CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Data Locality in Distributed Computing Applications in the Cloud

A thesis submitted in partial fulfillment of the requirements

For the degree of Master of Science in Computer Science

By

Shubhada Nithyananda Nayak

May 2021

The thesis of Shubhada Nithyananda Nayak is approved:

Dr. Robert D McIlhenny	Date
Dr. John Noga	Date
Dr. Mahdi Ebrahimi, Chair	Date

California State University, Northridge

Acknowledgment

I would like to express my profound gratitude to my advisor and chair, Dr. Mahdi Ebrahimi, for his continued motivation and guidance. Thank you for your support, feedback, and valuable input throughout this thesis project.

I would also like to extend warm thanks to my thesis committee members Dr. Robert Mcllhenny and Dr. John Noga, for their support during this thesis project.

I am indebted to my family for having given me this opportunity to pursue higher education in the United States. I credit this thesis to my mother, Mrs. Nayana Nayak, for her unconditional love and encouragement. I am forever grateful to my sister and her family for placing their faith in my abilities and for their constant support in my difficult times. This would not have been possible without the support of my fiancé; thank you for empowering me to achieve the best.

Table of Contents

Signature ii

Acknowledgment iii

Abstract viii

1. Introduction 1

1.1. Hadoop 2

1.1.1. Brief History 2

1.1.2. Hadoop and Hadoop Ecosystem 3

1.1.3. Hadoop YARN Architecture 6

1.1.4. Application Workflow in Hadoop YARN 9

1.2. MapReduce 10

1.2.1. MapReduce Architecture 10

1.2.2. Word Count Example in MapReduce 12

2. Data Locality 15

2.1. Rack Awareness in Hadoop 15

2.2. Replica Placement using Rack Awareness 16

2.3. Categories of Data Locality 17

2.4. Data Locality in Hadoop 18

3. Related Work 19

3.1. Introduction 19

3.2. Replication 20
3.3. Data Distribution Based on Node Capabilities 21
3.4. Adaptive Data Placement 21
4. Data Set 23
5. Building the Distributed Computing Framework on Cloud 24
5.1. Laying the Bricks 24
5.2. Building the REST APIs 25
5.3. REST APIs 29
5.4. Leader Server 34
5.5. Worker Server 37
6. Experiments 39
7. Results 42
8. Conclusions and Future Work 48
References 49

List of Tables

Table 1:Key-Value Pairs from Map Phase 12

Table 2:Sort & Shuffle Phase 13

Table 3: Reduce Phase 13

Table 4:Text file size and number of words 23

Table 5:Compute DCR 35

Table 6: File Splits using DCR 36

Table 7:NDCR Compute Rules 37

Table 8:AWS cluster 39

List of Figures

Figure 1: HDFS Architecture 4

Figure 2: Hadoop 7

Figure 3: Hadoop YARN Architecture 8

Figure 4: Application Workflow in YARN 9

Figure 5: Map Reduce Architecture 11

Figure 6: Rack Awareness in Hadoop 16

Figure 7: Default Word Count 1.5GB file 42

Figure 8: DCR Word Count 1.5GB file 43

Figure 9: NDCR Word Count 1.5GB file 43

Figure 10: Default Word Count 32KB file 44

Figure 11: DCR Word Count 32KB file 44

Figure 12: NDCR Word Count 32KB file 45

Figure 13: Default Word Count 455MB text file 45

Figure 14: DCR Word Count 455MB file 46

Figure 15: NDCR Word Count 455MB file 46

Figure 16: Performance comparisons of Data Distribution Schemes 47

Abstract

Data Locality in Distributed Computing Applications in the Cloud

By

Shubhada Nithyananda Nayak

Master of Science in Computer Science

The data processing in data-intensive applications has become increasingly complex. Data placement in a distributed computing environment is critical as it is the primary factor for determining tasks' performance and scheduling. A job is divided into multiple smaller tasks in a distributed computing system and run in various nodes in a large-scale cluster. The data placement strategies from a homogeneous cluster can hinder the performance in a heterogenous cluster by increasing the overhead of data transfer of unprocessed data from slow nodes to fast nodes. In a distributed computing environment, the emphasis is on the programming model and the distributed file system. Data movement is expensive than compute movement; the goal is to move the task closer to the data. This thesis explores data locality based on the capabilities of computation nodes and how this can be dynamically improved based on the load status and studies behavior on pushing data locality to stages as far as reduce. The distributed computing framework is simulated on AWS cloud with Leader and worker nodes performing word count operations on files of varying sizes based on their data computation ratios computed off their computing capabilities calibrated using matrix multiplication and dynamically improving them ratios based on CPU and memory usage statistics. The results show enhanced computation times and thus a novel way to achieve data locality in heterogeneous computing systems.

1. Introduction

With the evolution of technology, distributed systems are becoming very ubiquitous. In its simplest definition, a distributed computer system is a group of computing machines working together to achieve a single task and appear as a single machine to the end-user. It is becoming very complicated in data to measure the total volume of data stored electronically. There are various streaming data sources such as social networking websites, e-commerce user traffic, data archive stores, etc. Most importantly, these digital streams are growing rapidly as data becomes a crucial tool to add value to the business. The trend is for this data footprint to grow as more machines in the computing space will generate even more essential data. System logs, sensor data, retail transactions, user behavior, contribute to the growing mountain of data. As the volume of data available to access for the public is increasing, the goal is not merely to manage private data but also to develop the tools and technology to extract and analyze value from publicly available data resources. No matter how great the computing algorithms are, they can be easily defeated by the volume of data it has to process.

The elephant in the room is although the storage capacities have spiraled, the disk access speed, which is the speed at which data can be read from the disk, has not increased. This means long wait times to read all the data from the drive and even slower writes to the disk; all this is when we consider a single disk for the operation. What if there were multiple disks? This would reduce the read-write times. If numerous disks were storing a portion of the data, working in parallel, we could read and write in under a few minutes.

When we think of parallel operations, of the many problems, the first to arise is fault tolerance. As many hardware pieces begin to operate together, the chances of one or a few failing steadily increase. A solution to avoid this is replication; redundant copies of data are maintained by the

system. In the event of a hardware failure, there is a copy readily available. The second problem demanding attention is how to combine the data; a data read from one disk may need to be combined with data from several hundred disks. This involves data transfer between the computing nodes, which may prove to be a bottleneck, thereby bringing down the performance and increasing the computing times. MapReduce provides a programming model that simplifies and abstracts the problem of read-write and reforms it into computation over a set of keys and values. There are two parts to the MapReduce paradigm: the Map phase and the Reduce phase, and the sort, shuffle, and spill occur here in between. Hadoop is a distributed computing framework which, unlike the traditional systems, enables multiple workloads to run on the same data parallelly at a massive scale on industry commodity hardware. Hadoop comes with reliable shared storage: HDFS (Hadoop Distributed File System) and data analysis with MapReduce programming model.

1.1. Hadoop

1.1.1. Brief History

The creator of Hadoop is Doug Cutting. Hadoop has its origins in Apache Nutch, an open-source web search engine and the Apache Lucene Project. Apache Nutch was started in 2002 as a crawler and a search system engine, but soon it was realized the architecture would not scale for billions of pages on the web. Based on the Google File System used in production at Google, a way was sought to solve storage needs for the huge files generated as a part of the web crawl and indexing process. Google introduced MapReduce to the world, Nutch had a working implementation of MapReduce and later replaced all major algorithms with running using MapReduce. Yahoo further developed the Nutch project into Hadoop, a system that ran a web scale. Since then, Hadoop has seen a rapid rate of adoption at an enterprise scale. The industry has recognized Hadoop’s pivotal

role as a distributed storage system and analysis platform for big data. Hadoop is the most widely used framework for batch processing jobs and smaller tasks where the time isn't the primary consideration.

1.1.2. Hadoop and Hadoop Ecosystem

Hadoop Ecosystem is a suite that provides various services for big data problems. It encompasses the Apache project and various commercial tools and solutions. The main elements of Hadoop are Hadoop Distributed File System, MapReduce, Yarn, and Hadoop Common. These tools work collectively to provide services such as analysis, storage, data maintenance, etc.

Following are the components that collectively form a Hadoop ecosystem:

HDFS: Hadoop Distributed File System

YARN: Yet Another Resource Negotiator

MapReduce: Programming paradigm for data processing

Spark: In-Memory data processing

PIG, HIVE: Query-based processing of data services

HBase: NoSQL database

Spark MLlib: Machine learning algorithm libraries

Zookeeper: Cluster manager

Oozie: job scheduler

All the components revolve around data; that is the essence of Hadoop, which makes it easier for data handling and analysis.

- HDFS: Hadoop Distributed File System is the major component of the Hadoop Ecosystem; it handles storage of large amounts of data that is structured and unstructured across a cluster of nodes and thereby maintains metadata in the form of log files.

  HDFS consists of two core components which are the Name Node and Data Node. Name Node is the primary node that contains metadata, file system logs which is data about data. It requires fewer resources than the data nodes, plays the role of a leader node. The Data Nodes are the industry commodity hardware. The HDFS is the heart of the system, plays the role of maintaining coordination between cluster and the hardware.

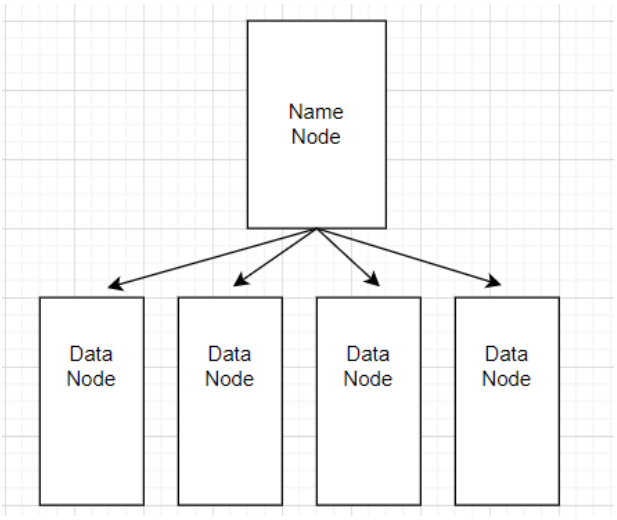

Figure 1: HDFS Architecture

- YARN: Yet Another Resource Negotiator manages the resources in the cluster. It handles scheduling and resource allocation for the Hadoop ecosystem. It mainly consists of Resource Manager, Nodes Manager, Application Manager. Resource allocation for the system's applications is taken care of by the resource manager; the allocation of node resources like CPU, memory is taken care of by the node manager. The Application Manager is the interface between the node manager and resource manager and handles the negotiations between them.

- MapReduce: By leveraging parallel and distributed algorithms, MapReduce carries the processing logic making it possible to write applications that can handle big data sets. The two most essential functions in this programming paradigm are Map() and Reduce(). Sorting, filtering, and organizing of the data are taken care of in the map phase. Map phase generates key-value pairs, which are processed in the reduce phase. Reduce as the name goes, aggregates the map data. It feeds on the output generated by the map phase and combines them to give the result.

- Pig: It is a query-based language that works on Pig Latin language. It provides a platform for structuring the data flow, processing, and analyzing big data sets. Pig abstracts the activities of MapReduce via commands and stores the results after processing in HDFS. It is a significant part of the Hadoop ecosystem and plays a pivotal role in optimizing and easing programming for Hadoop users.

- HIVE: Hive leverages SQL interface for reading and writing of big data sets. It is a query language called Hive Query Language. It enables real-time, and batches are processing while also supporting all the SQL datatypes making the processing quick. In short, it is a data warehouse software that facilitates the reading, writing, and management of large datasets residing in distributed storage systems.

- Apache Spark: It is a platform that handles all processing in-memory for an operation like real-time streaming and batch processing of enormous data sets. Spark is suited for real-time streaming applications, whereas Hadoop is suited for batch processing applications.

- Apache HBase: It is a NoSQL database that supports all kinds of data and is most suitable for unstructured data and capable of handling Hadoop and Spark Data. It provides the capabilities of Google Big Table; it is very tolerant when storing Big Data.

- Zookeeper: This tool resolves the management and synchronization of the resources in Hadoop, thereby reducing the incidences of inconsistency. Zookeeper overcomes these issues by handling synchronization, inter-component-based communication, grouping, and maintenance.

- Oozie: Handles task scheduling, managing the tasks, and binding them as a unit. There are two kinds of tasks or jobs, Oozie workflow and Oozie coordinator jobs. Oozie jobs mind how the tasks are executed; coordinator jobs are triggered because of an external stimulus.

1.1.3. Hadoop YARN Architecture

YARN stands for "Yet Another Resource Negotiator," it is a large-scale distributed operating system used for Big Data processing. YARN successfully separates the resource management layer from the data processing layer. YARN allows for different data processing engines like real-time-stream processing, interactive processing, graph processing, and batch processing to run and process data stored in a distributed file system like HDFS. It can dynamically allocate various resources and schedule the application processing. With big data, it is essential and necessary to manage the available resources so that every application can leverage them. YARN gained popularity due to its scalability, compatibility, optimal cluster utilization, and multi-tenancy.

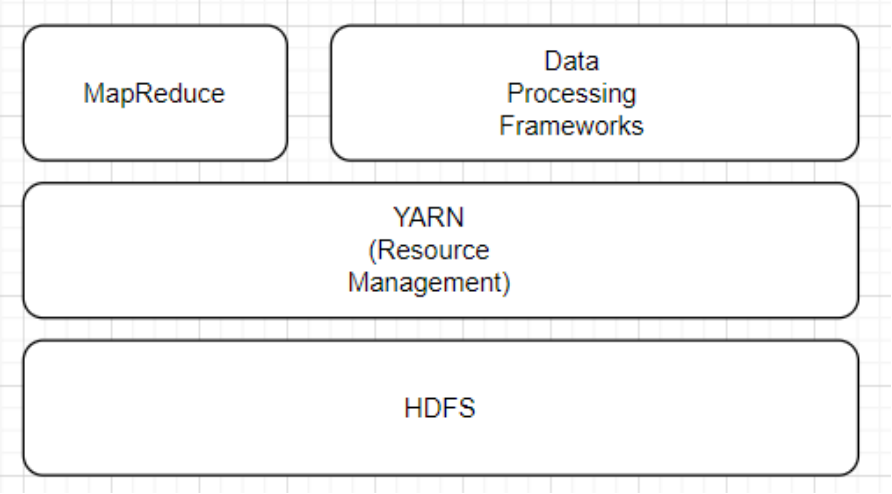

Figure 2: Hadoop

- Scalability: The YARN resource manager scheduler allows Hadoop to manage thousands of nodes in clusters efficiently.
- Compatibility: The built-in support for the MapReduce application makes it convenient to go with Hadoop.
- Cluster Utilization: The support for dynamic utilization in YARN makes way to optimized cluster utilization.
- Multi-tenancy: YARN support for multiple engine access makes it the best choice for organizations looking for multi-tenancy. The main components of the YARN architecture are client, resource manager, node manager.

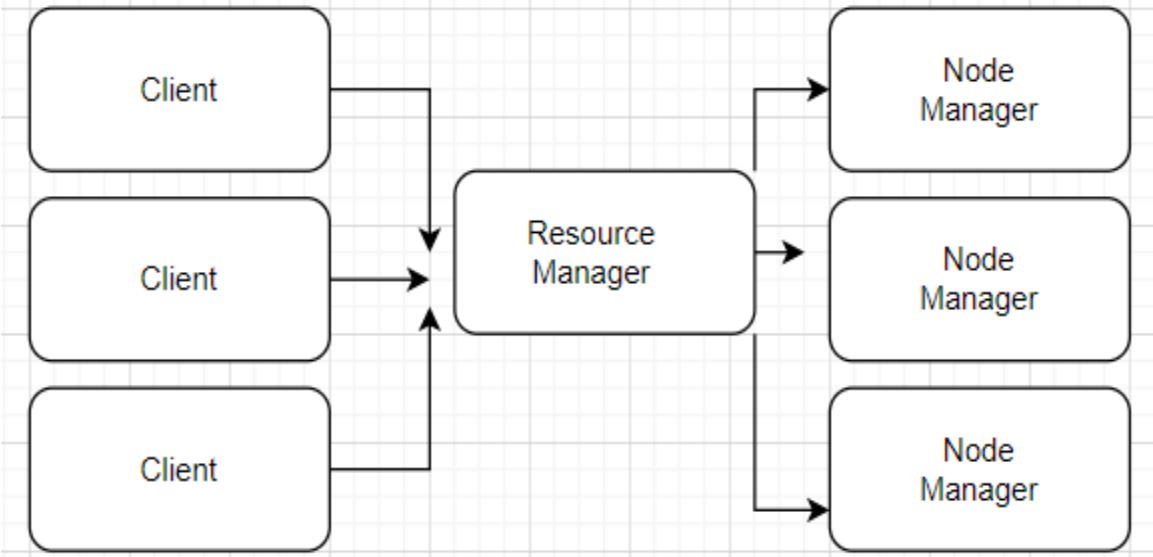

Figure 3: Hadoop YARN Architecture

The main components of YARN are:

- Client: Handles submission of map-reduce jobs.
- Resource Manager: The master daemon in YARN handles resource assignment and management for the applications. It comprises two major components, Scheduler, and Application Manager. The Scheduler is for scheduling based on application and available resources. It does not manage monitoring and tracking and does not account for task failure. The Application Manager is responsible for accepting the application and negotiating the container from the resource manager. In the event of a task failure, it restarts the application manager.
- Node Manager: Monitors the resource usage, performs log management and manages containers based on the resource manager's commands. It takes care of individual nodes on the Hadoop cluster and is responsible for creating the container process.
- Application Master: The application manager handles resource negotiation with the Resource Manager. The Application Master requests the resources from the resource manager while also

tracking the applications' status and progress. The Application Manager requests the container from the node manager by sending a container launch context. Once the application is started, it sends the health report to the resource manager quickly.

- Container: The collection of physical resources such as RAM, Disk, CPU on a node. The Container Launch Context invokes these containers, and it is a record with information on environment variables, security tokens, etc.

1.1.4. Application Workflow in Hadoop YARN

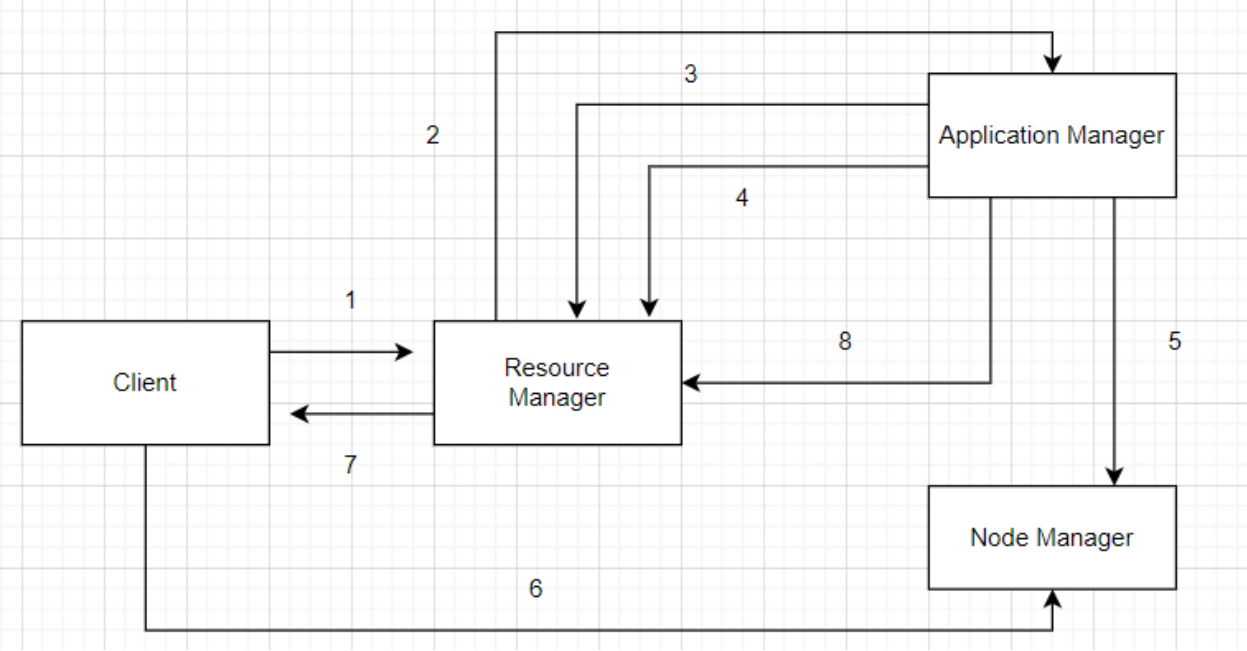

Figure 4: Application Workflow in YARN

Step 1: The client submits an application to the system.

Step 2: Resource Manager allocates container to start the Application Manager.

Step 3: Resource Manager requests registration from the Application Manager.

Step 4: The Application Manager requests containers from the Resource Manager.

Step 5: The Application Manager signals the Node Manager to launch containers.

Step 6: Application code is executed inside the container.

Step 7: The client submits a request to the Resource Manager/Application Manager to monitor application status.

Step 8: After the completion of the processing, the Application Manager unregisters with the Resource Manager.

1.2. MapReduce

The two primary components of HDFS are MapReduce and HDFS. MapReduce is a programming paradigm used for efficiently processing large data sets in a distributed fashion. The data is split and combined to produce the result. The libraries in MapReduce are written in multiple programming languages with optimizations. The goal of MapReduce in Hadoop is to Map each of the tasks, and then they are reduced to equivalent tasks for providing minimal overhead over the cluster network and reduced processing.

1.2.1. MapReduce Architecture

The MapReduce architecture's main components are Client, Job, MapReduce Master, Job-Pieces, Input, and Output. The MapReduce Client brings the job for processing to MapReduce, and multiple clients send jobs to MapReduce Manager for processing. The MapReduce job is the task the client wants to perform, for example, word count. This job comprises multiple tasks which are smaller and executed parallelly. The MapReduce Master handles the division of a job into job pieces. The result of all job-pieces combines is the result. Input data is fed to the MapReduce for processing, and output data results from the processing.

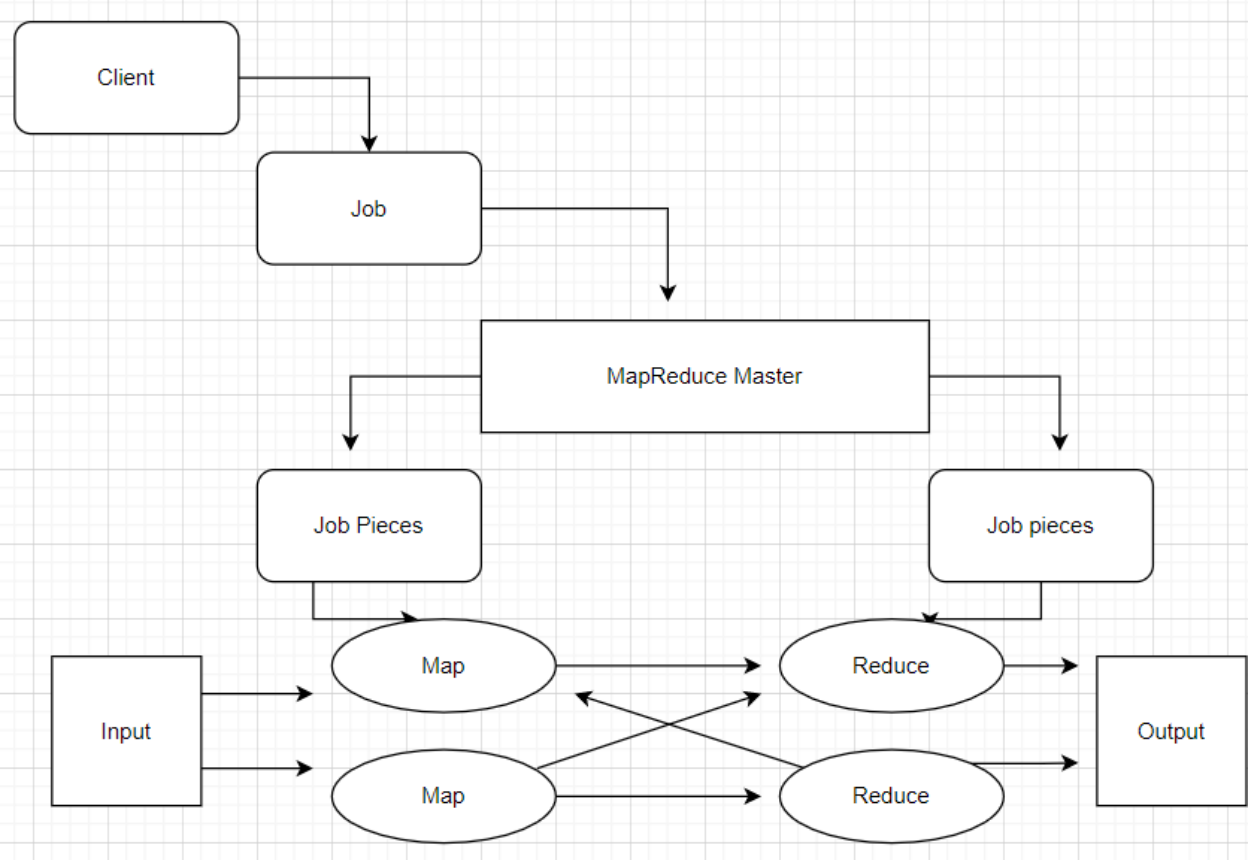

Figure 5:Map Reduce Architecture

In MapReduce, the client submits a job to the MapReduce Master; the master then divides the job into job pieces. These job-pieces are fed to Map and Reduce task, the code is Map and Reduce is user-specific according to the problem being solved. The input data supplied to the Map phase generates the key-value pairs as the output, this is then fed to the Reduce phase, and the output is stored in HDFS.

Map as the name goes maps the input data to key-value pairs which are sent to Reduce phase, and these key-value pairs are shuffled in Reduce phase sorted and sent to Reduce() algorithm. A Job Tracker's responsibility is to handle the management of resources and jobs across the node cluster. The Job Tracker also schedules each map on the Task Tracker running on the cluster's same data node. The Task Tracker is the actual worker node, performing processing on the Job Tracker's

instructions. Every worker has a Task Tracker deployed on it, and it is responsible for executing the Map and Reduce tasks as per the instructions by the Job Tracker.

1.2.2. Word Count Example in MapReduce

The Word Count operation occurs in 3 stages:

- Map Phase
- Shuffle Phase
- Reduce Phase

Map Phase

Considering a text file as input for the MapReduce operation to perform word count, the text is tokenized into words forming key-value pairs. The key here is the word from the text file, and the value is 1. For example, considering the string, "*this is my thesis draft, and the deadline is approaching,*" the map phase shall split this string into individual words, which in the case of this sentence is ten words with a value of 1 for each. Key-Value pairs from the map phase are:

(this, 1)
(is, 1)
(my, 1)
(thesis, 1)
(draft, 1)
(and, 1)
(the, 1)
(deadline, 1)
(is,1)
(approaching, 1)

Table 1:Key-Value Pairs from Map Phase

Shuffle Phase

After the completion of the Map phase, the Shuffle phase executes. Here the Key-Value pairs generated in the Map Phase are taken as input and sorted for ordering. Considering alphabetical order, the output of the Shuffle phase looks like this:

(and, 1) (approaching, 1) (deadline, 1) (draft, 1) (is, 1) (is, 1) (my, 1) (the, 1) (thesis,1) (this, 1)

Table 2:Sort & Shuffle Phase

Reduce Phase

In the reduce phase, all the keys are grouped, and the values for similar keys are added together; hence words with multiple occurrences come together. The reduce phase can be called an aggregation phase to transform key-value pairs into keys with values added. After the completed execution of the reduce phase, the keys appear to look as below.

(and, 1) (approaching, 1) (deadline, 1) (draft, 1) (is, 2) (my, 1) (the, 1) (thesis,1) (this, 1)

Table 3: Reduce Phase

The output is now the number of occurrences of each word in the file. The Word Count example executes the entire file and not just a single line or sentence as considered for this example.

2. Data Locality

Data Locality translates to moving the computation closer to data than the data to computation. Instead of moving large data sets to computation, data locality focuses on moving computation to nodes where the data resides. This reduces the congestion in the network and increases the overall performance of the system. In HDFS, data is divided into blocks and stored across the data nodes in the cluster. When a MapReduce job runs, the Name Node sends this MapReduce code to the data nodes where the data related to the job resides.

2.1. Rack Awareness in Hadoop

In HDFS, Rack is a collection of Data Nodes connected to the same network switch. A Hadoop cluster is deployed in several racks. To get the best performance and improve the network traffic during read-write, HDFS stores files across the nodes in a cluster. Name Node chooses Data Nodes on the current rack or other racks for data read and write.

Each rack consists of several Data Nodes, communication between Data Nodes on the same rack is much quicker and efficient compared to communication between Data Nodes on different racks. To reduce congestion in the network during read-write, Name Node chooses the closest Data Node for serving the client, and Name Node leverages the rack id information of each data Node for achieving this. The concept of selecting the closest Data Node using the rack id is called Rack Awareness in Hadoop.

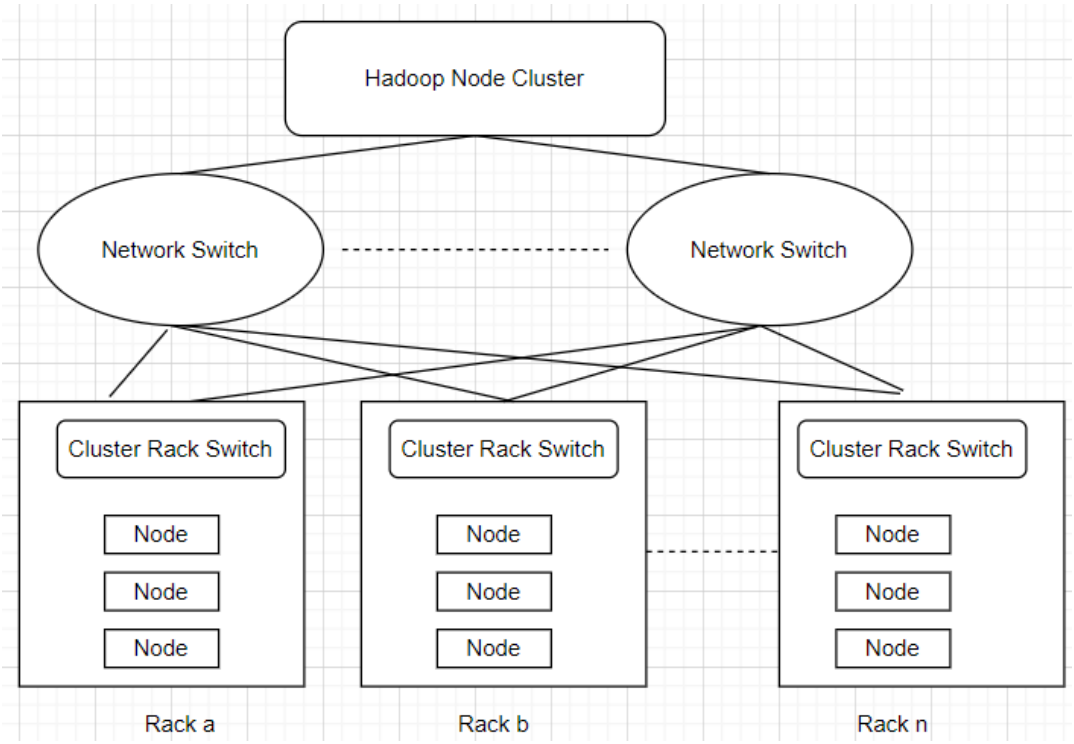

Figure 6: Rack Awareness in Hadoop

Rack Awareness helps address congestion in the network during file read and write, which improves the cluster performance. It helps achieve Fault Tolerance in the event of Data Node failure and in the event of rack failure. There is a reduction in latency and increased availability of data.

2.2. Replica Placement using Rack Awareness

HDFS stores replicas of data blocks for fault tolerance and high availability. If the replicas are all placed in a single rack, it leads to improved network bandwidth, but there is no copy of data to serve if the entire rack fails. If the replicas are stored on unique racks, then the writes' cost increases due to the transfer of blocks to multiple nodes on different racks. Thus, an intelligent way to place the replicas is called Rack Awareness Replica Placement.

By default, the replication factor in Hadoop is 3; this is, however, configurable. This means that Hadoop will, on default, create three replicas of every data node and store it in the cluster.

Using the Hadoop Rack Awareness Policy, which states that:

- In a single node, not more than one replica will be placed.
- In a single rack, not more than two replicas will be placed on different nodes.
- The number of racks on which the data blocks are replicated should be lesser than the number of replicas.

Hadoop placed the first replica on a data node on a local rack in the default replication, the second replica on a data node within the same rack, and the next replica on a different rack. Since the first two replicas are placed within the same rack this leads to improved performance as the inter rack communication is much faster due to higher bandwidth and lower latency.

Rack Awareness prevents data loss during failure and maximizes the read speed while also reducing the write cost and reducing the associated latency.

2.3. Categories of Data Locality

There are three main categories of data locality in Hadoop based on where the data is placed on the nodes in the cluster.

- Local Data Locality: When the mapper and the data it is to work on are on the same node in the cluster, local data locality is called. This is the best case as the data is very close to computation.
- Intra-Rack Data Locality: Due to several resource constraints, the Map task may not execute on the same node where the data is located. In this case, the map phase runs on a different data node but within the same rack.

- Inter-Rack Data Locality: This is the worst case and hence least preferred. The Map Phase runs on a data node in a different rack than the current rack where the data might be located.

Data Locality is hard to achieve when the size of the cluster increases and with heterogeneous nodes. As the number of the data nodes and the size of the data increases, fewer data will be local. In many large clusters, some nodes are slow while the others are fast, which creates an imbalance in the computation ratio data, and thus cluster is not homogeneous. Non-local data placement strains the network and affects the scalability making the network the bottleneck. Improved data locality brings faster execution and higher throughput; it reduces the congestion in the network and improves the Hadoop functionality and performance.

2.4. Data Locality in Hadoop

When a dataset is stored in HDFS, it is divided into blocks and stored across the cluster's data nodes. When a client requests a MapReduce task to be executed against the dataset, the individual mappers will process the data blocks or the input splits, as we can call them. When the mapper does not find the data it needs on the same node where the map task is executing, the data must be copied from the Data Node that contains it to the node where the map task is executing.

In a cluster with hundreds and thousands of data nodes, if each mapper tries to copy data from other nodes, this will be a network congestion bottleneck situation. So, in Hadoop, the aim is to move the computation to the data node rather than the data to the computation.

When a Job Tracker or the Application Master receives a job run request, it scans the cluster to find the nodes with the data resources to execute the Mappers and Reducers for the job. A decision is made on nodes where the mappers will be executed based on where each mapper's data is located. The aim is to improve the decisions to impact the data locality and drive the performance.

3. Related Work

3.1. Introduction

The Hadoop framework's development is aimed at effectively processing MapReduce applications. The application logic is supplied in the form of a Map and Reduce function using which the processing is performed. The HDFS is used to store the MapReduce application dataset on the data nodes in the Hadoop cluster in which the name node is the leader node for all the data nodes. The present data distribution methodologies are not suitable and efficient for a heterogeneous cluster. Data locality and its impact on performance are the major factors when scheduling tasks in the map phase. The task scheduling techniques in Hadoop should consider data locality in the cluster to improve the performance (Vrushali Ubarhande, 2015). Several methods for task scheduling have been experimented with and studied to understand data locality awareness during task scheduling.

Hadoop being the open-source implementation of the MapReduce model handles solutions to data-intensive applications by dividing larger tasks into smaller tasks. Each job is an operation on a partition of data in parallel (Chitharanjan, 2013). The available Hadoop data processing schemes are for homogeneous clusters (J. Xie, 2010) and do not perform the best in the setting of Hadoop as it is heterogeneous in the cloud (Kontagora, 2011).

Current data processing frameworks consider nodes in the cloud as homogeneous; in a heterogeneous cloud, the data may need to be transferred between low-speed nodes to nodes of higher speed for early task completion. As the data sets' size is large, data transfer between the nodes may consume very significant bandwidth, thereby decreasing the network performance. The goal is to improve the performance by effectively distributing by considering the different nodes' performance in the heterogeneous cluster (Vrushali Ubarhande, 2015).

Data Locality is defined as closeness between the data and the compute (Z. Guo, 2012), and seeks to move the execution of a task near or to a node where the data is residing for the job. In the older techniques for data distribution, data locality was achieved by reducing the number of fragments into which a data set was divided (F. Chung, 2006), this does not work in a heterogenous setting where the nodes, compute capacities, and data sizes vary from application to application in a cluster.

3.2. Replication

Another way to achieve data locality is to replicate; one way is to consider data placement on the same group of nodes (M. Y. Eltabakh, 2011), this allows for the control of where the data will be stored and solves the problem of a network bottleneck. Data relevance is an essential factor during data distribution, placement of relevant files on the same group of nodes if highly unlikely most of the times as the amount available on each node varies, co-location of data is a good idea only when there is ample amount of space on the nodes in the cluster. Because a file is divided into a fixed number of blocks that are already pre-determined, the co-location idea will not work when the data set is large (J. Jin, 2011).

Data may need to be transferred from a low-performing node to a high-performance node due to computational limitations in the heterogeneous cluster. As a result, a low-performing node can be the bottleneck. Because the network is an expensive resource compared to the node computational capabilities, the goal is to reduce the data movement for improved performance in distributed systems (J. Xie, 2010), also it is essential the measure the level of heterogeneity in the cluster to distribute the data between them effectively (J. Xie, 2010).

3.3. Data Distribution Based on Node Capabilities

Data Locality can be addressed by efficient data distribution to the nodes based on each node's computation ratio. The ratio can be calculated by running small jobs using the data distribution technique. However, these nodes on which the test runs may not have the same computing capabilities for the actual task to be executed using MapReduce. The approach to compute the heterogeneity by using test jobs may lead to congestion in the network (J. Xie, 2010). Increasing the replication factor can consume huge amounts of disk space and network bandwidth; hence the performance may decrease as the amount of replicated data increases making a careful judgment of policy of replication necessary (Z. Guo, 2012). Considering the high-performing nodes, it is possible that data were to be placed closer to them, thus redirecting all the tasks to high performance and achieving high data locality (A. Chervenak, 2007).

Insights can also be drawn using the nodes' logs, measuring the node capabilities using the executed node tasks. However, this may fail as the nodes' performance can vary due to multiple reasons, mainly the system factors. When new nodes are added to this heterogeneous environment, it causes issues as these new nodes have no historical logs.

3.4. Adaptive Data Placement

The load on every node in the cluster changes dynamically as the nodes' computational capabilities vary; the volume load of the jobs on each node can be used to adjust the load on every node used in the computation. Initially, to begin with, data is distributed to data nodes based on their computing rations in the Ratio Table; in the coming rounds of data distribution, the node load is taken into consideration to form the Load Distributions Pattern table, the load for each node is then computed based on the node load status (Avishan Sharafi, 2016).

The Load Distributions Pattern Table has the node load formulas for every node load state. The Ratio Table stores every node's computational capacity ratios in different job types, while the Load Distributions Patterns Table stores load parameters like CPU utilization and memory utilization of the cluster in other node load states. The node load states are compared with the cluster states to define the exact load assigned to the nodes. There are different node load states in a cluster: Underload, Normal load, Overload (Avishan Sharafi, 2016). The percentage of added workload for each node is computed based on where their current workloads fall, and the formula is determined from the Load Distributions Pattern Table.

In a heterogeneous cluster, Adaptive Data Placement (Avishan Sharafi, 2016) distributes the data fragments based on other nodes' computing capabilities, followed by calculating the appropriate workload based on every node's load parameters which improve data locality and increases the performance.

4. Data Set

In this thesis project, we are experimenting with data locality using word count operation by using text files of varying sizes. The main text of the file is extracted from the description of the popular Netflix show ‘Bridgeton’; text is repeated multiple times within each file to generates files of varying sizes.

For the experiment, we are using three files of sizes 32KB, 455MB, and 1.5GB; the number of words in each file is listed in the table below.

32KB	4924
455MB	71356877
1.5GB	233832377

Table 4: Text file size and number of words

We compute the word count of each of the files above by also varying the number of nodes in the cluster to understand the behavior and the time is taken, number of words in every file. The goal is to consider files of sizes that vary significantly with arbitrarily changing numbers of words; hence we have chosen to keep 32KB, 455MB, and 1.5GB as the file sizes.

5. Building the Distributed Computing Framework on Cloud

5.1. Laying the Bricks

In the experiment, we are building a simulation of the likes of distributed computing frameworks like Hadoop. We consider a cluster of 1 Leader node and 6 Worker nodes on the AWS cloud computing platform. The goal is to create a heterogeneous cluster on AWS with nodes of varying compute and memory capacities; hence, we use worker nodes of different tiers in AWS and consider a leader node of high performance.

The goal is to create separate Leader and Worker servers using Java 8, Spring Boot, and Tomcat on the nodes in the cluster; REST APIs are exposed on both the Leader and the worker servers for enabling communication among workers and the leaders as well as between the worker nodes. The REST APIs handle all the Create, Update, and Delete instructions between the Leader and Worker nodes. All the REST APIs shall be invoked using the Postman Client on the host machine that will be used to run the simulation experiments.

For the experiments, we are using manually created text files of varying sizes of 32KB, 455MB, 1.5GB create by using the description of a popular Netflix show ‘Bridgerton’, the simulation will run Word Count operation under different settings of the cluster and record the resource utilization and the computation times to under the efficiency of the data locality and thus the performance in the cluster.

The experiments are simulated in AWS via a windows host machine using Eclipse for the Leader and Worker server development. The code on AWS nodes is containerized using docker to facilitate ease of maintenance and dependency management. The server war files are deployed within the container to start the server on the nodes. The leader node shall possess all the text files which will be used for word count operations, and the file fragments are shared with the worker

nodes during the task execution phase. The Leader and the Worker nodes have a fixed disk storage capacity, and for the simulation, when this capacity is reached, the files shall be manually deleted to release the disk space for further simulation to be run. For the experiment's goal, we are not considering the fault tolerance of the cluster. Hence, replication is not implemented and will be considered for future work, as discussed in the later sections.

The experiment assumes that neither the Leader nor the worker fails, and in the event of a failure, the user shall manually restart the node on the cloud. The computing capabilities of the nodes will be calibrated using a parallel matrix multiplication program that will run on all the worker nodes, and the data returned to the Leader will be used to build the data computation ratio tables; this table will be used to distribute the data blocks to the worker nodes. The worker nodes will run a Cron job that updates the master frequently of the CPU and memory usage statistics to the leader node. This will be leveraged to improve the data computation ratios dynamically and hence the way data blocks shall be distributed for future tasks.

The workers expose REST APIs for communication with the Leader as well as with the other workers. The Leader communicates with the workers using these REST endpoints and waits to hear the worker nodes' response via the REST APIs exposed on the Leader. All the REST calls and operations happen in parallel in the cluster, keeping Leader to worker communication and worker-to-worker communication independent of the others in the cluster.

5.2. Building the REST APIs

Both the worker and the leader nodes have REST APIs exposed for communication between each other. The endpoint exposed on leader nodes is mostly for coordination between the Leader and the worker to track and complete the tasks. The leader also exposes some endpoints for

configurations by the users on the leader node. The worker node endpoints are for communication between the worker and the leader nodes and communication between the worker nodes.

The experiments are aimed to run three types of word count operations: DCR, Default, and NDCR.

- DCR: Data Computation Ratio Word Count, the operation is based on the distribution of the data blocks or fragments to the worker nodes based on their Data Computation Ratio calculated using the time taken to perform the parallel matrix multiplication program. For calibration, a 3000 X 3000 matrix is considered for parallel multiplication. The time taken on each worker node is returned to the Leader, which computes the Data Computation ratio using logic that will be discussed further. The mapper and the reducer tasks are optimized to consider this novel way of distribution.
- NDCR: Node Load Data Computation Ratio Word Count, the operation is based on the distribution of data blocks or fragments to the worker nodes based on their Data Computation Ratio; however, this ratio is dynamically updated every five minutes based on the node load statistics update by every worker node in consideration for the current task running. The DCR (Data Computation Ratio) is updated based on the range where the worker nodes' combined node load statistics fall, as discussed further. The mapper and the reducer tasks are optimized to consider this novel way of data distribution.
- Default: Default Word Count distributes the data fragments or data blocks under the assumption that the cluster is homogeneous, all the worker nodes are of the same computing capabilities, the reducer for the reduce operation, in this case, is chosen based on the First Come First Serve rule, the node that completes the map phase fastest becomes the reducer.

There are two servers developed, the leader server and the worker server. The leader server follows the following order of events initiated by the user on the host machine in DCR, NDCR, and Default.

1. The host creates the cluster on the AWS, all the nodes in the cluster are activated manually and set running.
2. The DNS and IP Address of all the worker instances is registered with the leader instance in the leader DNS table, and the host manually makes this API call on Leader using postman.
3. The host makes the API call on Leader to set the Master IP on the leader node. This is the leader node's IP, which will be conveyed to all the worker nodes in the subsequent calls.
4. The Computer DCR API is invoked on the leader node. This API calls Compute DCR Worker API on worker nodes that perform the matrix multiplication on workers and return the Leader's time.
5. After the worker performs the matrix multiplication, they call the callback endpoint on the leader node to update the times taken on different nodes for matrix multiplication.
6. The leader node computes the DCR using all the times returned, and this is a scheduled job running on the Leader which waits until all the nodes return, or it runs out of several retries.
7. Now the Leader and the workers are ready to take a word computation task. The leader node calls word compute-optimized API to perform the DCR Word Count Operation. This API works on fragmenting the text file and SCP it to the worker nodes, and call word count DCR worker endpoint on the worker nodes. The worker nodes will now perform parallelized word count and return the result to the reducer. The Leader elects the reducer before distributing the file fragments to the worker nodes, where the node with the most fragments becomes the reducer. The reducer returns the Leader's response, including the time taken, number of words,

and a map of each word with its frequency in the file submitted for the word count operation.

8. The worker nodes constantly update the Leader in regular intervals of their CPU and memory statistics called node load stats. The Leader uses this to dynamically improve the DCR ratio for every worker node for future tasks and data block distribution.
9. The Leader makes the word count optimized API call; with the dynamically improved DCR, it is classified as NDCR Word Count Operation. The workers received data blocks as per their improved data compute ratios, and the reducer returns the response to the leader node. This response contains the time is taken, the number of words, and a map with the frequency of every word.
10. The Leader can make a Default Word Count Operation call anytime. This considers the entire cluster as homogeneous, and every worker node receives the same number of data file fragments. The reducer here is chosen based on the FCFS rule, the worker node that completes the map phase the fastest becomes the reducer. The leader node picks the reducer and informs the worker nodes about the reducer node. The worker nodes send the node results to the reducer, which aggregates the results and informs the task's Leader. This response contains the time is taken, the number of words, and a map with the frequency of every word.

5.3. REST APIs

APIs to run the DCR/NDCR/Default Word Count Operation:

1. Leader Welcome API: http://master_ip:8080/hadoopsimulationmasterpoc

   This API is to check if the leader server is up and running in the Docker container.

   HTTP Method Type: Get

   Success Status Code: 200

   Response: “Hello World!”

2. Worker Welcome API: http://:worker_ip:8080/hadoopsimulationpoc

   This API is to check if the worker node is up and running in the Docker container.

   HTTP Method Type: Get

   Success Status Code: 200

   Response: “Hello World!”

3. Leader Build DNS: http://master_ip:8080/hadoopsimulationmasterpoc/master/build/dns

   This API is to build the leader DNS table of the workers in the cluster.

   HTTP Method Type: Post

   Success Status Code: 200

   Request: JSON map of all the worker’s DNS and IP.

   Response: JSON map of all the worker’s DNS and IP.

4. Leader Set Master IP:

   http://master_ip:8080/hadoopsimulationmasterpoc/master/add/master/config

   HTTP Method Type: Post

   Success Status Code: 200

   Request: master IP to be updated.

   Response: master IP that was updated.

5. Leader DCT Table Status:

   http://master_ip:8080/hadoopsimulationmasterpoc/master/get/dctTable/status

   This API is to check the DCT status; DCT Table is updated by the workers with the time taken to compute the matrix multiplication parallelly.

   HTTP Method Type: Get

   Success Status Code: 200

   Response: JSON map of all worker's matrix multiplication compute times.

6. Leader DCR Ratio Table Status:

   http://master_ip:8080/hadoopsimulationmasterpoc/master/get/dcrRatio/table

   This API returns the DCR computed by the Leader for the workers based on their compute times for the matric multiplication.

   HTTP Method Type: Get

   Success Status Code: 200

   Response: JSON map of the compute ratios of the workers along with their DNS. This ratio must be ready and available to perform the DCR Word Count or the NDCR Word Count.

7. Leader DCR/NDCR Word Count:

   http://master_ip:8080/hadoopsimulationmasterpoc/master/call/count/Children

   This API is invoked by the Leader on the worker cluster to begin a DCR or NDCR Word Count operation.

   HTTP Method Type: Post

   Success Status Code: 200

   Request: task name and file name. The task name is the unique name given to every word count operation performed, and the file name is the text file used for the operation.

   Response: time taken, map of words in file with their frequency, and the total words in a file.

8. Leader Default Word Count Operation:

   http://master_ip:8080/hadoopsimulationmasterpoc/default/master/call/wordcount/children/default

   This API is invoked by the Leader on the worker cluster to begin the default Word Count operation.

   HTTP Method Type: Post

   Success Status Code: 200

   Request: task name and file name. The task name is the unique name given to every word count operation performed, and the file name is the text file used for the operation.

   Response: time taken, map of words in file with their frequency, and the total words in a file.

Some of the many APIs for maintenance:

1. Set Reducer Configs:

   http://worker_ip:8080/hadoopsimulationpoc/slave/set/reducer/configs

   This API is to set the reducer config manually on the reducer node. Reducer configs will initialize a few things needed for the reducer to begin; this API lets this be done manually for testing.

   HTTP Method Type: Post

   Success Status Code: 200

   Request: task name and DNS of the reducer node.

   Response: confirmation message that reducer is set to begin.

2. Find Reducer for Default Word Count:

   http://master_ip:8080/hadoopsimulationmasterpoc/default/master/word/count/reducer/who

   This API is used to manually determine which worker node has been chosen as the leader's reducer under the FCFS scheme.

   HTTP Method Type: Get

   Success Status Code: 200

   Response: DNS of the reducer node

3. Set Master IP in Worker Manually:

   http://worker_ip:8080/hadoopsimulationpoc/slave/set/master/ip/manually

   This API is to update the master IP address in the worker nodes manually.

   HTTP Method Type: Post

Success Status Code: 200

Request: Ip address of the master and the DNS of the worker node where it must be updated.

Response: confirmation on update.

4. Get Worker Node Load Statistics:

   http://worker_ip:8080/hadoopsimulationmasterpoc/master/get/node/load/status

   This API is to manually query the CPU and memory usage of the worker nodes on the cluster.

   HTTP Method Type: Get

   Success Status Code: 200

   Response: CPU and memory usage statistics.

5. Get Leader Blocks Allocation Map:

   http://master_ip:8080/hadoopsimulationmasterpoc/master/blocks/allocated

   This API aims to look at the block allocations for the worker nodes in the cluster for the current task.

   HTTP Method Type: Get

   Success Status Code: 200

   Response: map of block allocation, DNS, and a number of blocks distributed against each DNS.

5.4. Leader Server

The Leader node is the head of the cluster and coordinates actions with all the cluster nodes. The worker nodes coordinate action with each other via the leader node. The leader node handles all the heavy lifting tasks for the word count operation, and this includes computation of data computation ratio, fragment the files and SCP over the network to the worker nodes, set reducer configs on the reducer node, elect the reducer node, gather the response from the reducer for the task completion.

The data computation ratio is the backbone of the DCR and NDCR word count operation. The nodes in the cluster return the matrix multiplication times to the leader node; the leader node utilizes the compute capabilities based on these times to compute the ratio according to which the blocks will be distributed. Based on the highest average response time taken, the ratio is computed, which will be used to compute each worker's number of blocks.

When the worker nodes return the time taken for matrix multiplication, this is updated in the leader node's DCRT HashMap. This is a map with a key as the DNS of the worker node, and the value is the time taken for the matrix multiplication. A task is scheduled on the Leader to run every 5 mins to check if the all the worker have updated the DCRT table or the number of retries have exhausted, one of these cases the DCR (Data Computation Ratio) gets computed.

For the computation of DCR, all the worker nodes are arranged in a Max Heap to get the node with the highest time taken for a response. The ratio for all the other nodes is computed as a ratio of this Highest Response Time.

This table gives a peek into the code that goes into computing the DCR for the worker nodes. The ratio is based off the Highest Response Time returned by one of the worker nodes.

```
public DCR_RatioTableObject computeDCRRatio() {
		// TODO Auto-generated method stub
		PriorityQueue<Pair> pq = new PriorityQueue<Pair>((a, b) -> ((int)
(b.time - a.time))); // max heap based on times
		HashMap<String, Long> times = HadoopMaster.config.getDCR();
		HashMap<String, Double> dcRatio = HadoopMaster.config.getDCRatio();

		for (String dns : times.keySet()) {
			Pair p = new Pair(dns, times.get(dns));
			pq.add(p);
		}
		Pair highest = null;
		if (pq.size() > 0) {
			highest = pq.peek();
		}
		// if there is just one node then the node should get all
		if (pq.size() == 1) {
			Pair temp = pq.poll();
			dcRatio.put(temp.dns, 1.0);
		} else {
			while (!pq.isEmpty()) {
				Pair temp = pq.poll();
				double val = (float) highest.time / temp.time;
				long dcr = highest.time / temp.time;
				dcRatio.put(temp.dns, val);
			}
		}
		DCR_RatioTableObject d = new DCR_RatioTableObject(dcRatio);
		d.setStatus(
				"If you see less than no of worker assigned, probably
you have queried too early or the nodes are dead. Try in 15 mins!");
		return d;

	}
```

Table 5: Compute DCR

Once this ratio is available, the value sorts the ratios in decreasing order to SCP the file blocks or fragments to the workers. The sum of the ratios is computed and used to determine the average with several file splits that will be considered; in the case of experiments, we consider every block size 64KB. Hence the number of blocks to the slowest node is computed as the quotient of the number of blocks divided by the sum of all the DCRs.

The number of blocks to any worker now is the product of its DCR value with the number of blocks to the slowest performing node.

Number of blocks to the slowest node = (Number of blocks file is split into) / (sum of all DCRs) Number of blocks to any node = (Number of blocks to the slowest node) * (DCR of the node)

Table 6: File Splits using DCR

For the DCR Word Count, the Leader chooses the nodes which receive the highest number of blocks based on the above distribution scheme. This is communicated beforehand to all the worker nodes so they are aware of where they must send their intermediate results; the Leader also sets the reducer configs on the node that is elected as the reducer.

In the case of NDCR, the DCRT table ratios change dynamically based on the node load updates sent by the worker nodes to the Leader. The worker nodes send their CPU and memory usage statistics to the Leader. The average of the CPU and the memory usages is computed. Two standard deviations of the CPU usages and memory usages is also calculated. Suppose the average CPU usage and memory usage varies from the previously recorded average. In that case, the ranges in which the worker node fall usage is analyzed to computed the reduced or increased load for the next job that will be assigned to the cluster.

There are three load regions where the current statistics of the node's load can fall. The node can be in a regular load state, under load state, and overload state. If the node is overloaded, then the DCR is improved just 10 percent to reduce the load on the worker; if the node is in under load state, then the node load is increased by 33 percent. If the node is in a normal load, its DCR is kept as is.

cpuUsage <= oneStdBelowMeanCPU and memUsage <=oneStdBelowMeanMem	Underload: 0.33
cpuUsage > oneStdBelowMeanCPU and memUsage > oneStdBelowMeanMem	Overload: 0.10
Anything within one STD is a normal load	Normal load

Table 7: NDCR Compute Rules

In Default Word Count Operation, all the worker nodes are assumed to be of the same compute capacities; hence the file fragments are divided equally. Also, the reducer is chosen as the worker who computes the word count first and returns to the Leader. All the other nodes are then informed about the reducer DNS and to send their intermediate results.

5.5. Worker Server

The worker server performs the word count operation for DCR/NDCR/Default modes. The worker performs parallelized word count on the data fragments distributed to the node and returns the intermediate result to the reducer node. All the worker nodes will be communicated to by the leader node about the reducer node.

Each worker node runs Cron jobs that update the leader node with the CPU and memory usage statistics. This Cron job starts running as soon as the worker node comes to life and frequently updates the Leader of the usage statistics until the host turns off the node. The worker nodes do not communicate with each other; all the communications go through the leader node. The host manually manages the file storage of all the worker nodes in entire memory issues.

For the word count operations, the reducer aggregates the results returned by the sibling nodes and returns the total time taken to perform the word count operation, the hash map for words along with their frequency of occurrence, and the total number of words in the file. Every operation is uniquely classified using the task name provided by the Leader by the host.

For the matrix multiplication operation, the worker nodes perform a parallelized matrix multiplication on a randomly generated 3000 X 3000 matrix and return the time taken for this multiplication operation to the leader node, which will be used for DCR ratio computations.

6. Experiments

In this thesis project, we perform nine different kinds of experiments to reinforce the findings and the relationship with the underlying data distribution schemes. We perform Data Computation Ratio Word Count Operation, Node Load Data Computation Word Count Operation, and Default Word Count Operation. For each of the three types of Word Count operations, we vary the number of worker nodes in the cluster and the text file size used for word count.

For the sake of experiments, we have considered clusters with two worker nodes, four worker nodes, and six worker nodes also test file of sizes 32KB, 455MB, and 1.5GB. The cluster is built on AWS and comprises heterogeneous nodes to simulate a heterogeneous cloud environment for the distributed computing frameworks. The instances chosen for worker nodes are AWS EC2 with varying CPU and memory capacities; the leader node is also an AWS EC2 instance.

Node Type	AWS EC2 Type
Leader Node	t3.large
Worker 1	t3.medium
Worker 2	t3.large
Worker 3	t2.medium
Worker 4	t2.xlarge
Worker 5	t2.large
Worker 6	t3a.medium

Table 8: AWS cluster

The cluster nodes come with varying compute capacities, and hence their performance varies and facilitates the simulation of a heterogenous cluster on the cloud.

The different nodes on the AWS cluster:

- T3.large: 2vCPUs and 8GB RAM, 8GB disk storage.
- T3.medium: 2vCPUs and 4GB RAM, 8GB disk storage.
- T2.medium: 2vCPUs and 4GB RAM, 8GB disk storage.
- T2.xlarge: 4vCPUs and 16GB RAM, 8GB disk storage.
- T2.large: 2vCPUs and 8GB RAM, 8GB disk storage.
- T3a.medium: 2vCPUs and 4GB RAM, 8GB disk storage.

The different experiments performed are:

- 2 Worker Experiment:

The cluster is comprised of the leader node and worker 1, worker 2. We perform DCR, NDCR, and Default word count operations on this cluster using files of sizes 32KB, 455MB, and 1.5GB and record the times taken in each case.

- 4 Worker Experiment:

The cluster is comprised of the leader node and worker 1, worker 2, worker 3, worker 4. We perform DCR, NDCR, and Default word count operations on this cluster using files of sizes 32KB, 455MB, and 1.5GB and record the times taken in each case.

- 6 Worker Experiment:

The cluster is comprised of the leader node and worker 1, worker 2, worker 3, worker 4, worker 5, worker 6. We perform DCR, NDCR, and Default word count operations on this cluster using files of sizes 32KB, 455MB, and 1.5GB and record the times taken in each case.

We also use a file whose input is sorted and feed it to word count, and this is understanding if a sorted file introduces better performance in word count operations also if better ways of sorting are needed for this case. The time taken in the nine experiments performed is recorded to plot graphs and understand how the data distribution schemes affect the word count performance in a distributed computing platform. The results are discussed in the next chapter.

7. Results

The experiments performed on the heterogenous AWS cluster have varying numbers of workers and file sizes used for the word count operations. Three types of data distribution schemes are used for the word count operation: DCR, NDCR, Default. Following are the graphical representations of the performance in each of the data distribution schemes given the number of workers in the cluster and the file size.

- 1.5 GB text file and 2,4,6 workers

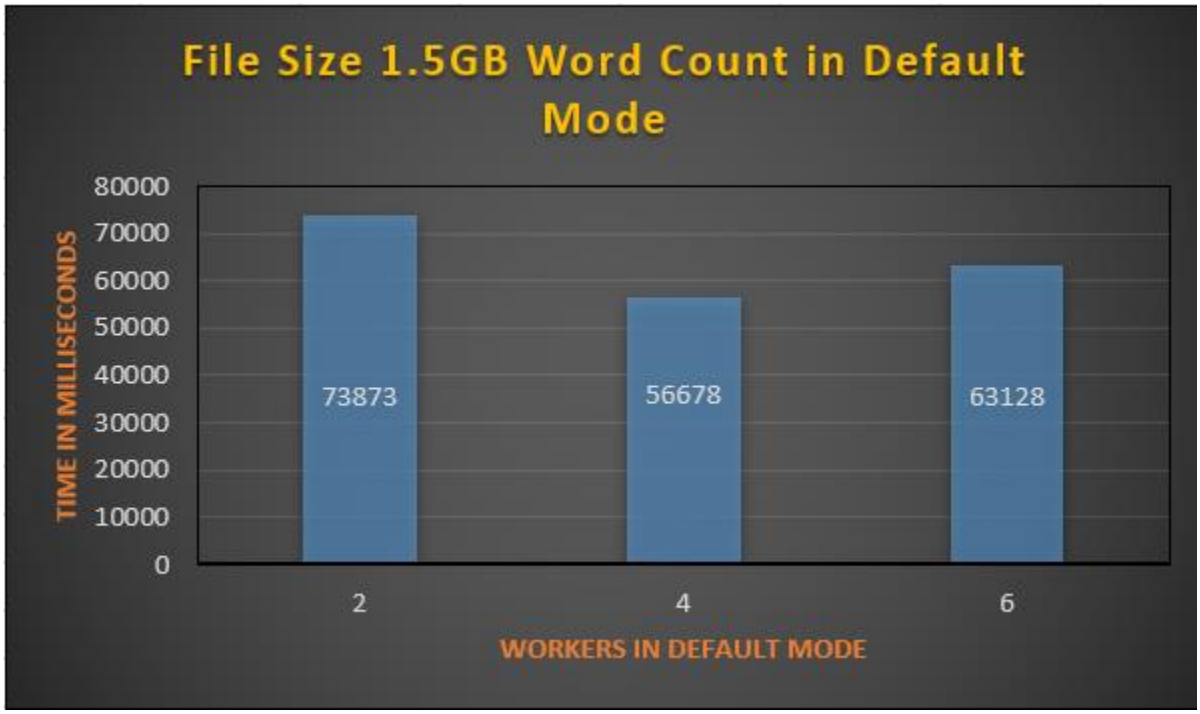

Figure 7: Default Word Count 1.5GB file

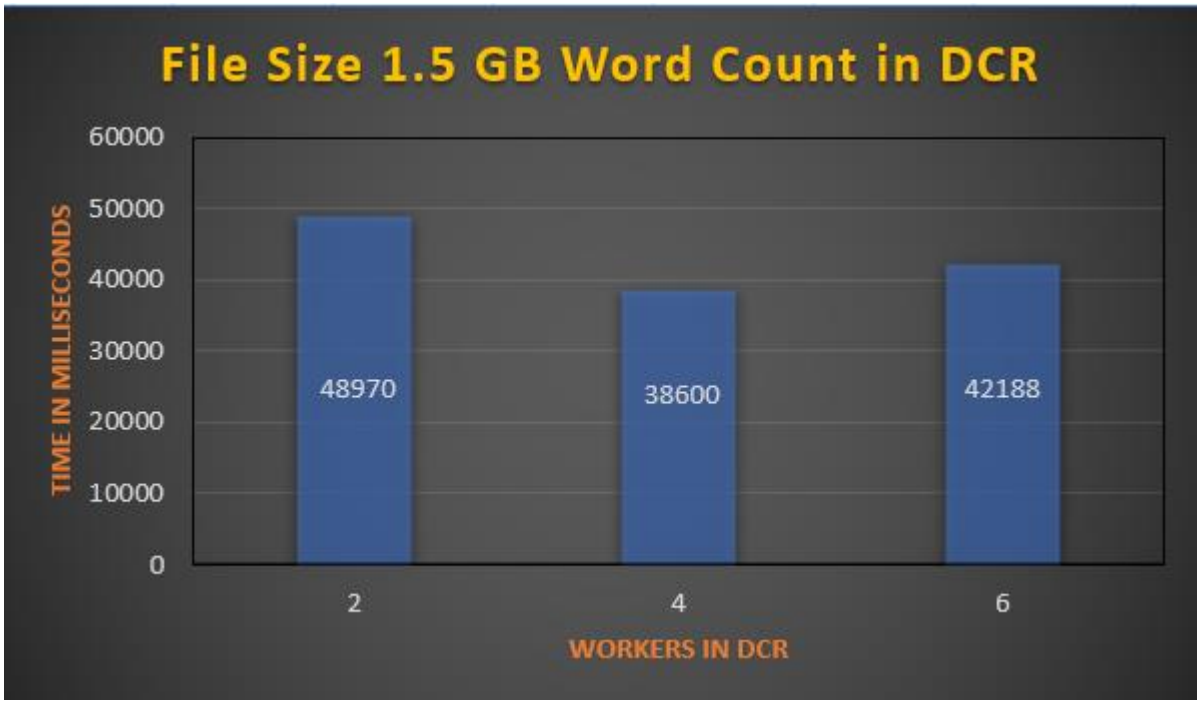

Figure 8: DCR Word Count 1.5GB file

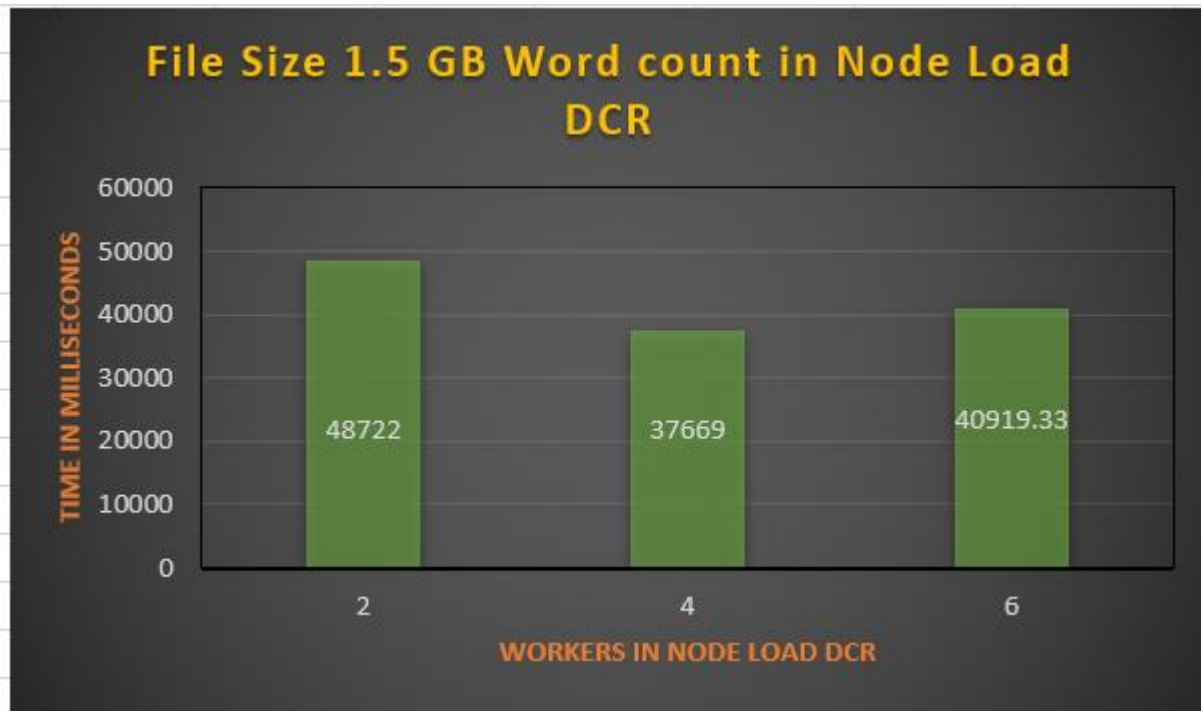

Figure 9:NDCR Word Count 1.5GB file

- 32KB text file and 2, 4, 6 workers.

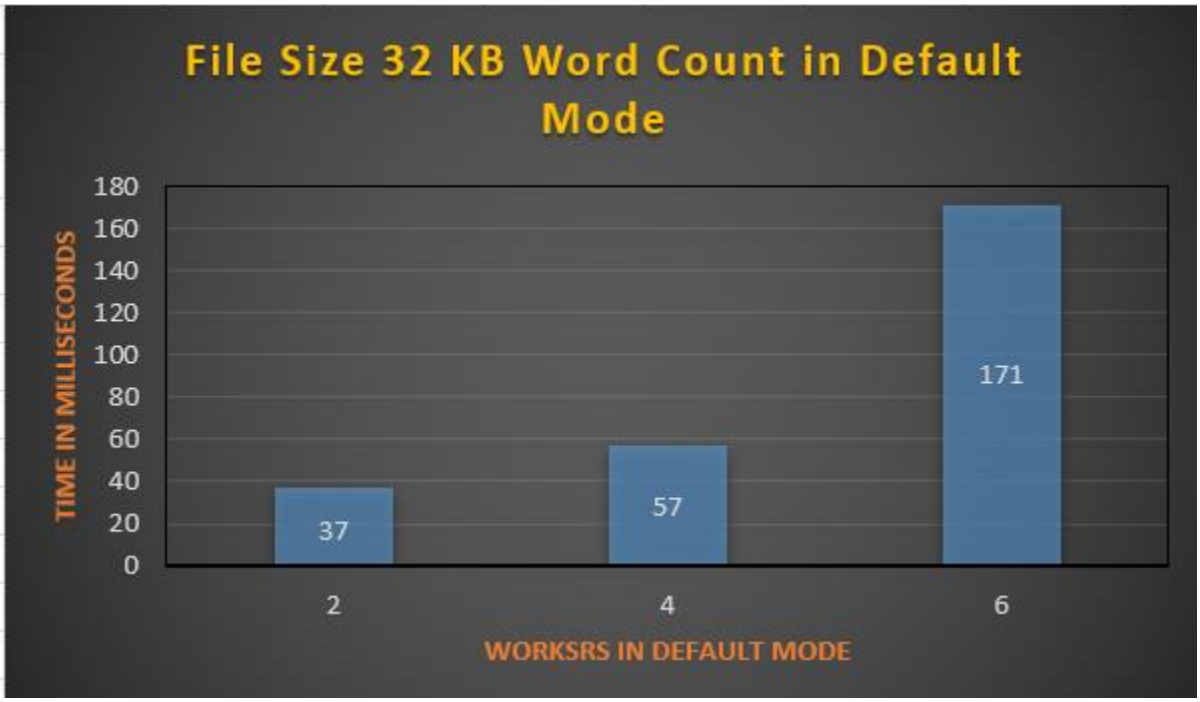

Figure 10: Default Word Count 32KB file

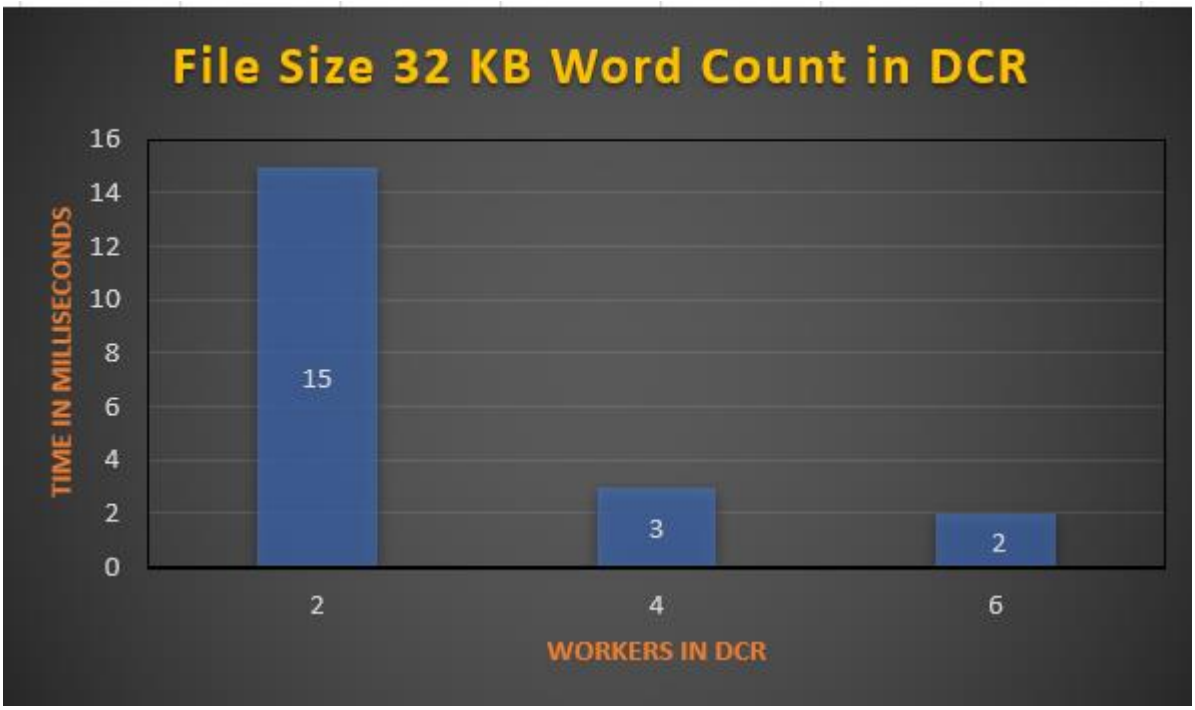

Figure 11: DCR Word Count 32KB file

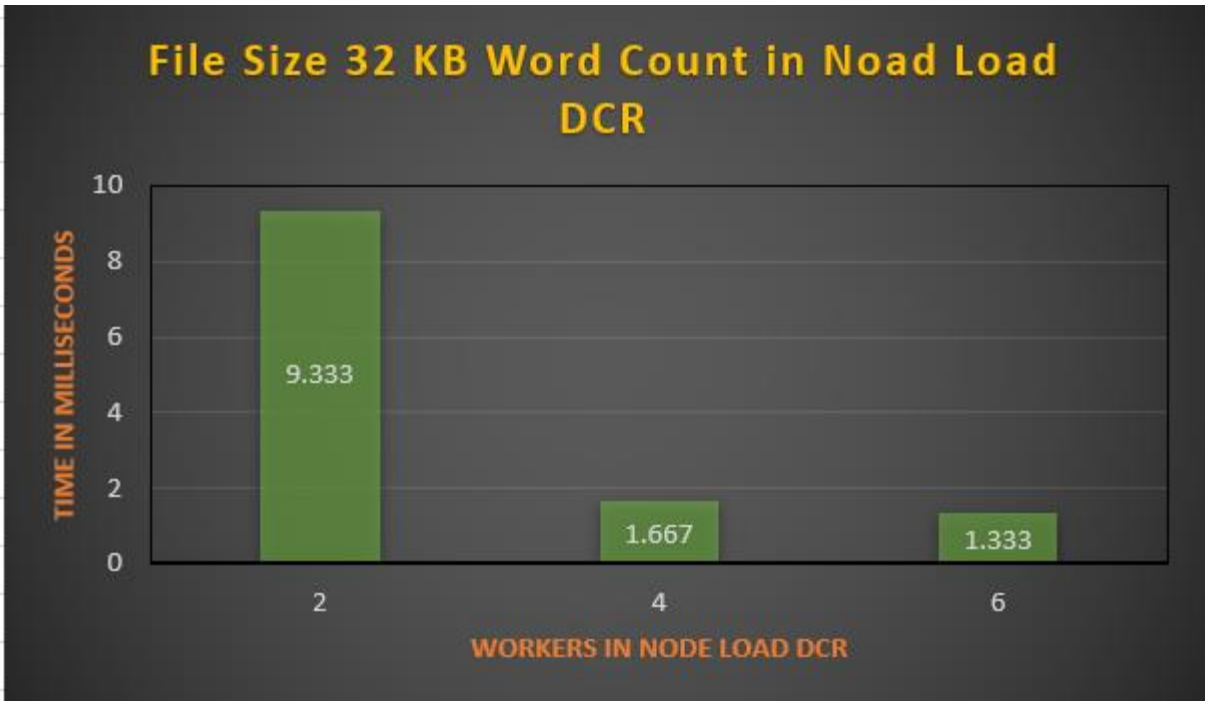

Figure 12: NDCR Word Count 32KB file

- 455MB text file and 2, 4, 6 workers.

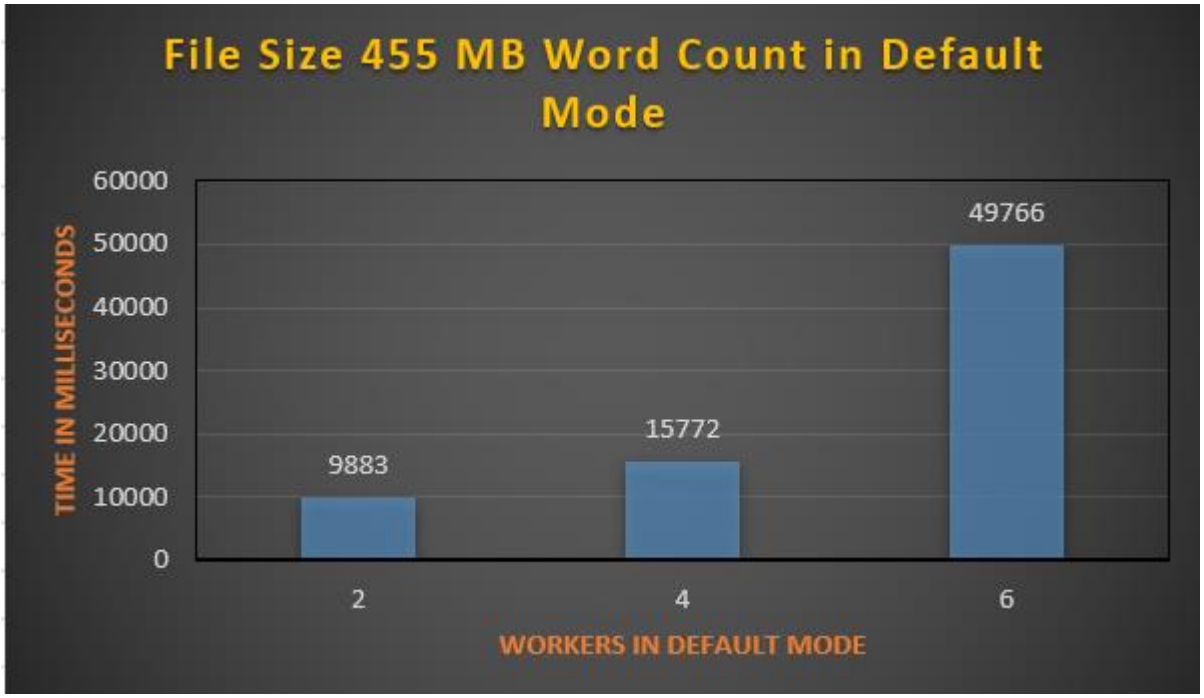

Figure 13: Default Word Count 455MB text file

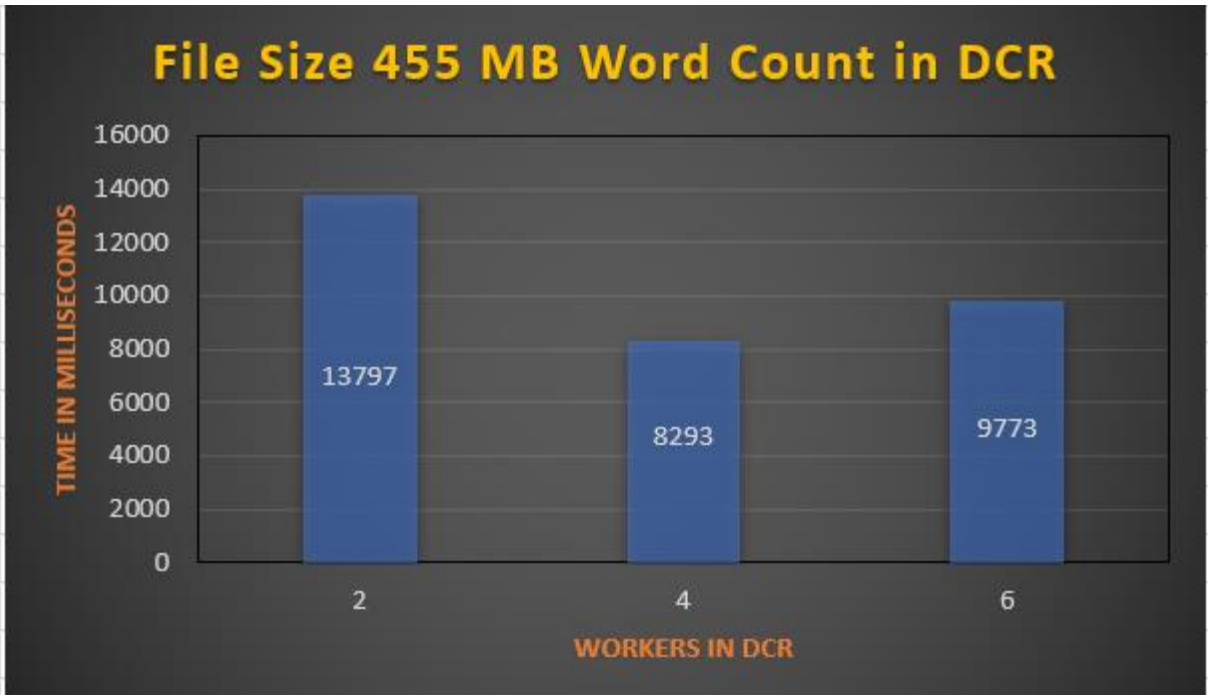

Figure 14: DCR Word Count 455MB file

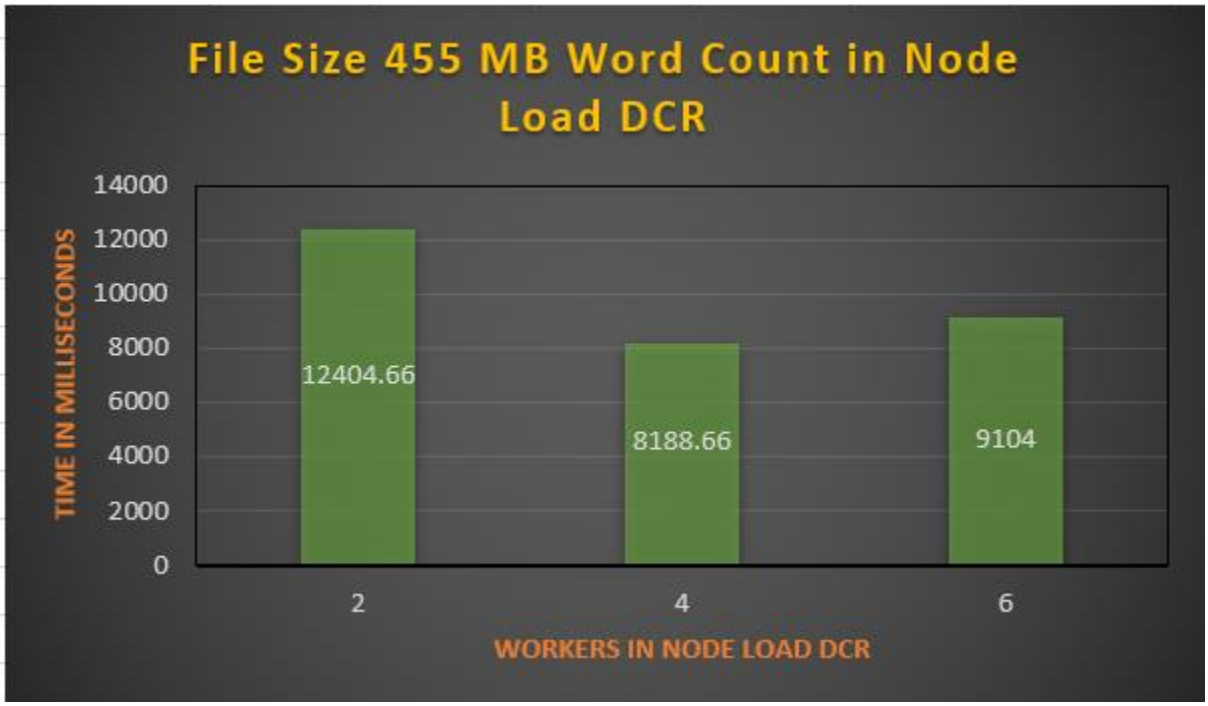

Figure 15: NDCR Word Count 455MB file

The results indicate improved time performance for each file size as the number of workers in the cluster increase. The NDCR is found to performs better in terms of the time taken over the DCR and Default. In every case, whether DCR / NDCR / Default, with the increasing number of workers, the performance increases. The NDCR times for each case are the average of three iterations of the NDCR Word Count operation.

Figure 16: Performance comparisons of Data Distribution Schemes

Using a sorted file sorted just based on the words, does not yield any better performance than the default word count operations, strongly suggesting better ways of sorting the data is needed to achieve key-based data locality in a cluster.

8. Conclusions and Future Work

This thesis proposes an idea to dynamically improve the data compute ratios of the worker nodes while also considering their compute capabilities in the first place to enhance the performance of heterogeneous clusters in the cloud. The traditional data distribution method assumes the same compute capabilities for all the worker nodes, which affects the performance in the heterogeneous cloud. The proposed solution bases the nodes' computing capacities based on their speeds of execution and their dynamically varying workloads.

Nine experiments were conducted with varying cluster workers, different file sizes, and different data distribution schemes to verify the improvements. It is seen with the dynamically changing workloads for the worker nodes the time taken for the word count operations decreases with an increasing number of workers. The NDCR word count operations perform better than DCR word count operations for files of a particular size. It is observed with the growing file sizes, the time taken for the operation increases across the different clusters; however, there is a decrease in time taken within a cluster with an increasing number of workers.

Default's performance is the least when compared with the DCR and NDCR word count operations. This is because the heterogeneity of the cluster is not considered during the task execution. The improved data locality in the case of NDCR and DCR data distribution schemes improves the performance by decreasing the time taken for the task in a cluster. There are considerable improvements in the performance. Even though a very significant increase might be due to the data sizes used in the experiment, the nodes may have gone through any network bottleneck during the calibration phase, and this investigation is for future work. Replication for fault tolerance can also affect the data computation ratio, which will be investigated in future work, data locality can be improved by handling the slow nodes and replication factor being addressed.

References

[1] **“Investigation of data locality and fairness in MapReduce** [Conference] / auth. Z. Guo G. Fox, and M. Zhou // HPDC. - Delft : ACM, 2012. - pp. 25-32.

[2] **A review on Hadoop–HDFS infrastructure extensions** [Conference] / auth. Chitharanjan A. Kala Karun and K. // ICT. - Tamil Nadu : IEEE, 2013. - pp. 132-137.

[3] **Adaptive Dynamic Data Placement Algorithm for Hadoop in Heterogeneous Environments.** [Journal] / auth. Avishan Sharafi and Ali Rezaee // Journal of Advances in Computer Engineering and Technology. - 2016. - 4 : Vol. 2. - pp. 17-34.

[4] **BAR: An efficient data locality driven task scheduling algorithm for cloud computing** [Conference] / auth. J. Jin J. Luo, A. Song, F. Dong, and R. Xiong // CCGRID. - NewPort Beach : IEEE, 2011. - pp. 295-304.

[5] **CoHadoop: Flexible data placement and its exploitation in Hadoop** [Journal] / auth. M. Y. Eltabakh Y. Tian, F. Ozcan, R. Gemulla, A. Krettek, and J. McPherson // Proc. VLDB Endow.. - 2011. - 9 : Vol. 4. - pp. 575-585.

[6] **Data placement for scientific applications in distributed environments** [Conference] / auth. A. Chervenak E. Deelman, M. Livny, M.-H. Su, R. Schuler, S. Bharathi,G. Mehta, and K. Vahi // GRID. - Austin : IEEE, 2007. - pp. 267-274.

[7] **Improving MapReduce performance through data placement in heterogeneous Hadoop clusters** [Conference] / auth. J. Xie S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin // IPDPS 2010. - Atlanta : IEEE, 2010. - pp. 1-9.

[8] **Investigation of data locality and fairness in MapReduce** [Conference] / auth. Z. Guo G. Fox, and M. Zhou // HPDC'12. - Delft : ACM, 2012. - pp. 25-32.

[9] **Maximizing data locality in distributed systems** [Journal] / auth. F. Chung R. Graham,

R. Bhagwan, S. Savage, and G. M. Voelker // J. Comput. Syst. Sci.. - 2006. - 8 : Vol. 72. - pp. 1309-1316.

[10] **Novel Data-Distribution Technique for Hadoop in Heterogenous Cloud Environments** [Conference] / auth. Vrushali Ubarhande Alina-Madalina Popescu, Horacio Gonzalez–Velez // Ninth International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS). - 2015.

[11] **Performance evaluation of MapReduce using full virtualization on a departmental cloud** [Journal] / auth. Kontagora H. Gonzalez-Velez and M. // Int. J.Appl. Math. Comput. Sci.. - 2011. - 2 : Vol. 21. - pp. 275-284.