National College of Ireland

Novel Algorithm with In-node Combiner for enhanced performance of MapReduce on Amazon EC2

MSc Research Project
Cloud Computing

Ashwini Rajaram Chandanshive
x15043584

School of Computing
National College of Ireland

Supervisor: Dr. Horacio Gonzalez-Velez

National College of Ireland
Project Submission Sheet – 2015/2016
School of Computing

Student Name:	Ashwini Rajaram Chandanshive
Student ID:	x15043584
Programme:	Cloud Computing
Year:	2016
Module:	MSc Research Project
Lecturer:	Dr. Horacio Gonzalez-Velez
Submission Due Date:	16/08/2017
Project Title:	Novel Algorithm with In-node Combiner for enhanced performance of MapReduce on Amazon EC2
Word Count:	4194

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are encouraged to use the Harvard Referencing Standard supplied by the Library. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action. Students may be required to undergo a viva (oral examination) if there is suspicion about the validity of their submitted work.

Signature:	
Date:	15th September 2017

PLEASE READ THE FOLLOWING INSTRUCTIONS:

1. Please attach a completed copy of this sheet to each project (including multiple copies).
2. **You must ensure that you retain a HARD COPY of ALL projects**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. Please do not bind projects or place in covers unless specifically requested.
3. Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Novel Algorithm with In-node Combiner for enhanced performance of MapReduce on Amazon EC2

Ashwini Rajaram Chandanshive
x15043584
MSc Research Project in Cloud Computing

15th September 2017

Abstract

To distribute large datasets over multiple commodity servers and to perform a parallel computation a Hadoop framework is used. A question that arises with any program is efficiency of the program and its completion time. MapReduce programming model uses the divide and conquer rule, the map (reduce) tasks consists of specific, well defined phases for data processing. However only map and reduce functions are custom and their execution time can be predicted by user. The execution time for the remaining phases is generic and totally depends on the amount of data processed by the phase and the performance of underlying Hadoop cluster. The optimization of I/O can contribute towards the better performance. Hence in this paper, we will look into such I/O bottlenecks that Hadoop framework faces and a possible solution to overcome the same. We have introduced an approach that will help to optimize I/O, the combining at a node level. This design has taken the traditional combiner to a next level wherein the number of intermediate results are reduced with the help of combiner at a node level which results in reduced network traffic between mappers and reducers.

1 Introduction

Hadoop is an open-source framework developed for storing data and running applications on clusters of commodity hardware. Hadoop is being a popular in the field of Big Data because of its unique features. First of all, Hadoop provides massive storage that can store nearly any kind of data, followed by its massive processing power and finally comes the ability to handle concurrent tasks.

MapReduce programming model (Dean and Ghemawat; 2008) is considered as a heart of Apache Hadoop, pioneered by Google especially for data intensive and large-scale data analysis applications. MapReduce programming model is actually based on two simple functions map and reduce that can be used for the functional programming. Map function processes an input data and generates a key-value pair and generates a set of intermediate key-value pairs while reduce functions takes that intermediate key-value pairs and merge them to create a final desired output. In short, Map functions takes the input data and creates s key-value pair and pass this output further, these output is then taken as

an input for reduce function and merged together to provide final output(**?**). A typical MapReduce programming model comprises of three simple steps.

(i) Map Phase : Name node is the one responsible to retrieve locations of input data nodes that are distributed over multiple data nodes. These input blocks are loaded from local disk into memory then corresponding blocks are processed by each mapper task. The main function of the Map task is to take the input process it and split it into smaller pieces and assign key-value pair. These intermediate results from each map task are happened in map output buffers.

(ii) Shuffle and Sort Phase : Once map task completed, the output of map tasks is merged and shuffled to corresponding reduce tasks across the network.

(iii) Reduce Phase : Received key groups are processed by each reduce task. As map phase stores its output into map buffers similarly reduce phase stores the input into reducer output buffers temporarily. Final output is written to HDFS once all groups are processed.

The figure below shows the three phases of MapReduce Figure 1

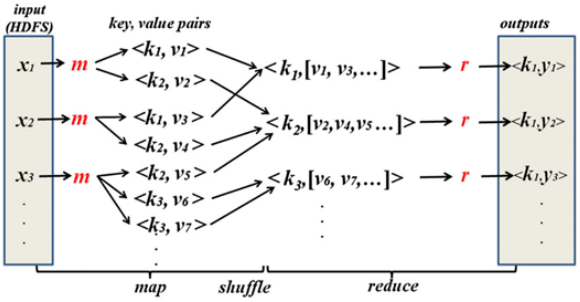

Figure 1: MapReduce three phases

Performance is becoming a key issue with increasing demand for both non-batch and real-time processing with Hadoop for MapReduce applications. An acceptable job completion time is an important aspect for performance-based jobs. Therefore to minimize the number of I/O must be an ultimate goal for an efficient MapReduce job in each I/O focused phases as given above. This work helps to understand How an aggregating intermediate results locally using the combiner can optimize the overall performance of a MapReduce job?

2 Related Work

The main purpose of the literature review is to evaluate work done so far in the area of performance improvement for the Hadoop MapReduce in order to understand the research gap. Section 2.1 describes the basic working of Hadoop MapReduce. The evolution of the existing technologies that contributed towards the performance enhancement of both Hadoop and MapReduce is covered under section 2.2.

2.1 Background

To process large dataset in a distributed environment MapReduce framework is considered to be a powerful model. Each MapReduce phase

2.1.1 MapReduce

For simplified data processing on large clusters Google developed a MapReduce programming model (Dean and Ghemawat; 2008). As per name, Map and Reduce form a pretty simple structure. The overall logic for the processing of the MapReduce is when MapReduce function is called, MapReduce programs divides user input data into smaller chunks and starts with many copies of program with one chunk each. Among these copies one of the copies becomes the master program that manages the complete execution. First, map tasks read the input that is divided into smaller chunks and generates (key/value) pairs from data. Further reduce tasks reads those values with same keys, process values and create output. The master program returns the desired output based on reduce tasks. In case of failure of any workers master re-executes failed tasks on another worker node.

2.1.2 Combiner

The Combiner provided as a Hadoop API that performs partial merging of intermediate data before sending them to reducers. In a case where intermediate results contain significant number of repetitions and that are meant for the same reducers, then the combiner can considerably reduce the intermediate results thus save on communication cost without even altering the final output.

Theoretically combiners should improve performance of MapReduce job considerably by intermediate combinable results which will help to cut down the network communication cost. However, combiners have a downside.

1) Combiners execution cannot be guaranteed at a times :
This means, there could be a case that Hadoop may choose not to execute the combiners if it identifies that the execution of the combiners is inefficient for the system. A known case is when number of spill files does not exceed the configurable limit. Other case might be where developers did not systematically control it.

2) Map output size is not optimized :
The output from maps is stored in in-memory buffers temporarily and combiners function

on the same before spilling further to local disk. Therefor it is clear that combiners do not reduce the number of output from maps.

2.1.3 In-Mappers combiners

The two problems associated with the traditional combiners are resolved by the of in-mapper combiner. In-mapper combiner is a concept where the combiner is introduced at the map method and to minimize volume of the intermediate results yield by it. In-mapper combiner stores and aggregate results in an associative array, that is indexed by output keys and produce the at the end of the map task. It is clear that with this approach a substantial reduction in the total number of emitted map output is observed. Figure shows the pseudo code for the word count MapReduce job using in mapper combiner.The total number The figure below shows the thre phases of MapReduce Figure 2

```
1: class Mapper
2:         method Setup()
3:                 H ← InitAssociativeArray()
4:         method Map(long id, twit t)
5:                 d ← ExtractDate(t)
6:                 W ← BagOfWords(t)
7:                 for all words w ∈ W do
8:                         H{d, w} ← H{d, w} + 1
9:         method Cleanup()
10:                for all date-word-pair dw ∈ H do
11:                        Emit(date-word-pair dw, count H{d, w})
```

```
1: class Reducer
2:         method Reduce(date-word-pair dw, counts [c1, c2, ....])
3:                 s ← InitCount()
4:                 for all count c ∈ counts do
5:                         s ← s + c
6:                 Emit(date-word-pair dw, sum s)
```

Figure 2: WordCount algorithm for In-mapper

The figure below shows the thre phases of MapReduce Figure 3

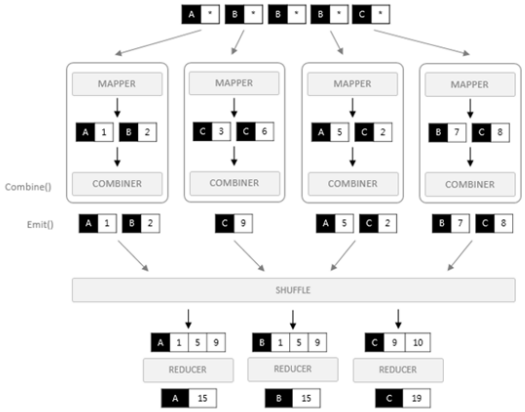

Figure 3: MapReduce job design with In-mapper

2.2 Literature Review

Hadoop is designed to handle large data and with its increasing use there is always a demand for increasing volume of data storage and analysis. That often results into a degraded performance. We have looked into some work that has been done in order to improve Hadoop performance. There are always been various approaches that had been taken to tackle this performance issue.

Gu et al. (2013)proposed a data prefetching mechanism in which data can be fetched in advance to corresponding compute node that will improve the MapReduce performance in heterogeneous or shared environment. Yao et al. (2015) came up with a new approach for the efficiency improvement wherein they developed a new scheduler, this scheduler will understand the workload patterns and tune the shared resources among users and the scheduling algorithms for each user dynamically for the performance improvement. Another study that has been carried out by Islam et al. (2016) shows data access strategies that can improve the read performance of Hadoop Distributed File System. Results based on this study shows that the HDFS read performance is improved by up to 33 compared to the default locality aware data access. In Babu (2010)the authors have compared MapReduce with parallel database system where on the other hand Dean and Ghemawat (2010) described the affect of the job configuration parameters on the Hadoop performance. The focus on architectural issues and their possible solutions which will help to improve the overall performance is given by (Jiang et al.; 2010)

During recent years, much attention is given to MapReduce environment, to performance modeling and workload management and (Herodotou, Dong and Babu; 2011; Herodotou, Lim, Luo, Borisov, Dong, Cetin and Babu; 2011; Verma et al.; 2011b,a)are some of the different approaches to predict of MapReduce application performance were offered.

Apart from this there is a need to develop a practical performance model that will give estimation about the performance and help to analyze the bottleneck for the MapReduce job. Literature has evidences that clearly show many work has already been done for the modeling of the MapReduce jobs and their performance. However, existing performance models are likely to ignore some of the important factors such as I/O congestion, tasks failure over clusters that will ultimately contribute towards the execution cost of the MapReduce jobs.

Hadoop Distributed File System is a best suitable for data-intensive applications due to its extensive scalability and fault tolerance schema.Ye et al. (2016) ported an MPI-SVM solver, which was originally developed for HPC environment to the HDFS which actually improved the pre-processing of data that requires huge amount of I/O operations by a deterministic scheduling. Thus, this design successfully eliminates the overhead lead by remote I/O operations, with the help of SVM algorithm and which will be beneficial to many other algorithms when trying to copy large scale of data.

I/O optimization becomes a discouraging work sometimes as applications source code is not always available. To understand the I/O behavior in such cases where source code is unavailable the traces of Hadoop plays very important role. With this method we can

not only understand the bottlenecks but also help to improve the performance. Feng et al. (2015) introduced a tool suite which is quite transparent tracing and analysis tool. This can be plugged into Hadoop system. This tolls also help to realise different approaches with tracing; can release the tracer, without any source code modification of target can trace the I/O operations. Additionally this work introduced an analyzer which provides new approaches to address I/O problems depending upon their access patterns. The experimental results from this study prove its effectiveness and the overhead can also be observed as low as 1.97.

Park (2016) proposed a new framework named HDFS-AIO. This will enhance HDFS with Adaptive I/O system. It basically supports various I/O methods for upper application, making it enable to select optimal I/O routines for the platform without even having the knowledge of the source code modification and re-compilation.

Furthermore, looking into the HDFS performance issue, it is observed that most common cause behind the HDFS performance degradation is poor I/O performance. To handle such situation we need to categorize this into two such as merging stored files into forms of databases or modification of the existing I/O feature of HDFS (Zhang et al.; 2012). Although first approach is successful in a way to improve throughput by indexing data blocks but does not take care of the I/O performance. The second approach will require a complete redesigning of the entire Hadoop system which sounds risky.

To reduce the intermediate result size (Dean and Ghemawat; 2008)suggested to use combiners in MapReduce jobs. The in-mapper combining design, which is actually considered as a improvement of the traditional combiner was introduced by (Lin and Schatz; 2010). This design executes the combining function under the map method.

Therefore the research done in this area can be conclude as given in below table. The table helps to understand the related work done so far to improve the performance of the MapReduce however we do realise that there is still a gap that encouraged us to work further towards the performance improvement of the same.

Related Paper	Work Done
(Dean and Ghemawat; 2008)	Introduced combiner, minimized the intermediate results
(Lin and Schatz; 2010)	Introduced In-mapper combined design, improved traditional design
(Zhang et al.; 2012)	Improved throughtput by indexing data blocks, poor I/O performance

Table 1: Summery of Related Work

3 Methodology

In this section we would like to look into the tools that are required to evaluate the performance of the MapReduce application, dataset that we used to check our approach of the performance optimization.

3.1 Proposed Solution

Step 1 : Input data is processed by Map

Step 2 : Map tasks divides data into smaller chunks and intermediate data is generated by assigning (key, value) pair

Step 3 : Intermediate result is from each map is then processed by in-node combiner

Step 4 : In-node combiner combines the intermediate result from each map from the same node and passes the output further to process to reduce

Step 5 : Shuffle and sort phase takes these as a input before passing it further to final reduce phase

Step 6 : Reduce phase takes the output from shuffle and sort and depending upon the key put the data together for final desired output.

However we cannot just trust combiner as a solution because we implement the same in any MapReduce job it is not always the case that will get executed. In-mapper design pattern may improve the execution speed of the MapReduce job by just reducing the number of intermediate results yield by mappers to reducers. As combiners may or may not run at all therefore in-mapper design is the best option to be sure that it will run once it is implemented. There are different ways in which in-mapper combiner can be implemented such as local and global implementation. In-Node Combiner (INC)

The idea to introduce In-node combiner is to combine results from map within a particular node. The advantage of this approach is that we do not have to manage the each and individual array of intermediate results from every mapper, intermediate arrays are merged into single result locally at that particular node.

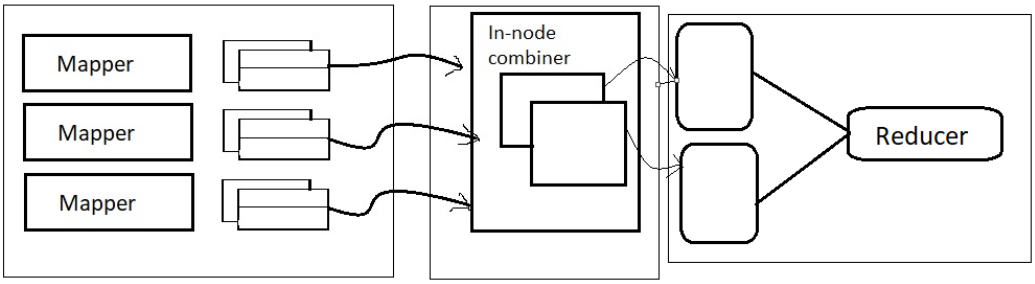

Figure 4: In-node Combiner Approach

The benefits that we can obtain from such approach are as below: 1) Minimized number of total results yield by node:
This local margining of results from mapper at the node level reduces the total number of intermediate results that passed further to reducers. Such output requires less amount of space in the map output buffer and last mapper sends the merged output further to

process. This also leads to less generation of spill files. Eventually with this approach we can achieve a reduced communication cost.

2) Function that takes care of the combining is executed separately:
In-mapper compromises between performance and parallelism by introducing combining function into map method. However in-node combiner the task such as merging output is done into a separate thread.

3.2 Tools and software involved

To evaluate the MapReduce performance in our experiment we have considered Amazon Elastic Compute Cloud (Amazon EC2) provided by amazon web services. We created our instances using EC2 which is a web service that provides secure, resizable compute capacity in the cloud. This simple web service interface from Amazon, allows obtaining and configuring capacity with minimal friction. The main reason that we chose to carry our experiment using EC2 is due to its easy configuration of instances that reduces the time required to obtain and boot of any server instance. Additionally, its elastic web-scale computing makes it unique choice for the experiment. Furthermore, its pay policy of pay as you go makes it clear and simple in terms of economics of computing where we just have to pay for capacity that we have actually use making it inexpensive choice. Also it is a very reliable and secure way of building our environment as Amazon EC2 provides not only failure resilient applications but also isolate them from common failure scenarios tool for the developers.

Hadoop an open-source implementation is a very popular choice of the Googles MapReduce model which was primarily developed by Yahoo. [1]. Yahoo servers use Hadoop, where at least 10,000 cores are responsible to generate hundreds of terabytes of data.[Yahoo! launches worlds largest hadoop production application. [2] On the other hand, Facebook uses Hadoop every day to process their more than 15 terabytes of new data. Along with Yahoo and Facebook there is huge number of other websites as well that uses the Hadoop implantation to manage their massive amount of data such as Amazon and Last.fm, to name a few.Pike et al. (2005) Hadoop is not only popular for web intensive applications it is also a preferred choice among scientific data-intensive applications and make maximum out of the Hadoop systems.Olston et al. (2008) (Pike et al.; 2005)

This popular implementation uses MapReduce programming model along with Hadoop Distributed File System to process massive amount of data. Its an easy deployment by configuring some variable that defines the paths and required nodes. Hadoop cluster supports master slave architecture where there is one master node that takes care of the management of all systems and jobs, and other worker nodes. Google proposed the MapReduce programming model to support data-intensive applications running on parallel computers like commodity clusters. With the help of the MapReduce framework simplifies the complexity of running distributed data processing functions across multiple nodes in a cluster. This approach of MapReduce enables a programmer to write their MapReduce functions to run in parallel across multiple nodes in the cluster with

[1] `http://lucene.apache.org/Hadoop`
[2] http://tinyurl.com/2hgzv7

no specific knowledge of distributed programming knowledge (Dean and Ghemawat; 2008)

3.3 About dataset

The dataset used for the experiment is a stock exchange data which contains 500,000 records about the stock rates for different countries and schema of data is as below.

id,exchange	stockname
sector	country
date	open
high	low
close	volume
adjcloseid	exchange
stockname	sector
country	date
open	high
low	close
volume	

3.4 Wordcound Algorithm

For the performance evaluation we have implemented a simple MapReduce algorithm. The word count algorithm counts occurrences of every word in the given dataset and provides results in the separate file.

4 Implementation

Implementation is an important part of any research. In this section we discuss the implementation in detail that we did for to carry out our research. Section 4.1 gives an idea about the Hadoop implementation on the amazon EC2, followed by section 4.2 gives the details about the input. Finally, under section 4.3 the various solutions that we required to fine tune our Hadoop /MapReduce implementation are described.

4.1 Implementation of Hadoop

For the Apache Hadoop implementation we created our instances on Amazon EC2. The instances are built using Ubuntu Server 16.04 LST with instance type t2.large. We have done the multi node cluster setup for our experiment wherein we have considered a master and 2 datanodes.

Before we start with the actual Hadoop implementation we have to take care of the runtime environment.
The figure below shows the per process runtime environment Figure 5

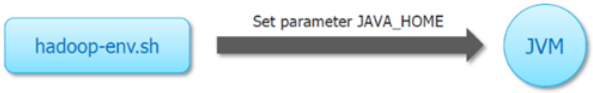

Figure 5: Per process runtime environment

The file Hadoop-env.sh is sourced by entire Hadoop core scripts and it is under conf/ directory To setup the multi node cluster we have to edit below files: [3]

1) core-site.xml
This is necessary for Hadoop daemons to understand where Namenode runs in the cluster. Additionally, it contains the configuration settings for Hadoop Core that are common to both HDFS and MapReduce for example I/O setting.
2) hdfs-site.xmln
This file contains the HDFS configuration settings that is NameNode, Secondary NameNode and the DataNodes.
3) mapred-site.xml
It consists of the MapReduce daemons such as the job tracker and the task-tracker.

4.2 Input to the Hadoop

As we are working on Amazon EC2 instances where we work on command line, thus to transfer the input dataset from local machine to AWS instance we have used Winscp.

4.3 MapReduce Implementation

The listing below gives the main components of MapReduce jobs. To execute any logic for MapReduce application it is required to have two Java classes namely Mapper and Reducer which is done by job.setMapperClass( ) and job.setReducerClass( ) respectively. The execution flow goes like this: First mappers output is written into the disk, before this execution, this intermediate output is processed by the shuffle and sort process and passed further to reducer to have the desired output. The figure below indicates the clas implementation of Map and Reduce Figure 6

5 Evaluation

We have carried out our experiment on Hadoop version 2.7.3 installed on Ubuntu 16.4 LST operating system and created our instances on Amazon EC2.

To evaluate the performance of the MapReduce task we have carried out three different approaches compared for a given input with for wordcount algorithm and HDFS as the source of Input for all three combining approaches. We have observed that number of Map tasks varies depending upon the input data size. In the experiment we have input the data set with 5 lacs records which is a stock exchange data for different countries

[3]https://www.edureka.co/blog/explaining-hadoop-configuration/

```
Job job = new Job(conf, ...);
job.setJarByClass(CombinerClass.class);

job.setMapperClass(...); //parsing input record
job.setReducerClass(...);
job.setCombinerClass(...);

job.setOutputKeyClass(...);
job.setOutputValueClass(...);

job.setInputFormatClass(...);
job.setOutputFormatClass(...);
```

Figure 6: Per process runtime environment

comprising of both number and text in it.

5.1 Experiment 1

In our first experiment, we executed a wordcount program with a traditional combiner and received results as below.

In Table 2 Traditional Combiner.

Approach	MapOutput	Reduce output
Traditional Combiner	7050905	676247

Table 2: Results for Traditional Combiner

5.2 Experiment 2

For experiment 2 we considered executing the wordcount program with In-mapper approach and the results of the same are as given in table below. The output of the same is given in table below.

In Table 3 Traditional Combiner.

Approach	MapOutput	Reduce output
In-Mapper Combiner	6003898	54279

Table 3: Results for In-mapper Combiner

5.3 Experiment 3

In this experiment we executed the wordcount program with our new approach that is In-node combiner. The output of the same has given in table below.

From the given results below is is observed that we have successfully reduced the number of map intermediate results at the node level. Additionally we have seen that when the time it taken to complete the execution is comparatively faster than the rest of the two approaches. As there are less number of intermediate results from map.

Approach	Map Output	Reduce output
In-node Combiner	573910	7

Table 4: Results for In-node Combiner

The table above shows the results we got in experiment 3. In this experiment, the dataset we considered in this experiment has 50 lacs records. When the wordcount program was executed, as per the table below we can see that 573910 Map output was generated and passed it further to the in-node combiner. In-node combiner processed 573910 records and merged these intermediate results locally that is at node level before passing it to Reduce phase.

Approach	Map Output	Reduce output
Traditional Combiner	7050905	676247
In-Mapper Combiner	6003898	54279
In-node Combiner	573910	7

Table 5: Results for all approches

Results showing execution time all approaches and completion time

In Table 6 Results from all the approaches.

Results that we received in our experiment with in-node approach shows that we are quite successful meeting our goal of reducing the number of intermediate results at node level.

Approach	Map	Reduce Time
Traditional	7050905	676247 27
In-Mapper	6003898	54279 14
In-node	573910	7 10

Table 6: Overview of all results

5.4 Discussion

With the introduction of the combiner at node level we tried to aggregate all intermediate map results at the node level. This extended approach to the traditional combiner has improved the performance in aspects of network I/O as there are less number of intermediate results shuffled across the network resulting in the reduced network communication cost. Additionally this approach made the execution of the wordcount algorithm faster. That is we gradually observed the reduction in the execution time. The less number of intermediate results made the algorithm execution faster therefore it would be possible that we can use the same approach for the big datasets in future to see the reduced execution time with better performance.

Therefore we can say that with in-node combiner we are able to improve the Hadoop/MapReduce performance.

6 Conclusion and Future Work

Tuning Hadoop/MapReduce
The only issue we faced during implementation and while starting up all Hadoop services is below errorFigure 7

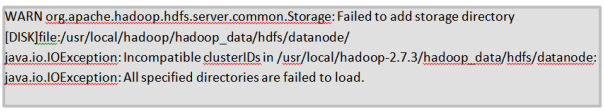

Figure 7: Per process runtime environment

In order to solve this error we had to follow the workaround as below:
sudo rm -rf /usr/local/Hadoop/Hadoop_data/hdfs/datanode/*

In this thesis we have proposed a new designing approach that will help developer to develop their very own applications. As seen in earlier section, the main factors responsible for the degradation of the Hadoop performance are Hadoop MapReduce job execution workflow and bottlenecks. Main factor to affect the performance is I/O performance. Thus, to achieve the better performance disk as well as network I/O through all the MapReduce execution phases should be maintained properly to have a significant results.

Our approach, we introduced the combiner at node level and to aggregate all the map results at the node level. This approach can be considered as an extension to the

traditional combiner. As it improves the performance of the traditional combiner. Thus it seems to be a better way of dealing with the network overhead as with this approach we are aggregating the map results at node level and then passing it further to the reducer for further processing of the final result. The experiment results show that in-node combiner is quite successful in order to reduce the number of intermediate results obtained from each map. As there are less number of map results shuffled over the network we can say that we have also successfully managed to reduce on the communication cost of the same. With the new approach of in-node mapper we aimed to improve the performance of the MapReduce tasks and with our findings we can say that the performance has been increased as we look into the job completion time of all the three approaches.

With our new approach we are able to say that we have the better I/O performance and in the future we would like to focus on the caching the results from the memory.

References

Babu, S. (2010). Towards automatic optimization of mapreduce programs, pp. 137–142.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters, *Communications of the ACM* **51**(1): 107–113.

Dean, J. and Ghemawat, S. (2010). Mapreduce: a flexible data processing tool, *Communications of the ACM* **53**(1): 72–77.

Feng, B., Yang, X., Feng, K., Yin, Y. and Sun, X.-H. (2015). Iosig+: on the role of i/o tracing and analysis for hadoop systems, pp. 62–65.

Gu, T., Zuo, C., Liao, Q., Yang, Y. and Li, T. (2013). Improving mapreduce performance by data prefetching in heterogeneous or shared environments, *International Journal of Grid and Distributed Computing* **6**(5): 71–82.

Herodotou, H., Dong, F. and Babu, S. (2011). No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics, p. 18.

Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F. B. and Babu, S. (2011). Starfish: A self-tuning system for big data analytics., **11**(2011): 261–272.

Islam, N. S., ur Rahman, M. W., Lu, X. and Panda, D. K. D. K. (2016). Efficient data access strategies for hadoop and spark on hpc cluster with heterogeneous storage, pp. 223–232.

Jiang, D., Ooi, B. C., Shi, L. and Wu, S. (2010). The performance of mapreduce: An in-depth study, *Proceedings of the VLDB Endowment* **3**(1-2): 472–483.

Lin, J. and Schatz, M. (2010). Design patterns for efficient graph algorithms in mapreduce, pp. 78–85.

Olston, C., Reed, B., Srivastava, U., Kumar, R. and Tomkins, A. (2008). Pig latin: a not-so-foreign language for data processing, pp. 1099–1110.

Park, J. K. (2016). Improving the performance of hdfs by reducing i/o using adaptable i/o system, pp. 3139–3144.

Pike, R., Dorward, S., Griesemer, R. and Quinlan, S. (2005). Interpreting the data: Parallel analysis with sawzall, *Scientific Programming* **13**(4): 277–298.

Verma, A., Cherkasova, L. and Campbell, R. H. (2011a). Aria: automatic resource inference and allocation for mapreduce environments, pp. 235–244.

Verma, A., Cherkasova, L. and Campbell, R. H. (2011b). Resource provisioning framework for mapreduce jobs with performance goals, pp. 165–186.

Yao, Y., Tai, J., Sheng, B. and Mi, N. (2015). Scheduling heterogeneous mapreduce jobs for efciency improvement in enterprise clusters, *IFIP/IEEE InternationalSymposium on IntegratedNetwork Management (IM2013):Short Paper* pp. 872–875.

Ye, M., Wang, J., Yin, J. and Zhang, X. (2016). Accelerating i/o performance of svm on hdfs, pp. 132–133.

Zhang, J., Wu, G., Hu, X. and Wu, X. (2012). A distributed cache for hadoop distributed file system in real-time cloud services, pp. 12–21.

A Source Code

```java
import java.io.IOException;
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.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;

public class WordCount {
        public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(WordCountMapper.class);
        job.setCombinerClass(WordCountReducer.class);
        job.setReducerClass(WordCountReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
    public static class WordCountMapper extends Mapper<Object, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(Object key, Text value, Context context) throws IOException,
                InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }
```

```java
    public static class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

}
```

Figure 1: Word Count Algorithm

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.fs.Path;

public class CombinerClass {

/**
Mapper class to filter only relevant data.
*/
public static class Map extends Mapper<LongWritable,Text,Text,IntWritable>
{
public void map(LongWritable key,Text value,Context context) throws IOException, InterruptedException
{
int i=1;
String S[]=value.toString().split("\t");
// Write output with exchange field as key and count 1 as value.
context.write(new Text(S[1]),new IntWritable(i));
}
}

/**
Combiner class to reduce the data at node level
*/
public static class Combiner extends Reducer<Text,IntWritable,Text,IntWritable>
{
public void reduce(Text key,Iterable<IntWritable> value,Context context) throws IOException, InterruptedException
{
int sum=0;
for(IntWritable v:value)
sum+=v.get();
```

```
context.write(key,new IntWritable(sum));
}
}

@SuppressWarnings("deprecation")
public static void main(String[] args) throws Exception
{

Configuration conf= new Configuration();

Job job = new Job(conf,"My Combiner Program");

job.setJarByClass(CombinerClass.class);
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
job.setCombinerClass(Combiner.class);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);

FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));

//exiting the job only if the flag value becomes false

System.exit(job.waitForCompletion(true) ? 0 : 1);
}

}
```

Figure 2: In Node Word Counr Algorithm