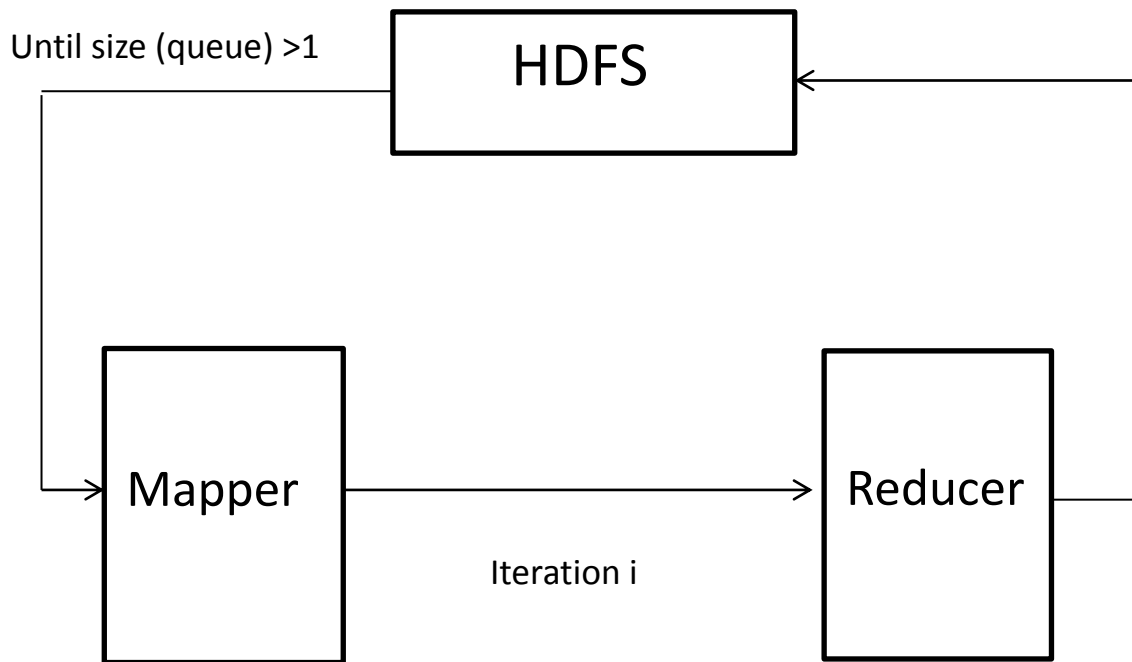


Figure 3.1: Working of a single Mapper and Reducer System

3.1 Proposed Algorithm:

Firstly, we will use a single mapper and reducer framework and implement this round robin algorithm in MapReduce environment. As we shall exhibit, this algorithm will take advantage of the interleaving of sequential and parallel computation that MapReduce offers algorithm designers.

Input : Graph $G (V, E)$

Output : Minimum spanning tree

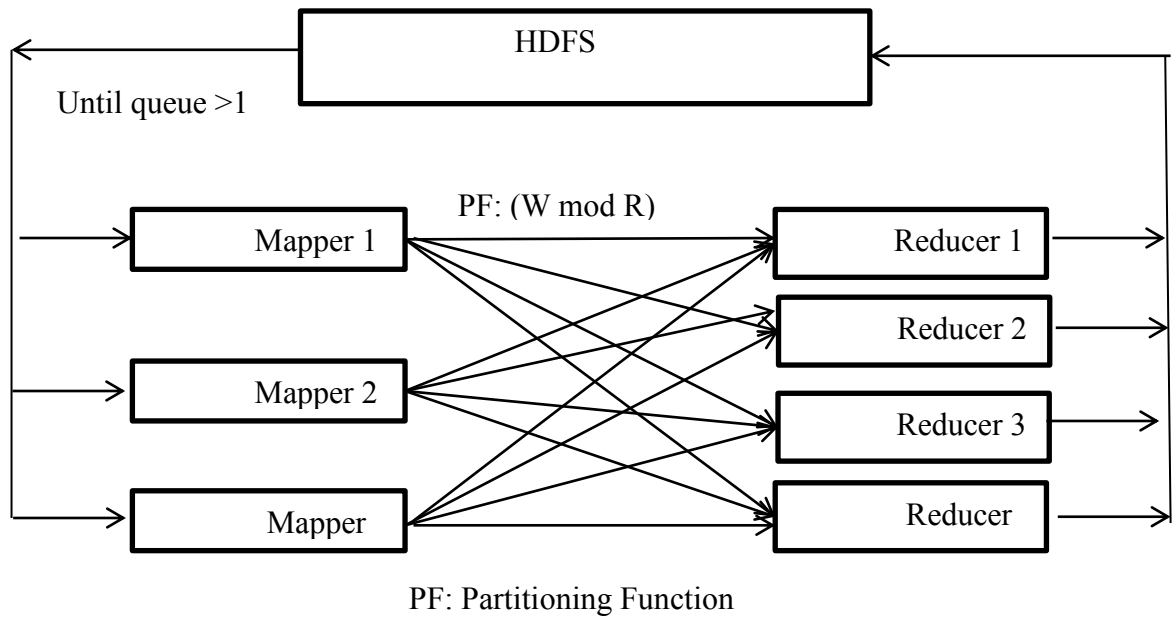


Figure 3.2: Working of a multiple Mapper and Reducer system

Initially, we apply an initialization round of MapReduce that converts an input graph $G (V, E)$ into the format $[\{x\}, h(x)]$; which will be suitable to work further round for RRMST-MapReduce algorithm. Where $\{x\}$ is union-find data structure and x is the canonical vertex of the set (initially that contains single vertex by applying the makeset (x) operation on each vertex of the graph G); and $h(x)$ is the leftist heap for a vertex ($x \in V$) that contains all the edges of the graph incident to vertex x . The outputs of initial round are inserted into the queue in HDFS. So the queue contains the vertex set with its heap stored in the HDFS (Hadoop Distributed File System) in the following format,

Queue: [$\langle \{A\}, h(A) \rangle, \langle \{B\}, h(B) \rangle, \langle \{C\}, h(C) \rangle, \dots \dots \dots \langle \{N\}, h(N) \rangle$]

The following data structure is used for implementing the algorithm in MapReduce,
Set {}: It is a union-find data structure that stores the edges of the blue trees that will form a minimum spanning tree.

Map: f = $[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$ is a set of ordered pairs that have no two same first element in the set.

Heap: h(x) is the leftist heap that stores all the edges of the graph incident to the vertex x or vertex set.

Cost: $(\text{Cost} = \text{cost} + w)$

Algorithms for Map phase:

Input: $\langle \text{Nid}, h(\text{Nid}) \rangle$

Class Mapper (Vertex set x, Heap h(x))

1. Mapper checks the minimum weighted edge in the heap by using FindMin(x) operation
2. It returns an edge like {x, y} with its weight. These are the intermediate keys.
3. These intermediate keys are used as input to the reducer.

Output: $[\text{Nid}, \langle \{x, y\}, h(\text{Nid}), w \rangle]$

Now partition function is applied on the intermediate keys so that these intermediate keys are perfectly sent to the particular user.

Partition Function ():

We are using a simple partition function, i.e. $((W) \bmod R)$.

Algorithm for Reduce phase:

Input: $\langle w, [\{x, y\}, h(x)] \rangle$

Class Reducer (w, List of edges and heaps)

1. Add edge (x, y) in set S.
2. Now, checks the condition: if $\text{Find}(x) \neq \text{Find}(y)$ then
3. Merge the heap tree of {x} and {y} into one; Merge $(h(x), h(y)) \rightarrow h(x)$
4. Delete (x, y) edge from the heap h(x).
5. Update the cost.

Output: $\langle \{\text{Nid}\}, [h(\text{Nid}), w]$

Compact Function ()

If $\text{Parent}(x) = \text{Parent}(y)$

Then, UNION (x, y)

Now these outputs are written into the HDFS. All vertex sets are inserted into the queue from rear side with their heaps. By combining these steps called around and these rounds are repeated until size (queue) >1.

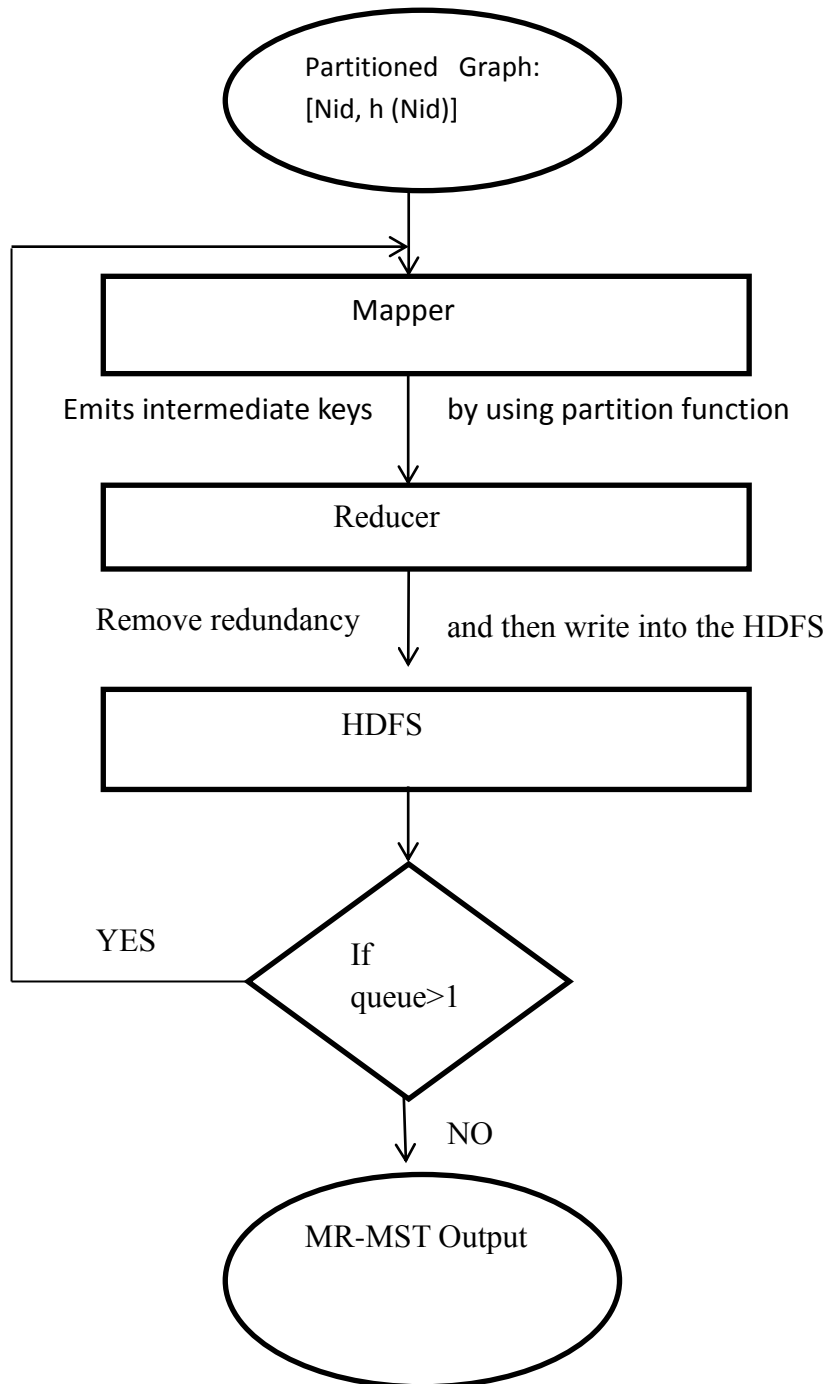


Figure 3.3: Workflow diagram of proposed algorithm

The first initialization round is applied, which makes the data used for next rounds in MapReduce framework. In this we make the partition of data and send it to the mapper. Mapper performs their function and with the help of partition function intermediate keys are sent to the reducers. The output of a round is minimum spanning forests. Because redundancy may be exist in the outputs of the reducer so we use a compact function which reduces the redundancy. So the output of the reducer is sent to the compact function. It merges the vertex sets of common vertices and forms the minimum spanning forest. After each round at most half of the minimum spanning forest generates to the previous one. Now, this output is send to the HDFS, and queue, cost, blue tree set are updated. This process repeats until $\text{size}(\text{queue}) > 1$. We can understand it from the following diagram,

3.2 Example:

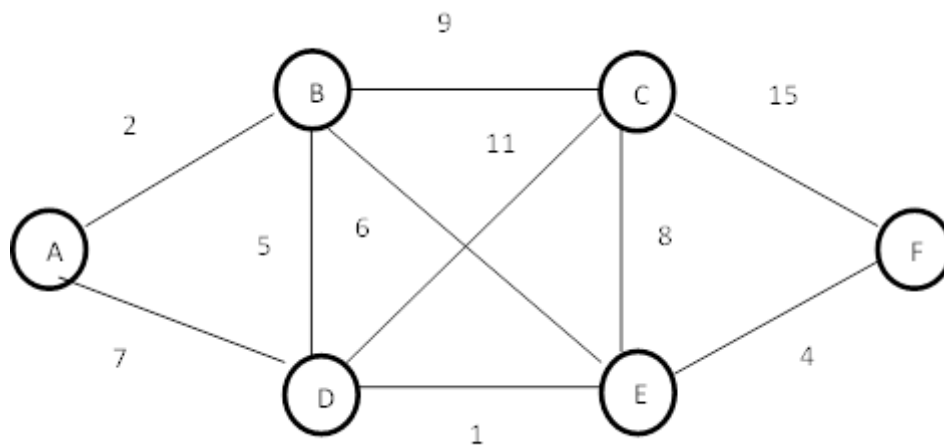


Figure 3.4: A simple graph $G(V, E)$

We process this graph by a single mapper and reducer,

First Round

Map: Graph G is input to the mapper. Mapper emits the intermediate keys by applying the partition function on the inputs in the following format

$$[nid, \langle \{x, y\}, h(nid), w \rangle],$$

So these are following:

$$\begin{aligned} & [A, \langle \{A, B\}, h(A), 2 \rangle], \\ & [B, \langle \{B, A\}, h(B), 2 \rangle], \\ & [C, \langle \{C, E\}, h(C), 8 \rangle], \\ & [D, \langle \{D, E\}, h(D), 1 \rangle], \\ & [E, \langle \{E, D\}, h(E), 1 \rangle], \\ & [F, \langle \{F, E\}, h(F), 4 \rangle]. \end{aligned}$$

Reduce:

Now these values are read by the reducer which checks the condition; if find (x) ≠ find (y) then merge the heaps of these edges vertex and this edge is written in the blue tree set. And cost is updated. So outputs are like,

$$\begin{aligned} & [n1, \langle \{A, B\}, h(A), 2 \rangle], \\ & [n2, \langle \{C, E\}, h(C), 8 \rangle], \\ & [n3, \langle \{D, E\}, h(D), 1 \rangle], \\ & [n4, \langle \{F, E\}, h(F), 4 \rangle]. \end{aligned}$$

Now the output of the reducer is act as input for the compact function which merges the edges which have something (i.e. vertex) common. The outputs of this function are like,

$$[\langle \{A, B\}, h(A), 2 \rangle, \langle \{C, D, E, F\}, h(C), 13 \rangle]$$

Now the graph is looking like,

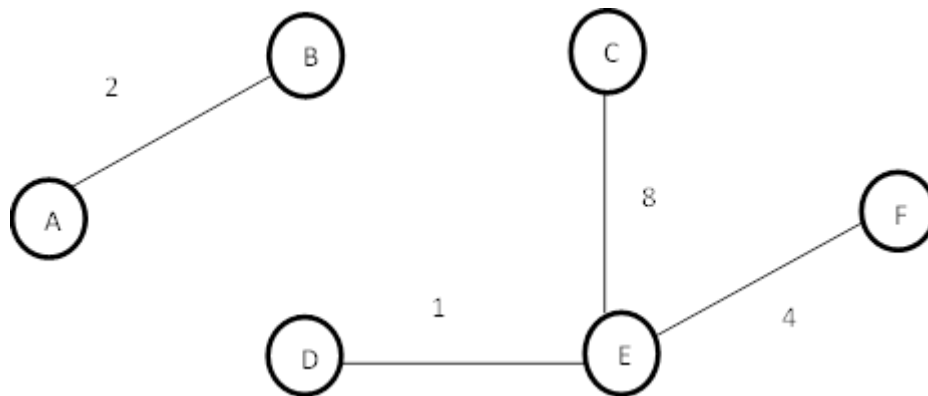


Figure 3.5: After the completion of first round

Now these values are inserted into the HDFS, Blue tree and cost is updated, the queue is updated. And first round is finished.

Next Round

Map: Output of the first round is input to the mapper. Mapper emits the intermediate keys in the format

$$[nid, \langle \{x, y\}, h(nid), w \rangle],$$

So these are followed:

$$\{\{A, B\}, \langle \{B, D\}, h(A), 5 \rangle\},$$

$$\{\{C, D, E, F\}, \langle \{D, B\}, 5 \rangle\}$$

Reduce: Now these values are read by the reducer which checks the condition; if $\text{find}(x) \neq \text{find}(y)$. So outputs are like,

$$\langle \{A, B, C, D, E, F\}, [h(A), 20] \rangle$$

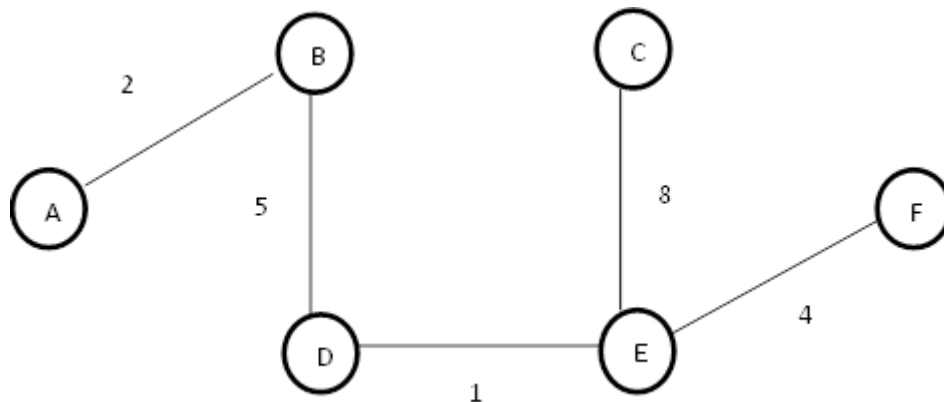


Figure 3.6: After the completion of second round, generated MST

We repeat, as much as round until $\text{size}(\text{queue}) > 1$. Here, in this example algorithm is terminated because after two rounds queue size became one.

3.3 Storage and performance complexities

There are several metrics that one can use to measure the efficiency of a MapReduce algorithm [38, 39]. These are following:

- t : the **number of rounds** (map-shuffle-reduce phases) that the algorithm uses.
- n_1, n_2, \dots the **reducer I/O sizes for round i** and n_{ij} is the size of the inputs and outputs for reducer j in round i .
- M_i : **the message complexity of round i** of the algorithm. It is the total size of the inputs and outputs for reducers in round i , that is, $M_i = \sum_j M_{ij}$ and $M = \sum_{i=1}^t M_i$ for the entire algorithm.
- I_i : the internal running time for round i . It is the maximum internal running time taken by a reducer in round i , where we assume $I_i \geq \max_j (n_{ij})$, since a reducer must have a running time that is at least the size of its inputs and outputs. For entire algorithm internal running time, $I = \sum_{i=1}^t r_i$.
- λ : the buffer size for reducers, that is, the maximum size of the working memory needed by a reducer to process its inputs and outputs (in addition to the storage used for the input itself), taken across all t rounds of the algorithm.
- L : the latency L of the shuffle network. It is the time that a mapper or reducer has to wait until it receives its first input in a given round.
- β : the bandwidth of the shuffle network. It is the number of elements in a MapReduce computation that can be delivered by the shuffle network in any time unit.

Thus the total running time, T , of an implementation of a MapReduce algorithm can be crudely characterized as follows:

$$T = O\left(\sum_{i=1}^t I_i + L + M_i/\beta\right)$$

$$= O(I + tL + M/\beta)$$

It is the **MapReduce running time**. For example, if there is D document which have n words. A simple word-count MapReduce algorithm has a worst-case performance of t being 1, M being $O(n)$, and I being $O(n)$; hence, its overall worst-case time performance is $O(n)$. Therefore, focusing exclusively on the number of rounds in a MapReduce algorithm can actually lead to inefficient algorithms. For example, if we consider the number of rounds, t , then the most efficient algorithm would always be one

that maps all the inputs to a single key and then has the reducer for this key perform a standard sequential algorithm to solve the problem. This approach would run in one round, but it would not use any parallelism. It would have a running time of $(\tau(n) + L + n/b)$ Where $\tau(n)$ is the running time of the sequential algorithm; hence, this MapReduce algorithm would be only as efficient as the best sequential algorithm.

3.3.1 Rounds/Time: Sequential round robin algorithm takes $O(m \log \log n)$ time complexity [20, 22]. The CREW PRAM model requires $\Omega(\log n)$ time [16, 37]. In MapReduce model complexity is calculated in number of rounds in which algorithm completes and provides the desired result. Proposed RRMST-MR (round robin minimum spanning tree algorithm in MapReduce) algorithm takes $O(\log n)$ rounds to generate MST, but in general, MST is generated in fewer rounds like in two or three.

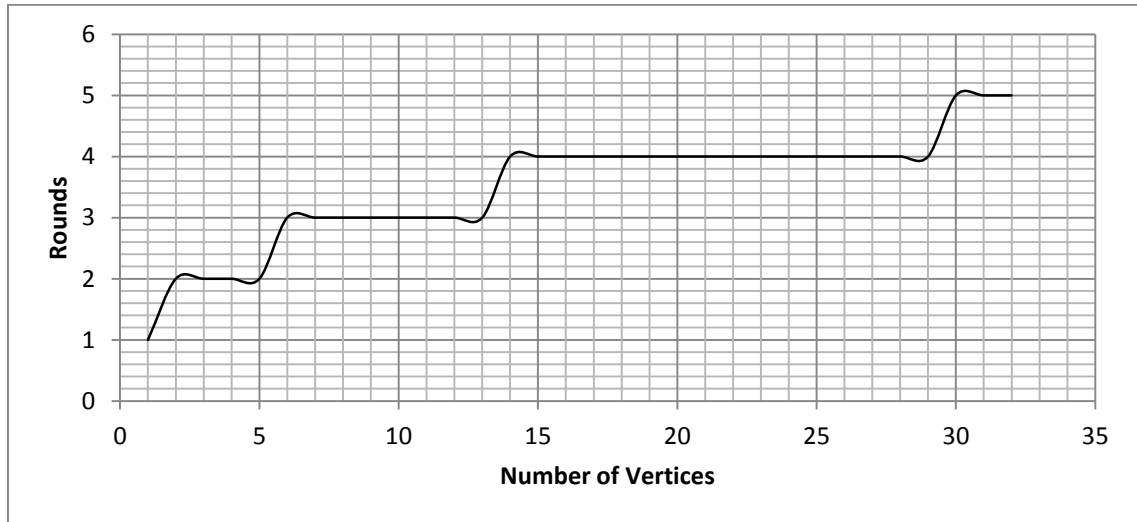


Figure 3.7: Graph vertices v/s rounds (initialization round is excluded)

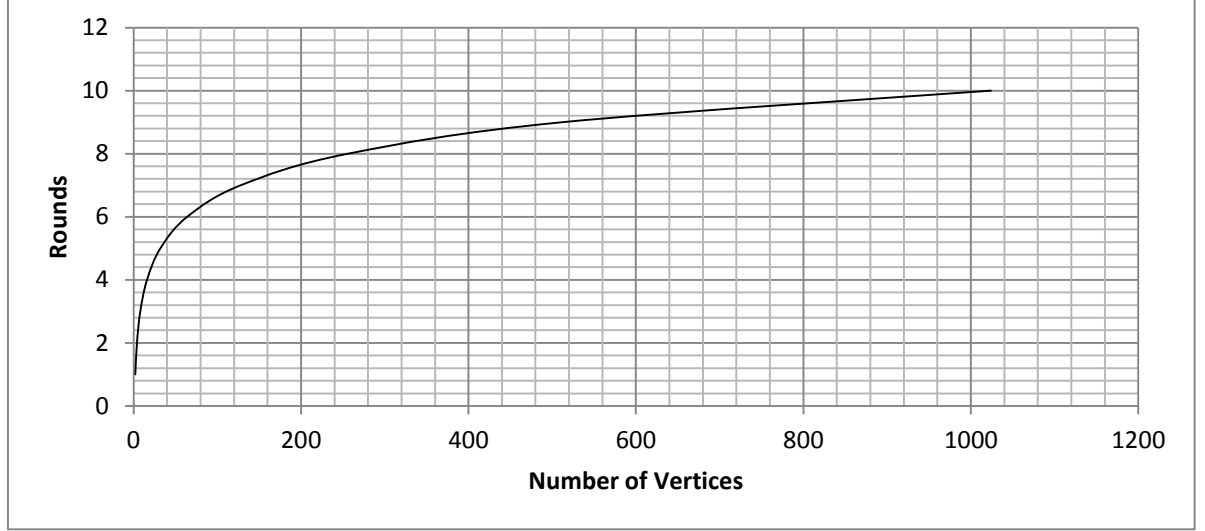


Figure 3.8: Graph rounds v/s vertices (when initialization round is included)

3.3.2 Memory: For an input of size N , and a sufficiently small $\varepsilon > 0$, there are $N^{1-\varepsilon}$ machines, each with $N^{1-\varepsilon}$ memory available for computation. As a result, the total amount of memory available to the entire system is $O(N^{2-2\varepsilon})$. An algorithm in MRC belongs to MRC^i if it runs in worst case $O(\log^i N)$ rounds [39]. Thus, when designing a MRC^0 algorithm there are three properties that need to be checked:

- **Machine Memory:** In each round the total memory used by a single machine is at most $O(N^{1-\varepsilon})$ bits.
- **Total Memory:** The total amount of data shuffled in any round is $O(N^{2-2\varepsilon})$ bits².
- **Rounds:** The number of rounds is a constant.

Let $G = (V, E)$ be an undirected graph, and denote by $n = |V|$ and $m = |E|$. We will call G , c -dense, if $m = n^{1+c}$ where $0 < c \leq 1$. In what follows, we assume that the machines have some limited memory η . We will assume that the number of available machines is $O(m/\eta)$. Notice that the number of machines is just the number required to fit the input on all of the machines simultaneously. All of our algorithms will consider the case where $\eta = n^{1+\varepsilon}$ for some $\varepsilon > 0$. For a constant, the algorithms we define will take a constant number of rounds and lie in MRC^0 [37, 38], beating the $\Omega(\log n)$ running time provided by the PRAM simulation constructions (see Theorem 7.1 in [37]). However, even when $\eta = O(n)$ our algorithms will run in $O(\log n)$ rounds. This exposes the memory vs. rounds tradeoff

since most of the algorithms presented take fewer rounds as the memory per machine increases.

The proposed algorithm is a semi-external memory algorithm. If a graph G is d-dense, $m = n^{1+d}$; and if $k = ND^{1/2}$, for some $d \geq 0$. It is the high probability that required working memory for any mapper or reducer is $O(|m|^{1-\epsilon})$, for $\epsilon > 0$. It reaches to $O(m)$ in the last round. At any time working memory is the size of the heap of a particular vertex set.

3.3.3 Communication cost: Communication cost is the number of communication between any two nodes in the framework or cluster. We are considering only the part of communication when reducers communicate with the HDFS when compact function is applied in the reducers. This is the extra cost because of such type of algorithms where we need to interact with the HDFS in the middle of processing of any round and this communication cost is around $O(n)$.

$$T(n) = \left(n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + \frac{n}{2^{k+1}} + \frac{n}{2^k} \right)$$

$$T(n) = n \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{2^{k+1}} + \frac{1}{2^k} \right)$$

$$T(n) = 2n \left(1 - \left(\frac{1}{2^k} \right) \right)$$

$$; \text{where } n = 2^k, k = \log n$$

$$T(n) = 2(n - 1)$$

$$T(n) = O(n)$$

3.3.4 I/O Cost: I/O cost has linear time complexity.

$$T(n) = \left(\left(n + \frac{n}{2} \right) + \left(\frac{n}{2} + \frac{n}{4} \right) + \left(\frac{n}{4} + \frac{n}{8} \right) + \dots + \left(\frac{n}{2^{k+1}} + \frac{n}{2^k} \right) \right)$$

$$T(n) = n + n \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{2^{k+1}} + \frac{1}{2^k} \right) + \frac{n}{2^k}$$

$$T(n) = n + 2n \left(1 - \left(\frac{1}{2^k} \right) \right) + \frac{n}{2^k}$$

$$; \text{where } n = 2^k, k = \log n$$

$$T(n) = 3n - 1$$

$$T(n) = O(n)$$

Chapter 4

IMPLEMENTATION

To exemplify the flexibility of the MapReduce theory and give some ideas how to use the framework some example's program are introduced. I have implemented this algorithm in a MapReduce framework in Hadoop. In the first section I introduced how to install Hadoop and how to configure a machine to run Hadoop. In the second section I introduced a MapReduce example, i.e. WordCount. In the third section we implemented the minimum spanning tree algorithm in MapReduce framework. Here, we will implement a minimum spanning tree in a MapReduce framework for single mapper and reducer and then we will extend our Round Robin Minimum spanning tree algorithm for multiple Mappers and Reducer.

We are implementing our basic algorithm on Hadoop Framework (Hadoop-2.2.0) on Linux (Ubuntu-12.04) platform. Hadoop is an open source framework written in Java language.

The HDFS provides the storage and the MapReduce executes the programs. Hadoop runs in three different modes:

1. Standalone mode- Standalone mode is appropriate for running MapReduce programs during development, since it is easy to test and debug the programs.
2. Pseudo-distributed mode: used for an emulated “cluster” for single computer; good for testing purpose.
3. Fully distributed operation mode: is used for a fully cluster.

The proposed algorithm is implemented on standalone mode.

4.1 Installation of Hadoop:

Prerequisite for Hadoop installation are, [11]

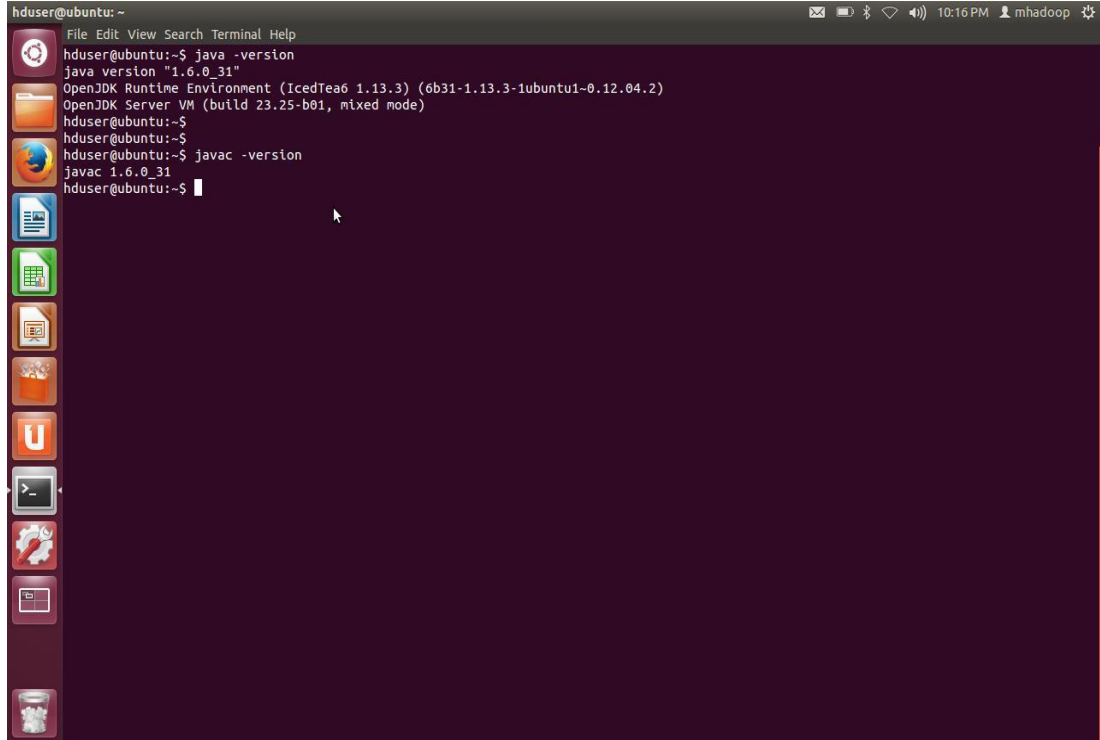
- 1> **Install Java:** Hadoop requires Java 1.5 above version. However, Java 1.6 is recommended for running Hadoop. To install Java these commands are used;
\$ sudo apt-get update
\$ sudo apt-get install sun-java6-jdk

```
$ sudo update-java-alternatives -s java-6-sun
```

If we want to check the version of java and javac following commands, (see in figure 4.1)

```
$ java -version
```

```
$javac -version
```



```
hduser@ubuntu: ~  
File Edit View Search Terminal Help  
hduser@ubuntu:~$ java -version  
java version "1.6.0_31"  
OpenJDK Runtime Environment (IcedTea6 1.13.3) (6b31-1.13.3-1ubuntu0-12.04.2)  
OpenJDK Server VM (build 23.25-b01, mixed mode)  
hduser@ubuntu:~$  
hduser@ubuntu:~$  
hduser@ubuntu:~$ javac -version  
javac 1.6.0_31  
hduser@ubuntu:~$
```

Figure 4.1: Checking java and javac version

2> **Adding a user** (dedicated Hadoop system): I am using a dedicated Hadoop user account for running Hadoop.

```
$ sudo addgroup mhadoop
```

```
$ sudo adduser --ingroup mhadoop hduser
```

3> **Configuring SSH:** Hadoop requires SSH access to manage its nodes (remote machines and local machine). SSH must be installed and SSHD must be running to use the Hadoop scripts that manage remote Hadoop daemons. So we have to generate the SSH key for the **hduser** user. If the openssh - server is not installed on the system, then first install it (apt get install openssh-server). Now generate the ssh

keys for hduser and test by following commands,

```
hduser@ubuntu:~$ ssh-keygen -t rsa -P ""
```

```
hduser@ubuntu: ~$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys
```

```
hduser@ubuntu:~$ ssh localhost
```

4> **Disabling IPv6:** To disable IPv6 on Ubuntu 12.04 LTS, open */etc/sysctl.conf* in the editor and add the following lines to the end of the file:

```
# disables ipv6
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

Steps to Install Hadoop:

1> Download Hadoop from the Apache Download Mirrors. There are many versions of Hadoop that are available on apache.org site. We have installed the stable version of Hadoop-2.2.0 which is available on site as Hadoop-2.2.0.tar.gz file and extract the contents of the Hadoop package to a location. I choose /usr/local. Then change the ownership of all files to the **hduser** user and **hadoop** group.

```
$ cd /usr/local
$ sudo tar xzf Hadoop-2.2.0.tar.gz
$ sudo mv Hadoop-2.2.0 hadoop
$ sudo chown -R hduser:Hadoop Hadoop
```

2> Update \$HOME/.bashrc

```
# Set Hadoop-related environment variables
export HADOOP_HOME=/usr/local/hadoop
# Set JAVA_HOME
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk
# Some functions and convenient aliases for running Hadoop-related
commands
unalias fs &> /dev/null
```



```

alias fs="hadoop fs"
unalias hls &> /dev/null
alias hls="fs -ls"
lzohead () {
hadoop fs -cat $1 | lzop -dc | head -1000 | less
}
# Add Hadoop bin/ directory to PATH
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin

```

3> Configuration:

The required environment variable we have to configure for Hadoop is `JAVA_HOME`. So open **conf/hadoop.env.sh** and update the following line:

```
export JAVA_HOME=/usr/lib/jvm/java-6-openjdk
```

We will configure the directory where Hadoop will store its data files, the network ports. We will use the directory `/app/hadoop/tmp` and change the permissions and required ownerships.

```

$ sudo mkdir -p /app/hadoop/tmp
$ sudo chown hduser:hadoop /app/hadoop/tmp
$ sudo chmod 750 /app/hadoop/tmp

```

Now add the following snippets in the respective configuration XML file between the tags `<configuration> ... </configuration>`.

In file **conf/core-site.xml**:

```

<property>
  <name>hadoop.tmp.dir</name>
  <value>/app/hadoop/tmp</value>
  <description>A base for other temporary directories.</description>
</property>
<property>
  <name>fs.default.name</name>
  <value>hdfs://localhost:54310</value>

```

```
<description>
```

The name of the default file system.

The uri's scheme determines the config property (fs.SCHEME.impl)

naming

the FileSystem implementation class.

```
</description>
```

```
</property>
```

In file **conf/mapred-site.xml**:

```
<property>
```

```
<name>mapred.job.tracker</name>
```

```
<value>localhost:54311</value>
```

<description>The host and port that the MapReduce job tracker runs at. If "local", then jobs are run in-process as a single map and reduce task.

```
</description>
```

```
</property>
```

In file **conf/hdfs-site.xml**:

```
<property>
```

```
<name>dfs.replication</name>
```

```
<value>1</value>
```

```
<description>
```

Default block replication. </description>

```
</property>
```

4> **Formatting the HDFS :**

The first step to starting up Hadoop installation is formatting the Hadoop filesystem which is implemented on top of the local filesystem of our "cluster". We need to do this the first time we set up a Hadoop cluster. To format the filesystem, run the following command (see figure)

```
hduser@ubuntu:~$ hdfs namenode -format
```

5> Starting single node cluster

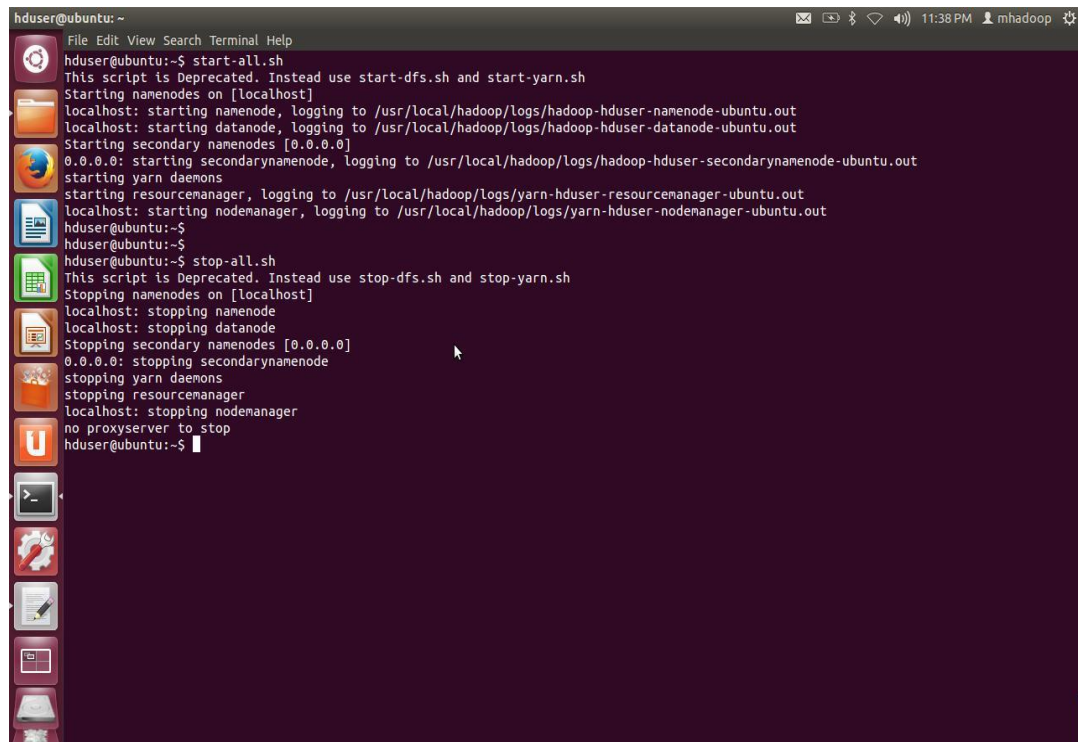
Run the command (see figure 4.2)

```
hduser@ubuntu:~$ hadoop start-all.sh
```

6> Stop single node cluster

Run the command (see figure 4.2)

```
hduser@ubuntu:~$ hadoop start-all.sh
```



```
hduser@ubuntu: ~
File Edit View Search Terminal Help
hduser@ubuntu:~$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /usr/local/hadoop/logs/hadoop-hduser-namenode-ubuntu.out
localhost: starting datanode, logging to /usr/local/hadoop/logs/hadoop-hduser-datanode-ubuntu.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/logs/hadoop-hduser-secondarynamenode-ubuntu.out
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop/logs/yarn-hduser-resourcemanager-ubuntu.out
localhost: starting nodemanager, logging to /usr/local/hadoop/logs/yarn-hduser-nodemanager-ubuntu.out
hduser@ubuntu:~$
hduser@ubuntu:~$
hduser@ubuntu:~$ stop-all.sh
This script is Deprecated. Instead use stop-dfs.sh and stop-yarn.sh
Stopping namenodes on [localhost]
localhost: stopping namenode
localhost: stopping datanode
Stopping secondary namenodes [0.0.0.0]
0.0.0.0: stopping secondarynamenode
stopping yarn daemons
stopping resourcemanager
localhost: stopping nodemanager
no proxyserver to stop
hduser@ubuntu:~$
```

Figure 4.2: Starting and stopping of hadoop cluster

7> Running a MapReduce job:

We will use the WordCount example, which reads text files and counts how often words occur. The input is text files and the output is text files, each line of which contains a word and the count of how often it occurred, separated by a tab. To do this we have to follow following steps:

7.1 Write a MapReduce program and save as a java file: WordCount.java

```
package mywordcount;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {
    public static class Map extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }
    }

    public static class Reduce extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
```

```

        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}

public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    JobClient.runJob(conf);
}
}

```

7.2 Compile the java code (see figure 4.3)

7.3 Make a jar file (see figure 4.4)

7.4 Copy local example data into the HDFS: Before we run the actual MapReduce job, we first have to copy the files from our local file system to Hadoop's HDFS by the following command, (see figure 4.4)

```
hduser@ubuntu:~ $hadoop dfs -copyFromLocal LocalFilePath inputPathHDFS
```

7.5 Run the jar file (see figure 4.5)

7.6 Retrieve the job result from the HDFS (see figure 4.6)

```

hduser@ubuntu:~
File Edit View Search Terminal Help
hduser@ubuntu:~$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /usr/local/hadoop/logs/hadoop-hduser-namenode-ubuntu.out
localhost: starting datanode, logging to /usr/local/hadoop/logs/hadoop-hduser-datanode-ubuntu.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/logs/hadoop-hduser-secondarynamenode-ubuntu.out
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop/logs/yarn-hduser-resourcemanager-ubuntu.out
localhost: starting nodemanager, logging to /usr/local/hadoop/logs/yarn-hduser-nodemanager-ubuntu.out
hduser@ubuntu:~$ sudo javac -Xlint -classpath $HADOOP_HOME/share/hadoop/common/hadoop-common-2.2.0.jar:$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.2.0.jar:$HADOOP_HOME/share/hadoop/common/lib/commons-cli-1.2.jar -d /home/hduser/Desktop/msharma/wclass /home/hduser/Desktop/msharma/WordCount.java
[sudo] password for hduser:
/usr/local/hadoop/share/hadoop/common/hadoop-common-2.2.0.jar(org/apache/hadoop/fs/Path.class): warning: Cannot find annotation method 'value()' in type 'org.apache.hadoop.classification.InterfaceAudience.LimitedPrivate': class file for org.apache.hadoop.classification.InterfaceAudience not found
1 warning
hduser@ubuntu:~$

```

Figure 4.3: Compiling a MapReduce program code

```

hduser@ubuntu:~
File Edit View Search Terminal Help
hduser@ubuntu:~$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /usr/local/hadoop/logs/hadoop-hduser-namenode-ubuntu.out
localhost: starting datanode, logging to /usr/local/hadoop/logs/hadoop-hduser-datanode-ubuntu.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/logs/hadoop-hduser-secondarynamenode-ubuntu.out
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop/logs/yarn-hduser-resourcemanager-ubuntu.out
localhost: starting nodemanager, logging to /usr/local/hadoop/logs/yarn-hduser-nodemanager-ubuntu.out
hduser@ubuntu:~$ sudo javac -Xlint -classpath $HADOOP_HOME/share/hadoop/common/hadoop-common-2.2.0.jar:$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.2.0.jar:$HADOOP_HOME/share/hadoop/common/lib/commons-cli-1.2.jar -d /home/hduser/Desktop/msharma/wclass /home/hduser/Desktop/msharma/WordCount.java
[sudo] password for hduser:
/usr/local/hadoop/share/hadoop/common/hadoop-common-2.2.0.jar(org/apache/hadoop/fs/Path.class): warning: Cannot find annotation method 'value()' in type 'org.apache.hadoop.classification.InterfaceAudience.LimitedPrivate': class file for org.apache.hadoop.classification.InterfaceAudience not found
1 warning
hduser@ubuntu:~$ jar -cvf /home/hduser/Desktop/wordcount.jar -C /home/hduser/Desktop/msharma/wclass / .
added manifest
adding: mywordcount/(in = 0) (out= 0)(stored 0%)
adding: mywordcount/WordCount.class(in = 1552) (out= 752)(deflated 51%)
adding: mywordcount/WordCount$Map.class(in = 1942) (out= 802)(deflated 58%)
adding: mywordcount/WordCount$Reduce.class(in = 1615) (out= 651)(deflated 59%)
hduser@ubuntu:~$ hdfs dfs -copyFromLocal /home/hduser/Desktop/msharma/w.txt /in/inputfile
hduser@ubuntu:~$ hdfs dfs -cat /in/inputfile
Management is a set of processes that can keep a complicated system of people and technology running smoothly.
hduser@ubuntu:~$ hadoop jar /home/hduser/Desktop/wordcount.jar mywordcount.WordCount /in/inputfile /out/outputfile
14/04/30 23:18:07 INFO Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id
14/04/30 23:18:07 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
14/04/30 23:18:07 INFO jvm.JvmMetrics: Cannot initialize JVM Metrics with processName=JobTracker, sessionId= - already initialized
14/04/30 23:18:07 WARN mapreduce.JobSubmitter: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
14/04/30 23:18:07 INFO mapred.FileInputFormat: Total input paths to process : 1
14/04/30 23:18:07 INFO mapreduce.JobSubmitter: number of splits:1
14/04/30 23:18:07 INFO Configuration.deprecation: user.name is deprecated. Instead, use mapreduce.job.user.name
14/04/30 23:18:07 INFO Configuration.deprecation: mapred.jar is deprecated. Instead, use mapreduce.job.jar
14/04/30 23:18:07 INFO Configuration.deprecation: mapred.output.value.class is deprecated. Instead, use mapreduce.job.output.value.class
14/04/30 23:18:07 INFO Configuration.deprecation: mapred.job.name is deprecated. Instead, use mapreduce.job.name
14/04/30 23:18:07 INFO Configuration.deprecation: mapred.input.dir is deprecated. Instead, use mapreduce.input.fileinputformat.inputdirs

```

Figure 4.4: Making a jar file of classes and execution of a program

```

hduser@ubuntu:~
File Edit View Search Terminal Help
14/04/30 23:18:10 INFO mapreduce.Job: Job job_local1366556774_0001 completed successfully
14/04/30 23:18:10 INFO mapreduce.Job: Counters: 32
File System Counters
  FILE: Number of bytes read=6732
  FILE: Number of bytes written=378536
  FILE: Number of read operations=0
  FILE: Number of large read operations=0
  FILE: Number of write operations=0
  HDFS: Number of bytes read=222
  HDFS: Number of bytes written=138
  HDFS: Number of read operations=13
  HDFS: Number of large read operations=0
  HDFS: Number of write operations=4
Map-Reduce Framework
  Map input records=1
  Map output records=18
  Map output bytes=183
  Map output materialized bytes=208
  Input split bytes=87
  Combine input records=18
  Combine output records=16
  Reduce input groups=16
  Reduce shuffle bytes=0
  Reduce input records=16
  Reduce output records=16
  Spilled Records=32
  Shuffled Maps =0
  Failed Shuffles=0
  Merged Map outputs=0
  GC time elapsed (ms)=0
  CPU time spent (ms)=0
  Physical memory (bytes) snapshot=0
  Virtual memory (bytes) snapshot=0
  Total committed heap usage (bytes)=396886016
File Input Format Counters
  Bytes Read=111
File Output Format Counters
  Bytes Written=138
hduser@ubuntu:~$ hdfs dfs -cat /out/outputfile
cat: '/out/outputfile': is a directory
hduser@ubuntu:~$

```

Figure 4.5: Output of the WordCount MapReduce job

```

hduser@ubuntu:~
File Edit View Search Terminal Help
hduser@ubuntu:~$ hdfs dfs -cat /out/outputfile/*
Management 1
a 2
and 1
can 1
complicated 1
is 1
keep 1
of 2
people 1
processes 1
running 1
set 1
smoothly. 1
system 1
technology 1
that 1
hduser@ubuntu:~$

```

Figure 4.6: Retrieving the output from the HDFS

Example: calculate the value of pi

```
hduser@ubuntu:~  
File Edit View Search Terminal Help  
hduser@ubuntu:~$ hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.2.0.jar pi 2 10  
Number of Maps = 2  
Samples per Map = 10  
Wrote input for Map #0  
Wrote input for Map #1  
Starting Job  
14/04/30 23:25:46 INFO Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id  
14/04/30 23:25:46 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=  
14/04/30 23:25:46 INFO input.FileInputFormat: Total input paths to process : 2  
14/04/30 23:25:46 INFO mapreduce.JobSubmitter: number of splits:2  
14/04/30 23:25:46 INFO Configuration.deprecation: user.name is deprecated. Instead, use mapreduce.job.user.name  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.jar is deprecated. Instead, use mapreduce.job.jar  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.map.tasks.speculative.execution is deprecated. Instead, use mapreduce.map.spe  
culative  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.reduce.tasks is deprecated. Instead, use mapreduce.job.reduces  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.output.value.class is deprecated. Instead, use mapreduce.job.output.value.cla  
ss  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.reduce.tasks.speculative.execution is deprecated. Instead, use mapreduce.redu  
ce.speculative  
14/04/30 23:25:46 INFO Configuration.deprecation: mapreduce.map.class is deprecated. Instead, use mapreduce.job.map.class  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.job.name is deprecated. Instead, use mapreduce.job.name  
14/04/30 23:25:46 INFO Configuration.deprecation: mapreduce.reduce.class is deprecated. Instead, use mapreduce.job.reduce.class  
14/04/30 23:25:46 INFO Configuration.deprecation: mapreduce.inputformat.class is deprecated. Instead, use mapreduce.job.inputformat.cl  
ass  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.input.dir is deprecated. Instead, use mapreduce.input.fileinputformat.inputd  
ir  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.output.dir is deprecated. Instead, use mapreduce.output.fileoutputformat.out  
putdir  
14/04/30 23:25:46 INFO Configuration.deprecation: mapreduce.outputformat.class is deprecated. Instead, use mapreduce.job.outputformat.  
class  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.map.tasks is deprecated. Instead, use mapreduce.job.maps  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.output.key.class is deprecated. Instead, use mapreduce.job.output.key.class  
14/04/30 23:25:46 INFO Configuration.deprecation: mapred.working.dir is deprecated. Instead, use mapreduce.job.working.dir
```

Figure 4.7: Execution of pi MapReduce example

```
hduser@ubuntu:~  
File Edit View Search Terminal Help  
14/04/30 23:25:18 INFO mapreduce.Job: Job job_local1718134111_0001 completed successfully  
14/04/30 23:25:18 INFO mapreduce.Job: Counters: 32  
File System Counters  
FILE: Number of bytes read=3054719  
FILE: Number of bytes written=5083260  
FILE: Number of read operations=0  
FILE: Number of large read operations=0  
FILE: Number of write operations=0  
HDFS: Number of bytes read=7670  
HDFS: Number of bytes written=13195  
HDFS: Number of read operations=253  
HDFS: Number of large read operations=0  
HDFS: Number of write operations=135  
Map-Reduce Framework  
Map input records=10  
Map output records=20  
Map output bytes=180  
Map output materialized bytes=280  
Input split bytes=1470  
Combine input records=0  
Combine output records=0  
Reduce input groups=2  
Reduce shuffle bytes=0  
Reduce input records=20  
Reduce output records=0  
Spilled Records=40  
Shuffled Maps=0  
Failed Shuffles=0  
Merged Map outputs=0  
GC time elapsed (ms)=279  
CPU time spent (ms)=0  
Physical memory (bytes) snapshot=0  
Virtual memory (bytes) snapshot=0  
Total committed heap usage (bytes)=3518496768  
File Input Format Counters  
Bytes Read=180  
File Output Format Counters  
Bytes Written=97  
Job Finished in 4.08 seconds  
Estimated value of Pi is 3.14800000000000000000  
hduser@ubuntu:~$
```

Figure 4.7: Execution of pi MapReduce example

4.2 Proposed MST Algorithm

We have written the code in java language and save it as Mst.java.

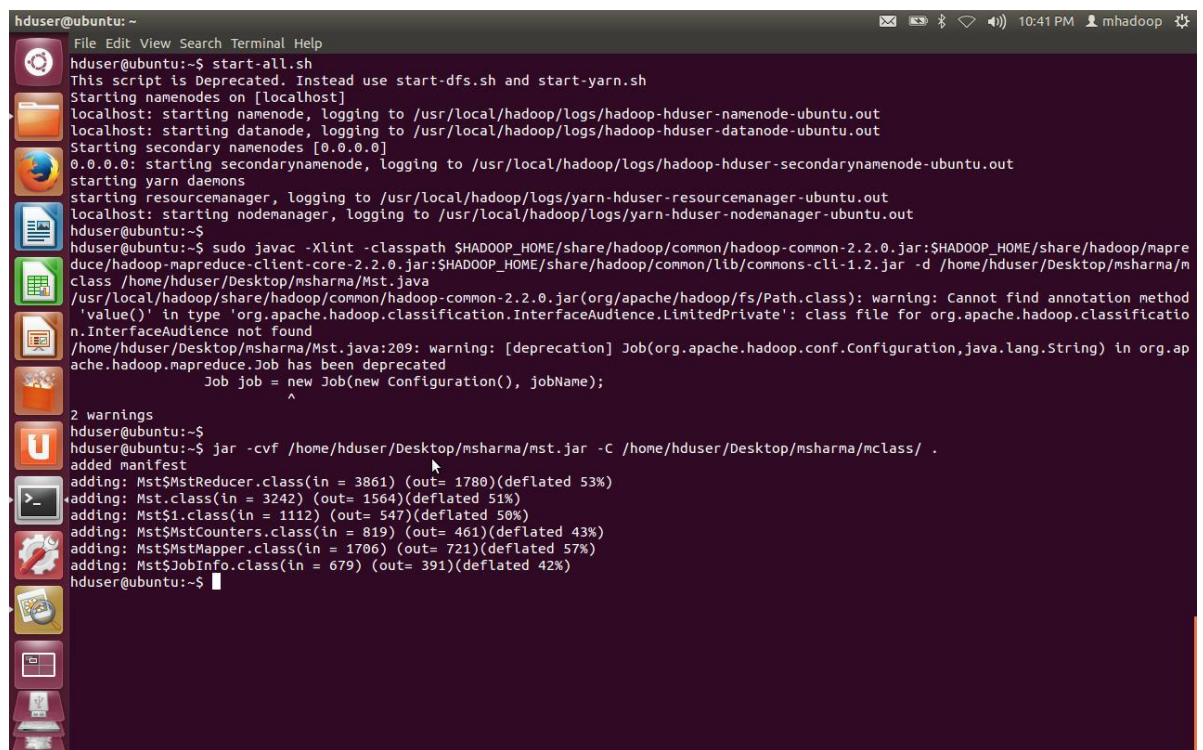
Now compile this file by javac compiler.

Now, make a jar file and put all the classes into this jar file.

Now, start the hadoop cluster and insert the local data files (Graph as a input) into the HDFS (hadoop distributed file system).

Now, run that jar file by the command.

Retrieve the output to local from the HDFS.



```
hduser@ubuntu: ~
File Edit View Search Terminal Help
hduser@ubuntu:~$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [localhost]
localhost: starting namenode, logging to /usr/local/hadoop/logs/hadoop-hduser-namenode-ubuntu.out
localhost: starting datanode, logging to /usr/local/hadoop/logs/hadoop-hduser-datanode-ubuntu.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/logs/hadoop-hduser-secondarynamenode-ubuntu.out
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop/logs/yarn-hduser-resourcemanager-ubuntu.out
localhost: starting nodemanager, logging to /usr/local/hadoop/logs/yarn-hduser-nodemanager-ubuntu.out
hduser@ubuntu:~$
hduser@ubuntu:~$ sudo javac -Xlint -classpath $HADOOP_HOME/share/hadoop/common/hadoop-common-2.2.0.jar:$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.2.0.jar:$HADOOP_HOME/share/hadoop/common/lib/commons-cli-1.2.jar -d /home/hduser/Desktop/msharma/mclass /home/hduser/Desktop/msharma/Mst.java
/usr/local/hadoop/share/hadoop/common/hadoop-common-2.2.0.jar(org/apache/hadoop/fs/Path.class): warning: Cannot find annotation method 'value()' in type 'org.apache.hadoop.classification.InterfaceAudience.LimitedPrivate': class file for org.apache.hadoop.classification.InterfaceAudience not found
/home/hduser/Desktop/msharma/Mst.java:209: warning: [deprecation] Job(org.apache.hadoop.conf.Configuration,java.lang.String) in org.apache.hadoop.mapreduce.Job has been deprecated
    Job job = new Job(new Configuration(), jobName);
                ^
2 warnings
hduser@ubuntu:~$
hduser@ubuntu:~$ jar -cvf /home/hduser/Desktop/msharma/mst.jar -C /home/hduser/Desktop/msharma/mclass/ .
added manifest
adding: Mst$MstReducer.class(in = 3861) (out= 1780)(deflated 53%)
adding: Mst.class(in = 3242) (out= 1564)(deflated 51%)
adding: Mst$I.class(in = 1112) (out= 547)(deflated 50%)
adding: Mst$MstCounters.class(in = 819) (out= 461)(deflated 43%)
adding: Mst$MstMapper.class(in = 1706) (out= 721)(deflated 57%)
adding: Mst$JobInfo.class(in = 679) (out= 391)(deflated 42%)
hduser@ubuntu:~$
```

Figure 4.8: Compiling and making a jar file

```

hduser@ubuntu:~
File Edit View Search Terminal Help
hduser@ubuntu:~$ hdfs dfs -copyFromLocal /home/hduser/Desktop/msharma/m.txt /in/insertInHDFS
hduser@ubuntu:~$
hduser@ubuntu:~$ hadoop jar /home/hduser/Desktop/msharma/mst.jar Mst /in/insertInHDFS /out/m20
14/05/09 22:47:58 INFO Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id
14/05/09 22:47:58 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
14/05/09 22:47:58 WARN mapreduce.JobSubmitter: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
14/05/09 22:47:58 INFO input.FileInputFormat: Total input paths to process : 1
14/05/09 22:47:59 INFO mapreduce.JobSubmitter: number of splits:1
14/05/09 22:47:59 INFO Configuration.deprecation: user.name is deprecated. Instead, use mapreduce.job.user.name
14/05/09 22:47:59 INFO Configuration.deprecation: mapred.jar is deprecated. Instead, use mapreduce.job.jar
14/05/09 22:47:59 INFO Configuration.deprecation: mapred.reduce.tasks is deprecated. Instead, use mapreduce.job.reduces
14/05/09 22:47:59 INFO Configuration.deprecation: mapred.output.value.class is deprecated. Instead, use mapreduce.job.output.value.class
14/05/09 22:47:59 INFO Configuration.deprecation: mapreduce.map.class is deprecated. Instead, use mapreduce.job.map.class
14/05/09 22:47:59 INFO Configuration.deprecation: mapred.job.name is deprecated. Instead, use mapreduce.job.name
14/05/09 22:47:59 INFO Configuration.deprecation: mapreduce.reduce.class is deprecated. Instead, use mapreduce.job.reduce.class
14/05/09 22:47:59 INFO Configuration.deprecation: mapred.input.dir is deprecated. Instead, use mapreduce.input.fileinputformat.inputdir
14/05/09 22:47:59 INFO Configuration.deprecation: mapred.output.dir is deprecated. Instead, use mapreduce.output.fileoutputformat.outputdir
14/05/09 22:47:59 INFO Configuration.deprecation: mapred.map.tasks is deprecated. Instead, use mapreduce.job.maps
14/05/09 22:47:59 INFO Configuration.deprecation: mapred.output.key.class is deprecated. Instead, use mapreduce.job.output.key.class
14/05/09 22:47:59 INFO Configuration.deprecation: mapred.working.dir is deprecated. Instead, use mapreduce.job.working.dir
14/05/09 22:48:00 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_local426786921_0001
14/05/09 22:48:00 WARN conf.Configuration: file:/app/hadoop/tmp/mapred/staging/hduser426786921/.staging/job_local426786921_0001/job.xml:an attempt to override final parameter: mapreduce.job.end-notification.max.retry.interval; Ignoring.
14/05/09 22:48:00 WARN conf.Configuration: file:/app/hadoop/tmp/mapred/staging/hduser426786921/.staging/job_local426786921_0001/job.xml:an attempt to override final parameter: mapreduce.job.end-notification.max.attempts; Ignoring.
14/05/09 22:48:00 WARN conf.Configuration: file:/app/hadoop/tmp/mapred/local/LocalRunner/hduser/job_local426786921_0001.xml:an attempt to override final parameter: mapreduce.job.end-notification.max.retry.interval; Ignoring.
14/05/09 22:48:00 WARN conf.Configuration: file:/app/hadoop/tmp/mapred/local/LocalRunner/hduser/job_local426786921_0001.xml:an attempt to override final parameter: mapreduce.job.end-notification.max.attempts; Ignoring.
14/05/09 22:48:00 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
14/05/09 22:48:00 INFO mapreduce.Job: Running job: job_local426786921_0001
14/05/09 22:48:00 INFO mapred.LocalJobRunner: OutputCommitter set in config null
14/05/09 22:48:00 INFO mapred.LocalJobRunner: OutputCommitter is org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter
14/05/09 22:48:01 INFO mapred.LocalJobRunner: Waiting for map tasks
14/05/09 22:48:01 INFO mapred.LocalJobRunner: Starting task: attempt_local426786921_0001_m_000000_0
14/05/09 22:48:01 INFO mapred.Task: Using ResourcecalculatorProcessTree : [ ]
14/05/09 22:48:01 INFO mapred.MapTask: Processing split: hdfs://localhost:54310/in/insertInHDFS:0+63

```

Figure 4.9: Insertion of the data files into HDFS and Running of the job

```

hduser@ubuntu:~
File Edit View Search Terminal Help
hduser@ubuntu:~$ hdfs dfs -cat /in/insertInHDFS
6      A      C
5      C      E
3      C      F
2      C      D
1      B      C
2      B      D
4      A      D
4      E      F
4      D      F
5      A      B
hduser@ubuntu:~$ hadoop jar /home/hduser/Desktop/msharma/mst.jar Mst /in/insertInHDFS /out/m26
14/05/16 16:58:52 INFO Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id
14/05/16 16:58:52 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
14/05/16 16:58:52 WARN mapreduce.JobSubmitter: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
14/05/16 16:58:52 INFO input.FileInputFormat: Total input paths to process : 1
14/05/16 16:58:52 INFO mapreduce.JobSubmitter: number of splits:1
14/05/16 16:58:52 INFO Configuration.deprecation: user.name is deprecated. Instead, use mapreduce.job.user.name
14/05/16 16:58:52 INFO Configuration.deprecation: mapred.jar is deprecated. Instead, use mapreduce.job.jar
14/05/16 16:58:52 INFO Configuration.deprecation: mapred.reduce.tasks is deprecated. Instead, use mapreduce.job.reduces
14/05/16 16:58:52 INFO Configuration.deprecation: mapred.output.value.class is deprecated. Instead, use mapreduce.job.output.value.class
14/05/16 16:58:52 INFO Configuration.deprecation: mapreduce.map.class is deprecated. Instead, use mapreduce.job.map.class
14/05/16 16:58:52 INFO Configuration.deprecation: mapred.job.name is deprecated. Instead, use mapreduce.job.name
14/05/16 16:58:52 INFO Configuration.deprecation: mapreduce.reduce.class is deprecated. Instead, use mapreduce.job.reduce.class
14/05/16 16:58:52 INFO Configuration.deprecation: mapred.input.dir is deprecated. Instead, use mapreduce.input.fileinputformat.inputdir
14/05/16 16:58:52 INFO Configuration.deprecation: mapred.output.dir is deprecated. Instead, use mapreduce.output.fileoutputformat.outputdir
14/05/16 16:58:52 INFO Configuration.deprecation: mapred.map.tasks is deprecated. Instead, use mapreduce.job.maps
14/05/16 16:58:52 INFO Configuration.deprecation: mapred.output.key.class is deprecated. Instead, use mapreduce.job.output.key.class
14/05/16 16:58:52 INFO Configuration.deprecation: mapred.working.dir is deprecated. Instead, use mapreduce.job.working.dir
14/05/16 16:58:53 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_local54122444_0001
14/05/16 16:58:53 WARN conf.Configuration: file:/app/hadoop/tmp/mapred/staging/hduser54122444/.staging/job_local54122444_0001/job.xml:

```

Figure 4.10: Data in HDFS and execution of MST algorithm

```
hduser@ubuntu: ~  
File Edit View Search Terminal Help  
14/05/16 18:15:47 INFO mapreduce.Job: Counters: 33  
File System Counters  
  FILE: Number of bytes read=13728  
  FILE: Number of bytes written=383944  
  FILE: Number of read operations=0  
  FILE: Number of large read operations=0  
  FILE: Number of write operations=0  
  HDFS: Number of bytes read=126  
  HDFS: Number of bytes written=30  
  HDFS: Number of read operations=13  
  HDFS: Number of large read operations=0  
  HDFS: Number of write operations=4  
Map-Reduce Framework  
  Map input records=13  
  Map output records=10  
  Map output bytes=80  
  Map output materialized bytes=106  
  Input split bytes=103  
  Combine input records=0  
  Combine output records=0  
  Reduce input groups=6  
  Reduce shuffle bytes=0  
  Reduce input records=10  
  Reduce output records=5  
  Spilled Records=20  
  Shuffled Maps =0  
  Failed Shuffles=0  
  Merged Map outputs=0  
  GC time elapsed (ms)=0  
  CPU time spent (ms)=0  
  Physical memory (bytes) snapshot=0  
  Virtual memory (bytes) snapshot=0  
  Total committed heap usage (bytes)=396886016  
MstMstCounters  
  totalWeight=14  
File Input Format Counters  
  Bytes Read=63  
File Output Format Counters  
  Bytes Written=30  
The total weight of the MST is : 14  
hduser@ubuntu:~$
```

Figure 4.11: Output of the MST

CONCLUSION AND FUTURE WORK

5.1 Conclusion

MapReduce is a programming model to run the jobs parallel in the cluster and cloud. At the programmer's point of view, MapReduce hides the complexity of data distribution, load balancing and fault tolerance. MapReduce is a very good for processing huge data in a parallel way. It is the heart of cloud computing. We have decomposed Round Robin algorithm in MapReduce programming module to run on Hadoop framework, and analyzed that MapReduce provide a better approach to run jobs in parallel. In our work, Minimum Spanning Tree (MST) was taken as a basis for analyzing the efficiency and programmability of large scale graph processing because it is one of the most studied combinatorial problems and consists of different types of workings at each step. We give a MapReduce version of round robin minimum spanning tree algorithm. Our algorithm generates minimum spanning tree only in $O(\log n)$ rounds which is better than $\Omega(\log n)$ of PRAM algorithms and $O(m \log \log n)$ of sequential algorithm.

5.2 Future Work

In this project I configured Hadoop for a Standalone mode only. My Future work is to enhance the approach to standalone mode to pseudo distributed mode and fully distributed mode. This algorithm will work very well for the small graph problem. My future scope to remains to improve our optimized design patterns even further. For example, Partitioning could be done to cluster based on actual graph topology. We can extend this MapReduce so that reducer can interact with HDFS in between the processing.

REFERENCES

1. J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in Proceedings of the 6th symposium on Operating Systems Design and Implementation, USENIX Association OSDI '04, pp. 137–150, 2004.
2. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. Communications of the ACM, vol. 51.1, pp. 107-113, 2008
3. J. Dean and S. Ghemawat, “MapReduce: A flexible data processing tool,” Communications of the ACM, vol. 53.1, pp. 72–77, ACM, 2010.
4. S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles, SOSP 03, ACM, pp. 29–43, 2003.
5. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” ACM Transactions on Computer Systems, vol. 26, no. 2, pp. 401–426, ACM, 2008.
6. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “Above the clouds: A Berkeley view of cloud computing,” Tech. Rep. EECS-2009-28, UC Berkeley Reliable Adaptive Distributed Systems Laboratory, 2009.
7. Hadoop: the Definitive Guide by Tom White, First Edition
8. Hadoop tutorial: Website. <http://hadooptutorial.wikispaces.com>
9. Hadoop wiki - powered by. <http://wiki.apache.org/hadoop/PoweredBy>.
10. Apache. Hadoop. <http://lucene.apache.org/hadoop/,2006>
11. Apache Hadoop: Website. <http://hadoop.apache.org/>
12. Dis Disco: massive data minimal code <http://discoproject.org/>
13. Amazon Elastic Compute Cloud (Amazon EC2). <http://aws.amazon.com/ec2/>
14. R. E. Tarjan, Data Structures and Network Algorithms, Chapter 6, SIAM, Philadelphia, 1983.
15. R. L. Graham, P. Hell, On the history of the minimum spanning tree problem, Ann. of the History of Comp., 7, pp. 43-57, 1985.

16. CuKneyt F. Bazlamac, Khalil S. Hindi Minimum-weight spanning tree algorithms A survey and empirical study”, Computers & Operations Research 28, pp. 767-785, 2001
17. Boruvka O. O jisteHm probleHmu minimaH lnmHm. PraHca MoravskeH Pz’ mHrodove\deckeH Spolec’nosti, pp. 337-58, 1926 [in Czech].
18. J. B. Kruskal, On the shortest spanning sub tree of a graph and the travelling salesman problem, Proc. AMS, 7, pp. 48-50, 1956.
19. R. C. Prim, Shortest connection networks and some generalizations, Bell System Tech. Journal, 1957.
20. M. L. Fredman, R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM. 34 (3), pp. 596-615, 1987.
21. Yao A. An $O(|E| \log \log |V|)$ algorithm for "finding minimum spanning trees. Information Processing Letters, pp. 421-431, 1975.
22. Cheriton D, Tarjan RE. Finding minimum spanning trees. SIAM Journal on Computing, pp. 724-742, 1976
23. H. Gabow,T. Spencer and R. Rarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. Combinatorial, vol.6, no.2, pp.109-122, 1986.
24. P. N. Klein, R. E. Tarjan, A Randomized Linear-Time Algorithms for Finding Minimum Spanning Trees, Proc. 26th ACM STOC, pp. 9-15, 1994.
25. B. CHAZELLE, “A minimum spanning tree algorithm with inverse-Ackermann type complexity”, Journal of ACM, vol.47, no.6, pp.1028-1047, 2000.
26. S. Pettie, V. Ramachandran, A shortest path algorithm for real-weighted undirected graphs, SIAM J. Comp., vol. 34, no. 6, pp. 1398-1431, 2002
27. S.Chung, A. Condon. Parallel Implementation of Boruvka's Minimum Spanning Tree Algorithm. 10th International Parallel Processing Symposium (IPPS '96), pp.302, 1996.
28. R.Gallager, P. Humblet and P.Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. ACM Trans. Programming Languages and Systems, vol.5, no.1, pp.66-77, Jan. 1983.
29. S. Kang and D. A. Bader. An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs. In PPOPP’09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and practice of parallel programming, New York,

NY, USA, 2009.

30. K. W. Chong, Y. Han and T. W. Lam. Concurrent threads and optimal parallel minimum spanning tree algorithm. *Journal of ACM*, vol.48, pp.297-323, 2001.
31. R. S. Barr, R. V. Helgaon and J. L. Kennington. Minimal spanning trees: An empirical investigation of parallel algorithms. *Parallel Computing*, vol. 12, no. 1, pp. 45-52, October 1989.
32. Blelloch, Guy E., Maggs, Bruce M.: Parallel algorithms. In: *ACM Computing Surveys*, vol. 28, no. 1, pp. 51-54, 1996.
33. L. Page, S. Brin, R. Motwani and T. Winograd, "The PageRank citation ranking: Bringing order to the Web". *Proceedings of the 7th International World Wide Web Conference*, pp. 161-172, 1998.
34. Bahman Bahmani, Kaushik Chakrabarti, Dong Xin Fast Personalized PageRank on MapReduce, SIGMOD'11, Athens, Greece, June 12–16, 2011.
35. SangwonSeo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim and Seungryoul Maeng, HAMA: An Efficient Matrix Computation with the MapReduce Framework. In *IEEE Second International Conference on Cloud Computing Technology and Science*, pages 721-726, Jul. 2010.
36. J. Lin and M. Schatz, Design patterns for efficient graph algorithms in MapReduce. *MLG '10: Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pages 78-85, 2010.
37. H. Karloff, S. Suri and S. Vassilvitskii., "A model of computation for MapReduce." *Symposium on Discrete Algorithms (SODA)*, 2010.
38. Michael T. Goodrich, "Simulating parallel algorithms in the MapReduce framework with applications to parallel computational geometry." *Second Workshop on Massive Data Algorithmic (MASSIVE 2010)*, June 2010.
39. Silvio Lattanzi et al., "Filtering: A Method for Solving Graph Problems in MapReduce", *SPAA'11*, San Jose, California, USA, June 2011.
40. Steven J. Plimpton and Karen D. Devine, *MapReduce in MPI for Large-scale Graph Algorithms*, 29 Nov 2010 version To appear in special issue of *Parallel Computing* in 2011.
41. Bergantinos, Gustavo and Vidal-Puga, Juan, The folk solution and Boruvka's

algorithm in minimum cost spanning tree problems, MPRA Paper No. 17839, posted 13. October 2009.

42. Bin Wu, YaHong Du, Cloud-based Connected Component Algorithm, International Conference on Artificial Intelligence and Computational Intelligence, 2010.
43. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: A distributed Storage System for Structured Data. In OSDI 2006.
44. Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, D. Stott Parker, Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters, at SIGMOD'07, Beijing, China, June 12–14, 2007.
45. Tanenbaum and M. Van Steen, Distributed Systems: Principles and Paradigms, Prentice Hall, Pearson Education, USA, 2002.
46. Robert Endre Tarjan, "Data Structure and Algorithms", Bell Laboratories, Murray Hill, New Jersey, pp. 78-81, 1983.

PUBLICATION

- Sharma Mohit and Saha Suman, “An Efficient Round Robin Algorithm in MapReduce Framework”, INROADS International Conference on Innovative Advancement in Engineering and Technology (IAET), vol. 3, pp. 106-10, 2014.

APPENDIX

Code: MST algorithm

```
import java.io.IOException;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Counters;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.util.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class Mst extends Configured implements Tool {
```

```

static enum MstCounters { totalWeight }

public static class MstMapper extends Mapper<Object, Text, IntWritable, Text> {
public void map(Object key, Text value, Context context)
    throws IOException, InterruptedException {

    String inputTokens[] = value.toString().split("\t");
    String weight = inputTokens[0] ;

try {
    int wt = Integer.parseInt(weight);
    IntWritable iwWeight = new IntWritable(wt);
    Text srcDestPair = new Text();
    srcDestPair.set(inputTokens[1] + ":" + inputTokens[2]);
    context.write(iwWeight,srcDestPair);
    }
catch((NumberFormatException ex) {
    }
}
}

```

```

static class MstReducer extends Reducer<IntWritable, Text, Text, Text> {
Map<String, Set<String>> node_AssociatedSet = new HashMap<String, Set<String>>();
public void reduce(IntWritable inputKey, Iterable<Text> values, Context context)

```

```

    throws IOException, InterruptedException {
    String strKey = new String();
    strKey += inputKey;
    Text outputKey = new Text(strKey);
    for (Text val : values) {

```

```

//boolean values to check if the two nodes belong to the same tree, useful for cycle detection

```

```

    boolean ignoreEdgeSameSet1 = false;
    boolean ignoreEdgeSameSet2 = false;
    boolean ignoreEdgeSameSet3 = false;

```

```

        Set<String> nodesSet = new HashSet<String>();
String[] srcDest = val.toString().split(":");
        //getting the two nodes of an edge
String src = srcDest[0];
String dest = srcDest[1];
//check if src and dest belong to the same tree/set, if so, ignore the edge
ignoreEdgeSameSet1 = isSameSet(src, dest);
//form the verticesSet
nodesSet.add(src);
nodesSet.add(dest);
ignoreEdgeSameSet2 = unionSet(nodesSet, src, dest);
ignoreEdgeSameSet3 = unionSet(nodesSet, dest, src);
if (!ignoreEdgeSameSet1 && !ignoreEdgeSameSet2 && !ignoreEdgeSameSet3)
{
        long weight = Long.parseLong(outputKey.toString());

        context.getCounter(MstCounters.totalWeight).increment(weight);
        context.write(outputKey, val);
                }
        }
}

private boolean unionSet(Set<String> nodesSet, String node1, String node2) {
        boolean ignoreEdge = false;

        /* boolean value to determine whether to ignore the edge or not. If the map does not
contain the key, add the key, value pair */
        if (!node_AssociatedSet.containsKey(node1)) {
                node_AssociatedSet.put(node1, nodesSet);
        }
else {
        // get the set associated with the key
Set<String> associatedSet = node_AssociatedSet.get(node1);

```

```

        Set<String> nodeSet = new HashSet<String>();
        nodeSet.addAll(associatedSet);
        Iterator<String> nodeItr = nodeSet.iterator();
        Iterator<String> duplicateCheckItr = nodeSet.iterator();
/* first check if the second node is contained in any of the sets from node1 to nodeN if so, ignore
the edge as the two nodes belong to the same set/tree */
        while(duplicateCheckItr.hasNext()){
            String node = duplicateCheckItr.next();
            if(node_AssociatedSet.get(node).contains(node2)){
                ignoreEdge = true;
            }
        }
/* if the associatedSet contains elements {node1 , node2, ..., nodeN}. Get the sets associated with
each of the element from node1 to nodeN */
        while (nodeItr.hasNext()) {
            String nextNode = nodeItr.next();
            if (!node_AssociatedSet.containsKey(nextNode)) {
                node_AssociatedSet.put(nextNode, nodesSet);
            }
/* add the src and dest to the set associated with each of the elements in the associatedSet, the src
and dest will get added to the set associated with node1 to nodeN */
            node_AssociatedSet.get(nextNode).addAll(nodesSet);
        }
        return ignoreEdge;
    }

    private boolean isSameSet(String src, String dest) {
        boolean ignoreEdge = false;
// boolean value to check whether the edge should be ignored iterating through the map
        for (Map.Entry<String, Set<String>> node_AssociatedSetValue :
node_AssociatedSet.entrySet()) {

```

```

        Set<String> nodesInSameSet = node_AssociatedSetValue .getValue();
//if the src and dest of an edge are in the same set, ignore the edge
        if (nodesInSameSet.contains(src)
            && nodesInSameSet.contains(dest)) {
            ignoreEdge= true;
        }
    }
    return ignoreEdge;
}
}

```

//the method to call the functions that run the jobs

```

public int run(String[] args) throws Exception {
    formMSTJob(args[0], args[1]);
    return 0;
}

```

//method to run the job that forms the MST

```

private void formMSTJob(String inputPath, String outputPath)
    throws Exception {
    Job mstJob = getMSTJobConf(); //get the job configurations
    FileInputFormat.setInputPaths(mstJob, new Path(inputPath));
    // setting the input files for the job
    FileOutputFormat.setOutputPath(mstJob, new Path(outputPath));
    // setting the output files for the job
    mstJob.waitForCompletion(true);
    Counters jobCnts = mstJob.getCounters();
    //get all the counters associated with mstJob
    long totalWeight = jobCnts.findCounter(MstCounters.totalWeight)
        .getValue();
    System.out.println("The total weight of the MST is " + totalWeight);
}
}

```

```

protected Job setupJob(String jobName, JobInfo jobInfo) throws Exception {
    Job job = new Job(new Configuration(), jobName);
    job.setJarByClass(jobInfo.getJarByClass());
    job.setMapperClass(jobInfo.getMapperClass());
    if (jobInfo.getCombinerClass() != null)
    job.setCombinerClass(jobInfo.getCombinerClass());
    job.setReducerClass(jobInfo.getReducerClass());
    job.setNumReduceTasks(3);
    job.setOutputKeyClass(jobInfo.getOutputKeyClass());
    job.setOutputValueClass(jobInfo.getOutputValueClass());
    return job;
}

```

//get the job configuration for formMST mapper and reducer

```

private Job getMSTJobConf() throws Exception {
    JobInfo jobInfo = new JobInfo() {
        @Override
        public Class<? extends Reducer> getCombinerClass() {
            return null;
        }
        @Override
        public Class<?> getJarByClass() {
            return Mst.class;
        }
        @Override
        public Class<? extends Mapper> getMapperClass() {
            return MstMapper.class;
        }
        @Override
        public Class<?> getOutputKeyClass() {

```

```

        return IntWritable.class;
    }
    @Override
    public Class<?> getOutputValueClass() {
        return Text.class;
    }
    @Override
    public Class<? extends Reducer> getReducerClass() {
        return MstReducer.class;
    }
};
return setupJob("formMST", jobInfo);
}

public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(), new Mst(), args);
    if (args.length != 2) {
        System.err.println("Usage: MST <in> <output > ");
        System.exit(2);
    }
    System.exit(res);
}

public abstract class JobInfo {
    public abstract Class<?> getJarByClass();
    public abstract Class<? extends Mapper> getMapperClass();
    public abstract Class<? extends Reducer> getCombinerClass();
    public abstract Class<? extends Reducer> getReducerClass();
    public abstract Class<?> getOutputKeyClass();
    public abstract Class<?> getOutputValueClass();
}
}

```