A FRAMEWORK FOR AUTOMATIC OPTIMIZATION OF MAPREDUCE PROGRAMS BASED ON JOB PARAMETER CONFIGURATIONS

By

PRAVEEN KUMAR LAKKIMSETTI

B. Tech., Vellore Institute of Technology University, 2009

A REPORT

Submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2011

Approved by:

Major Professor
Dr. Mitchell L. Neilsen

Abstract

Recently, cost-effective and timely processing of large datasets has been playing an important role in the success of many enterprises and the scientific computing community. Two promising trends ensure that applications will be able to deal with ever increasing data volumes: first, the emergence of cloud computing, which provides transparent access to a large number of processing, storage and networking resources; and second, the development of the MapReduce programming model, which provides a high-level abstraction for data-intensive computing. MapReduce has been widely used for large-scale data analysis in the Cloud [5]. The system is well recognized for its elastic scalability and fine-grained fault tolerance.

However, even to run a single program in a MapReduce framework, a number of tuning parameters have to be set by users or system administrators to increase the efficiency of the program. Users often run into performance problems because they are unaware of how to set these parameters, or because they don't even know that these parameters exist. With MapReduce being a relatively new technology, it is not easy to find qualified administrators [4].

The major objective of this project is to provide a framework that optimizes MapReduce programs that run on large datasets. This is done by executing the MapReduce program on a part of the dataset using stored parameter combinations and setting the program with the most efficient combination and this modified program can be executed over the different datasets. We know that many MapReduce programs are used over and over again in applications like daily weather analysis, log analysis, daily report generation etc. So, once the parameter combination is set, it can be used on a number of data sets efficiently. This feature can go a long way towards improving the productivity of users who lack the skills to optimize programs themselves due to lack of familiarity with MapReduce or with the data being processed.

Table of Contents

List of Figures ... v

List of Tables ... vii

Acknowledgements ... viii

Chapter 1 - INTRODUCTION ... 1

Chapter 2 - BACKGROUND ... 4

2.1 HADOOP ... 4

2.1.1 MapReduce ... 4

2.1.2 HDFS (Hadoop Distributed File System) ... 7

2.1.3 Hadoop Stack ... 8

2.1.4 Applications ... 9

Chapter 3 - RELATED WORK ... 10

Chapter 4 - IMPLEMENTATION ... 16

4.1 Optimizer ... 19

4.2 Contributor ... 21

Chapter 5 - RESULTS AND EVALUATION ... 24

5.1 Vector Space Model Evaluation ... 25

5.2 PageRank Evaluation ... 28

5.3 Word Count Evaluation ... 31

5.4 Sort Evaluation ... 34

5.5 Optimizer Evaluation ... 38

5.5.1 Anagram Evaluation ... 38

5.5.2 Distributed Grep Evaluation ... 42

5.6 Testing ... 45

5.6.1 Unit Testing ... 45

5.6.2 Functional Testing (Black Box) ... 46

5.6.3 Integration testing: ... 47

Chapter 6 - CONCLUSION and FUTURE WORK 48
Chapter 7 - REFERENCES 50

List of Figures

Figure 1 – MapReduce Implementation Overview. 5
Figure 2 - MapReduce Execution Overview [1]. 6
Figure 3– HDFS Architecture [7]. 8
Figure 4– Hadoop Framework Stack [13]. 8
Figure 5– Hadoop Applications [14]. 9
Figure 6– Parts of MapReduce Job [2]. 10
Figure 7– A subset of Job Configuration parameters in Hadoop [4]. 11
Figure 8 – Response Surface of Terasort MapReduce program in 1D and 2D over 50GB dataset with varying parameters [4]. 12
Figure 9 – Response Surface of Terasort MapReduce program in 1D and 2D over 75GB dataset with varying parameters [4]. 12
Figure 10– Architecture of the Project. 18
Figure 11– Design of the optimizer. 20
Figure 12– Design of the contributor. 23
Figure 13 – Graph for varying maps in VSM. 25
Figure 14 – Graph for varying reducers in VSM 25
Figure 15 – Graph for varying io.sort.factor in VSM 26
Figure 16 – Graph for varying io.sort.record.percent in VSM 26
Figure 17 – Graph for varying mapreduce.task.io.sort.mb in VSM 27
Figure 18 – Graph for varying io.file.buffer.size in VSM 27
Figure 19 – Graph for varying maps in PageRank. 28
Figure 20 – Graph for varying reducers in PageRank 28
Figure 21 – Graph for varying io.sort.factor in PageRank 29
Figure 22 – Graph for varying io.sort.record.percent in PageRank 29
Figure 23 – Graph for varying mapreduce.task.io.sort.mb in PageRank 30
Figure 24 – Graph for varying io.file.buffer.size in PageRank 30
Figure 25 – Graph for varying maps in Word Count. 31
Figure 26 – Graph for varying reducers in Word Count. 31
Figure 27 – Graph for varying io.sort.factor in Word Count. 32

Figure 28 – Graph for varying io.sort.record.percent in Word Count 32
Figure 29 – Graph for varying mapreduce.task.io.sort.mb in Word Count 33
Figure 30 – Graph for varying io.file.buffer.size in Word Count 33
Figure 31 – Graph for varying maps in Sort 34
Figure 32– Graph for varying reducers in Sort 34
Figure 33 – Graph for varying io.sort.factor in Sort 35
Figure 34 – Graph for varying io.sort.record.percent in Sort 35
Figure 35– Graph for varying mapreduce.task.io.sort.mb in Sort 36
Figure 36 – Graph for varying io.file.buffer.size in Sort 36
Figure 37 – Anagram Execution time on Cranfield Data Set comparison graph. 38
Figure 38 – Anagram Execution time on NSF Data Set comparison graph. 40
Figure 39 – Grep Execution time on Cranfield Data Set comparison graph. 42
Figure 40 – Grep Execution time on NSF Data Set comparison graph. 43

List of Tables

Table 1 - Performance of Hadoop 50GB TeraSort when the mapred.reduce.tasks, io.sort.factor, and io.sort.record.percent parameters are varied [4]. 13
Table 2 – Job configuration parameters considered for the project along with their ranges. 17
Table 3 – Job configuration parameters considered for making an empirical study of parameter impact on MapReduce program performance. 24
Table 4 – System Configuration used for running the project. 24
Table 5 – Execution Times of Anagram MapReduce Program on Cranfield Data Set 39
Table 6 – Execution Times of Anagram MapReduce Program on NSF Data Set 40
Table 7 – Execution Times of Grep MapReduce Program on Cranfield Data Set 42
Table 8 – Execution Times of Grep MapReduce Program on Cranfield Data Set 44

Acknowledgements

I offer my sincere gratitude first to my advisor, **Dr. Mitchell L. Neilsen**, for his valuable suggestions and guidance to my research work. He has been a constant source of inspiration for me throughout my master's work. It has been an intellectually stimulating and rewarding experience for me to work with him. I truly feel privileged to have worked under him. I also thank the members of my graduate committee, **Dr. Daniel Andresen** and **Dr. Gurdip Singh**, for all their advice and encouragement.

I also like to thank **Dr. Doina Caragea** and **Dr. William Hsu** for giving me suggestions regarding the installation and algorithms implementing MapReduce.

I would like to thank my family members and friends for supporting and encouraging me to pursue this degree. Their encouragement was a key factor in successful completion of my degree.

Chapter 1 - INTRODUCTION

MapReduce is a Google's framework that provides Automatic parallelization and distribution, Fault-tolerance and I/O scheduling. Apache Hadoop is an Apache implementation of MapReduce framework that follows the design laid out in the original paper [1].

The main problem MapReduce addresses is processing large amount of data that requires lot of processing capability. MapReduce handles this problem by parallelizing and distributing the load over a number of commodity machines that are in clusters where the clusters are highly scalable. The other major property that MapReduce addresses is the automatic fault tolerance capability.

MapReduce is used for processing and generating large data sets. The Map function takes a key/ value pair as input and generates a pair of intermediate key/value pairs. These intermediate values are supplied to the Reduce function via an Iterator. The Reduce function then aggregates these values and produces the required output key/value pair. In the MapReduce paradigm, the program is subdivided into M splits and is supplied to several machines for concurrent execution. In this, one acts as a Master and the others are considered as Workers. The master assigns map and reduce jobs to the workers. After the task has been completed, output of the MapReduce is supplies as a set of R output files. The master is embedded with a data structure that maintains the state and identity of every worker machine. It also pings every worker periodically in order to check if there are any failures. If it finds any failures in any worker, it re schedules the task for an idle worker. Periodic checkpoints in the master data structure allow the execution to run smoothly even if the master fails. The program can be re- run from the previous check pointed state. Sometimes there might be errors caused while debugging and also due to bugs in records. Such records are skipped.

Many enterprises continuously collect large datasets that record customer interactions, product sales, results from advertising campaigns on the Web, and other types of information. Facebook collects 15 TeraBytes of data each day into its PetaByte-scale data warehouse [16].

Powerful telescopes in astronomy, particle accelerators in physics, and genome sequencers in biology are putting massive volumes of data into the hands of scientists. The ability to perform scalable and timely analytical processing of these datasets to extract useful information is now a critical ingredient for success. A number of enterprises use Hadoop MapReduce in production deployments for applications such as Web indexing, data mining, report generation, log file analysis, financial analysis, scientific simulation, and bioinformatics research. MapReduce frameworks are well suited to run on cloud computing platforms. For programs written in this model, the run-time system automatically parallelizes the processing across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and storage [4].

To run the program as a job in Hadoop, a job configuration object is created; and the parameters of the job are specified. Apart from the job configuration parameters, there are a large number of other parameters whose values have to be specified before the job can be run in a MapReduce framework like Hadoop. The settings of these parameters control various aspects of job behavior during execution such as memory allocation and usage, concurrency, I/O optimization, and network bandwidth usage. The submitter of a Hadoop job has the option to set these parameters either using a program-level interface or through XML configuration files. More than 190 parameters are specified to control the behavior of a MapReduce job in Hadoop. A fairly large subset of these parameters displays strong performance interactions with one or more other parameters. Even though the programming in MapReduce has emerged as a model for developing data-intensive processing applications, the configuration design-space of MapReduce has not been studied in detail [4]. A number of tuning parameters have to be set by users or system administrators to increase the efficiency of the MapReduce program. Users often run into performance problems because they are unaware of how to set these parameters, or because they don't even know that these parameters exist. With MapReduce being a relatively new technology, it is not easy to find qualified administrators. In this report, a framework that optimizes MapReduce programs that run on large datasets based on parameter combinations in proposed.

The proposed system will execute the given MapReduce program (Java Implementation) on a subset of the given dataset over a set of pre-executed parameter combinations and setting the best parameter combination to the input MapReduce program. Here the pre-executed parameter combinations are obtained by extracting the best two parameter combinations for each of the algorithms (Vector Space Model (indexing), PageRank (graph generation), Word Count, Distributed Word Sort (Sorting)) in the initial stage. The user can train the system to suit his domain by running the domain specific programs in the Contributor module of the proposed framework. We know that many MapReduce programs are used over and over again in applications like daily weather analysis, log analysis, daily report generation etc. So, once the parameter combination is set to the given MapReduce program, it can be used on number of data sets efficiently on daily basis. The working and implementation of the frame work is discussed in detail in the later chapters.

The rest of this paper first discusses background in chapter 2, and then discusses the related work in chapter 3. The implementation is described in detail in Chapter 4. Chapter 5 describes how the system is evaluated and presents the results. Chapter 6 presents our conclusions and describes future work.

Chapter 2 - BACKGROUND

2.1 HADOOP

In 2004, Google published papers describing their Google File System and MapReduce algorithms. Doug Cutting, a Yahoo employee and Open Source Evangelist, partnered with a friend to create Hadoop, an opensource implementation of GFS and MapReduce. Hadoop is a software system that could handle arbitrarily large amounts of data using a distributed file system, and distribute it to be worked on by an arbitrary number of workers, using MapReduce. Adding more storage or more workers is simply a matter of connecting new machines to the network—there is no need for larger devices or specialized disks or specialized networking.

There are many subprojects in Hadoop but the two main parts of Hadoop that are required for this project are:

1) MapReduce
2) HDFS

2.1.1 MapReduce

Hadoop supports the MapReduce model, which was introduced by Google as a method of solving problems involving large data processing with large clusters of inexpensive machines executing map/reduce jobs. In simple words, it is a programming model for processing huge volume of data sets over set of machines. MapReduce is the key algorithm that the Hadoop MapReduce engine uses to distribute work around a cluster. The other major property that MapReduce addresses is the automatic fault tolerance capability.

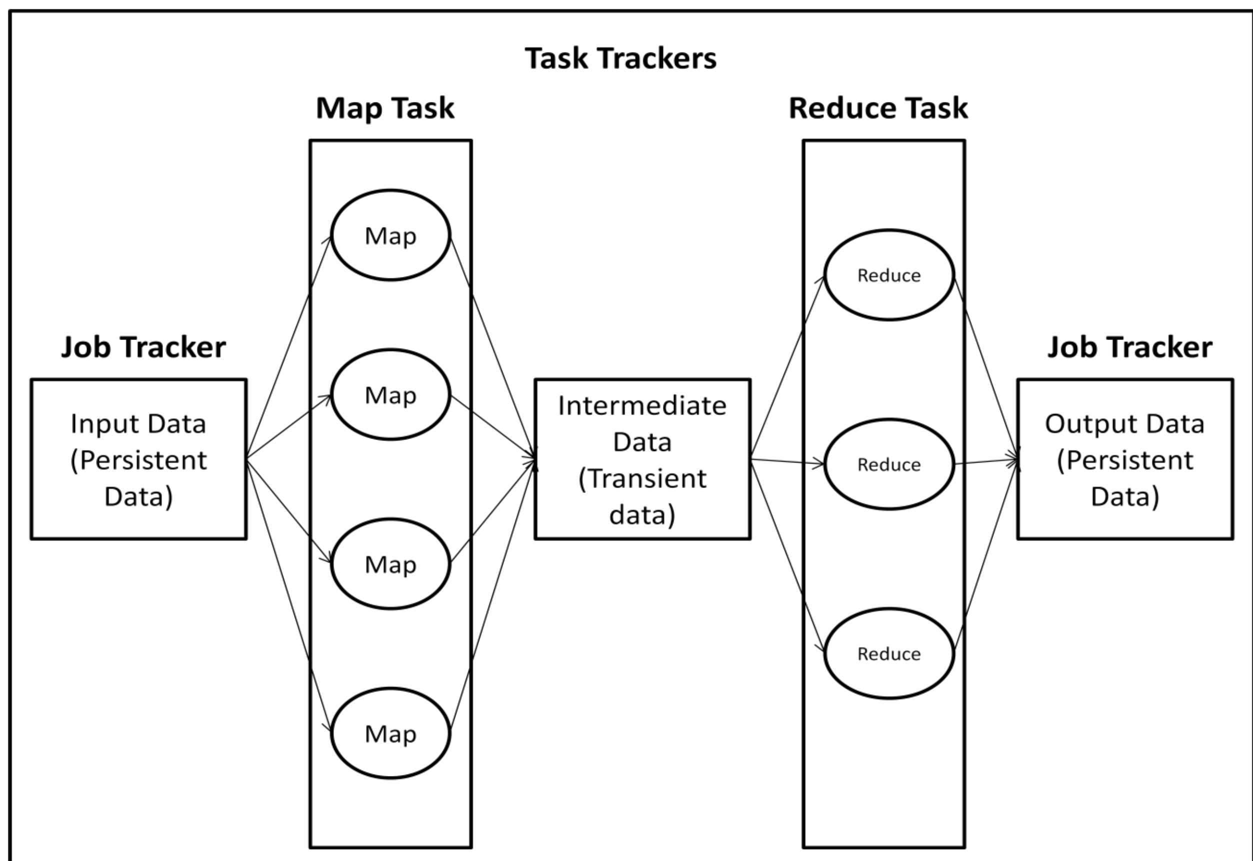
2.1.1.1 Implementation

The model is based on two distinct steps for an application:

- Map: An initial ingestion and transformation step, in which individual input records can be processed in parallel.
- Reduce: an aggregation or summarization step, in which all associated records, must be processed together by a single entity.

The core concept of MapReduce in Hadoop is that input may be split into logical chunks, and each chunk may be initially processed independently, by a map task. The results of these individual processing chunks can be physically partitioned into distinct sets, which are then sorted. Each sorted chunk is passed to a reduce task. The reducer executes on the sorted chunk of data and outputs the data.

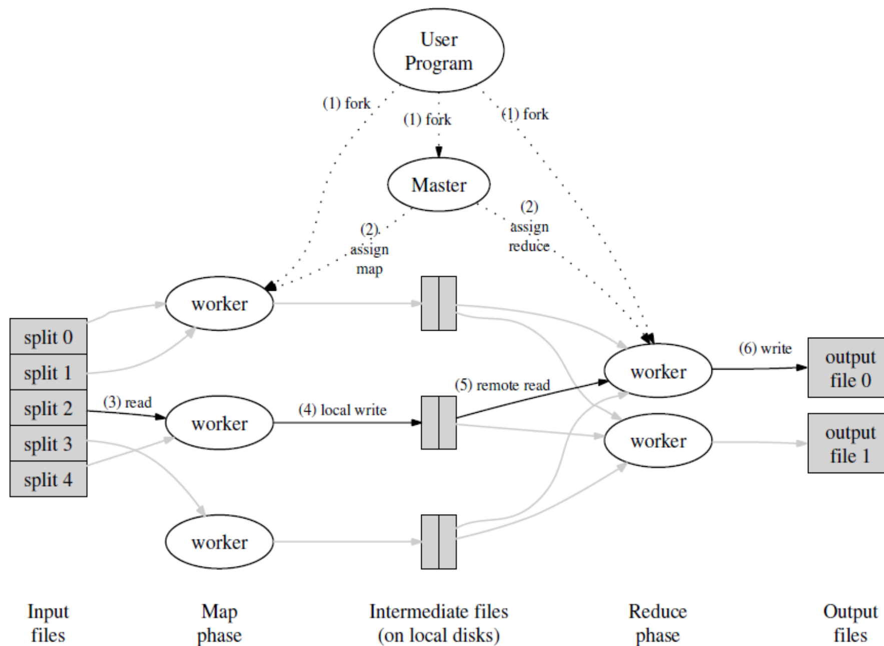
Figure 1 – MapReduce Implementation Overview.



2.1.1.2 Execution Overview

The Map function takes a key/ value pair as input and generates a pair of intermediate key/value pairs. These intermediate values are supplied to the Reduce function via an Iterator. The Reduce function then aggregates these values and produces the required output key/value pair. In the MapReduce paradigm, the program is subdivided into M splits and is supplied to several machines for concurrent execution. In this, one acts as a Master and the others are considered as Workers. The master assigns map and reduce jobs to the workers. After the task has been completed, output of the MapReduce is supplies as a set of R output files. The master is embedded with a data structure that maintains the state and identity of every worker machine. It also pings every worker periodically in order to check if there are any failures. If it finds any failures in any worker, it re schedules the task for an idle worker. Periodic checkpoints in the master data structure allow the execution to run smoothly even if the master fails. The program can be re- run from the previous check pointed state. Sometimes there might be errors caused while debugging and also due to bugs in records. Such records are skipped.

Figure 2 - MapReduce Execution Overview [1].



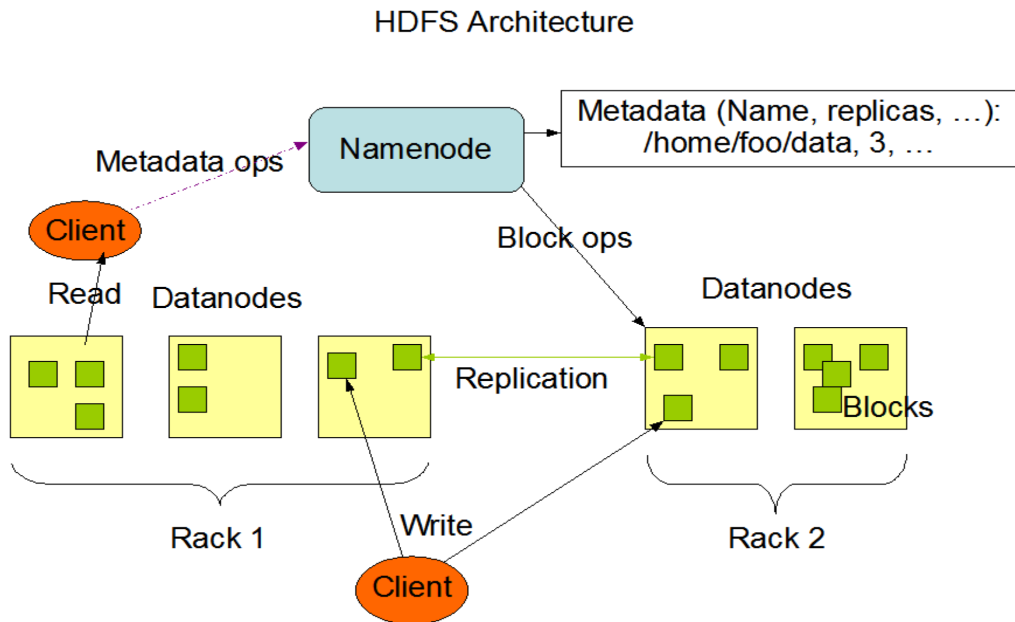
2.1.2 HDFS (Hadoop Distributed File System)

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets.

The architecture of HDFS is as simple as Master-Slave architecture, with the unique Master server, called as “Name Node” takes care of the file operations like opening, closing and renaming, and the slaves, called as “Data Node” are responsible for read and write operations over the files.

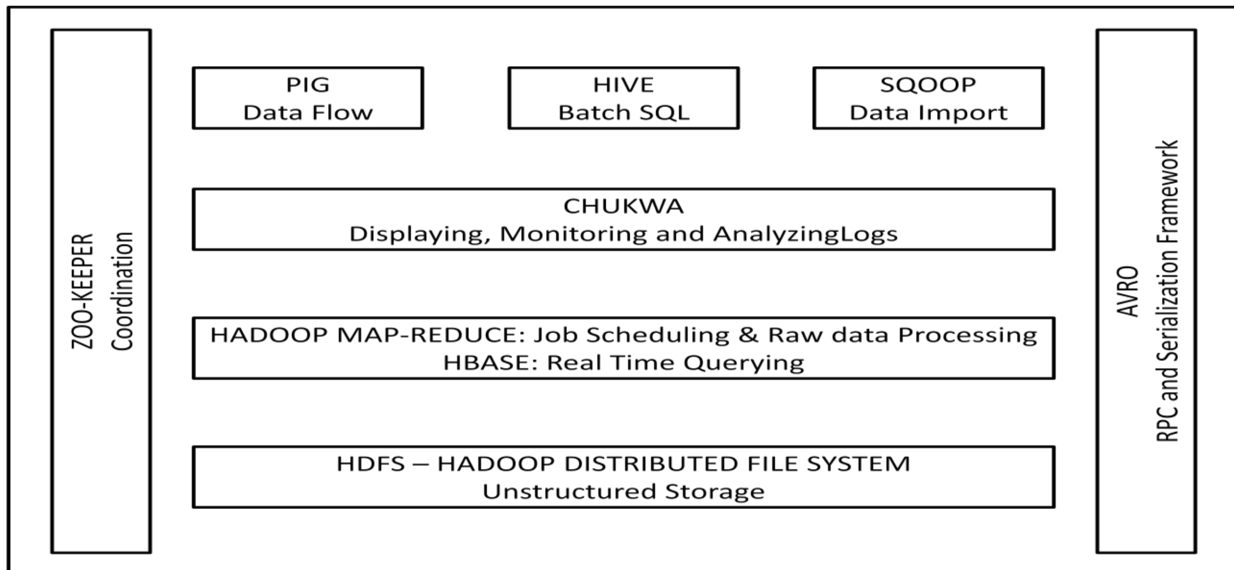
The Name Node is the single point failure in HDFS and because of this HDFS does not provide High Availability. The file system will be offline when the name node goes down and the transactions will be repeated once the server is live again. However the secondary name-node periodically takes a snap shot of the primary name-node and saves in remote system for restoring the primary name-node once it returns back from failure. The key to be noted here is that the secondary name-node does not act as a backup for the primary; instead it helps to restore the primary to its latest transaction before its failure point. The Data Nodes are the large group clients that communicate within themselves through Remote Procedure Call (RPC) to keep the replication of the files intact and to make sure the high availability of the files over multiple nodes in the cluster [13].

Figure 3– HDFS Architecture [7].



2.1.3 Hadoop Stack

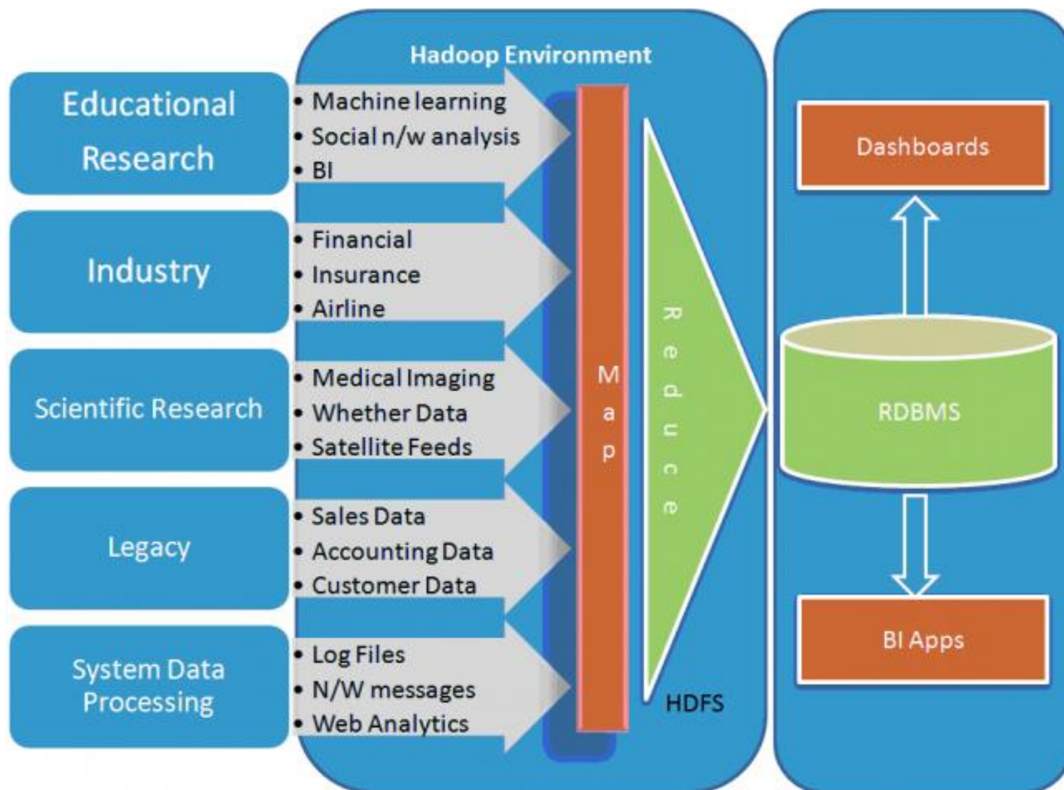
Figure 4– Hadoop Framework Stack [13].



2.1.4 Applications

There are many fields where Hadoop is being used. Some of the real time applications are given below in the figure. Hadoop processes and analyzes variety of new and older data to extract meaningful business operations intelligence.

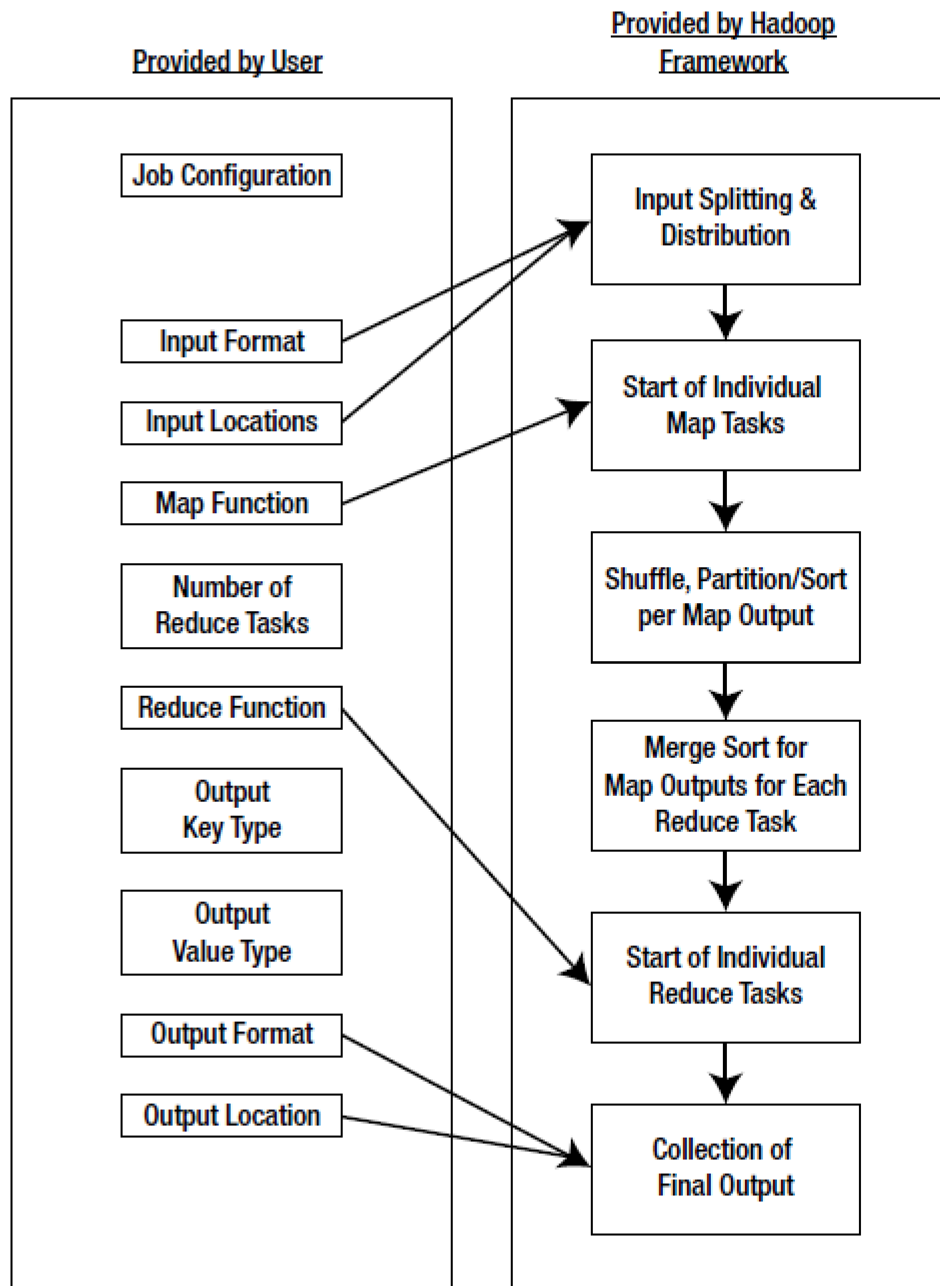
Figure 5– Hadoop Applications [14].



Chapter 3 - RELATED WORK

The MapReduce programs are generally executed in two stages. The user configures and submits a MapReduce job to the framework, which in turn will decompose the job into a set of map tasks; shuffles, a sort, and a set of reduce tasks. The framework manages the distribution, executions of the tasks, collects the output, and report the status to the user.

Figure 6– Parts of MapReduce Job [2].



The Job configuration parameters play a vital role in the performance of the MapReduce programs. To provide some empirical evidence to demonstrate differences in job running times between good and bad parameter settings in Hadoop, we take help from the paper "Towards Automatic Optimization of MapReduce Programs" by Shivnath Babu, Duke University.

The environment used for the experiment consisted of a Hadoop cluster running on 17 nodes, with 1 master and 16 worker nodes. Each node has a dual core 2GHz AMD processor, 1.8GB RAM, 30 GB of local space, and runs Linux in a Xen virtual machine. Each worker node was set to run at most 4 mapper tasks and 2 reducer tasks concurrently. Table below lists the subset of job configuration parameters that were considered in their experiments. The MapReduce program that was used for the experiment was Terasort.

Figure 7– A subset of Job Configuration parameters in Hadoop [4].

Parameter Name	Brief Description and Use	Default Value	Values Considered
mapred.reduce.tasks	Number of reducer tasks	1	[5,300]
io.sort.mb	Size in MegaBytes of map-side buffer for sorting key/value pairs	100	[100,200]
io.sort.record.percent	Fraction of io.sort.mb dedicated to metadata storage	0.05	[0.05,0.15]
io.sort.factor	Number of sorted streams to merge at once during sorting	10	[10,500]
io.file.buffer.size	Buffer size used to read/write (intermediate) sequence files	4K	32K
mapred.child.java.opts	Java control options for all mapper and reducer tasks	-Xmx200m	-Xmx[200m,300m]
mapred.job.shuffle.input.buffer.percent	% of reducer task's heap used to buffer map outputs during shuffle	0.7	0.7,0.8
mapred.job.shuffle.merge.percent	Usage threshold of mapred.job.shuffle.input.buffer.percent to trigger reduce-side merge in parallel with the copying of map outputs	0.66	0.66,0.8
mapred.inmem.merge.threshold	Another reduce-side trigger for in-memory merging; off when 0	1000	0
mapred.job.reduce.input.buffer.percent	% of reducer task's heap to buffer map outputs while applying reduce	0	0,0.8
dfs.replication	Block replication factor in Hadoop's HDFS filesystem	3	2
dfs.block.size	HDFS block size (data size processed per mapper task in our setting)	64MB	128MB

The experiments were conducted on two data sets, 50GB and 75GB. The data sets have been generated using TeraGen program from the TeraSort package. The three parameters, mapred.reduce.tasks, io.sort.record.percent and io.sort.factor, are varied in these figures while all other job configuration parameters are kept constant. When executed over the datasets, the following results were obtained.

Figure 8 – Response Surface of Terasort MapReduce program in 1D and 2D over 50GB dataset with varying parameters [4].

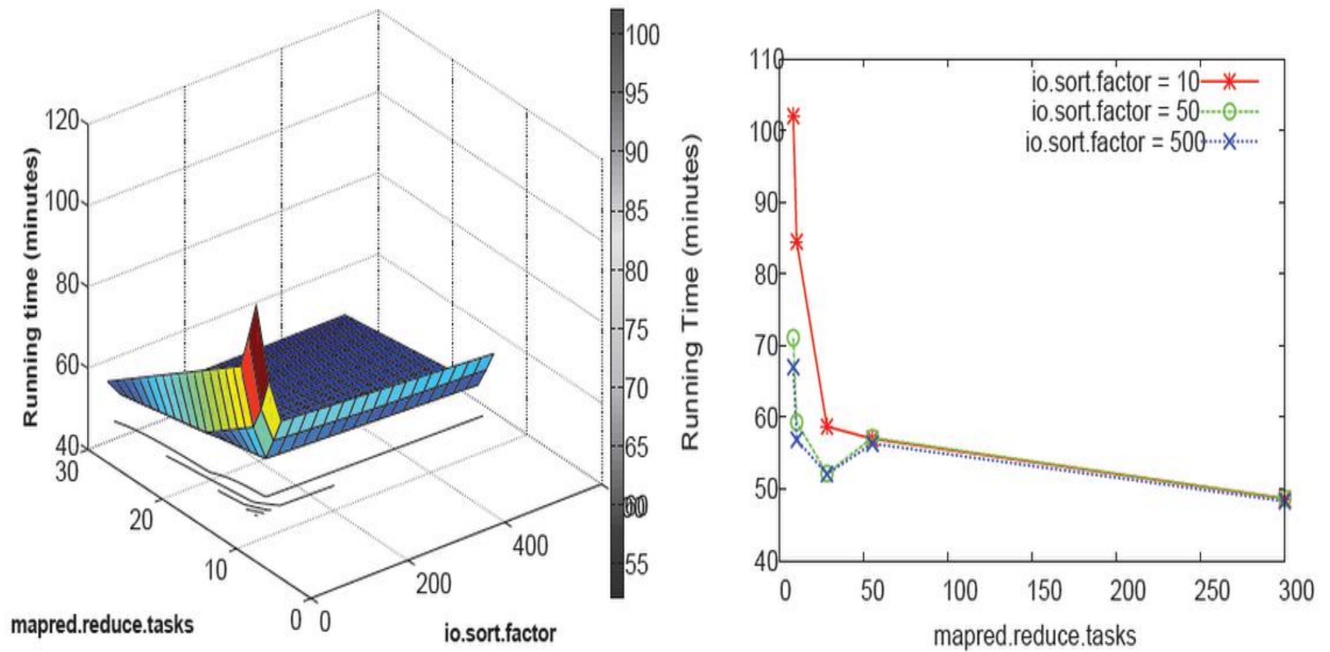


Figure 9 – Response Surface of Terasort MapReduce program in 1D and 2D over 75GB dataset with varying parameters [4].

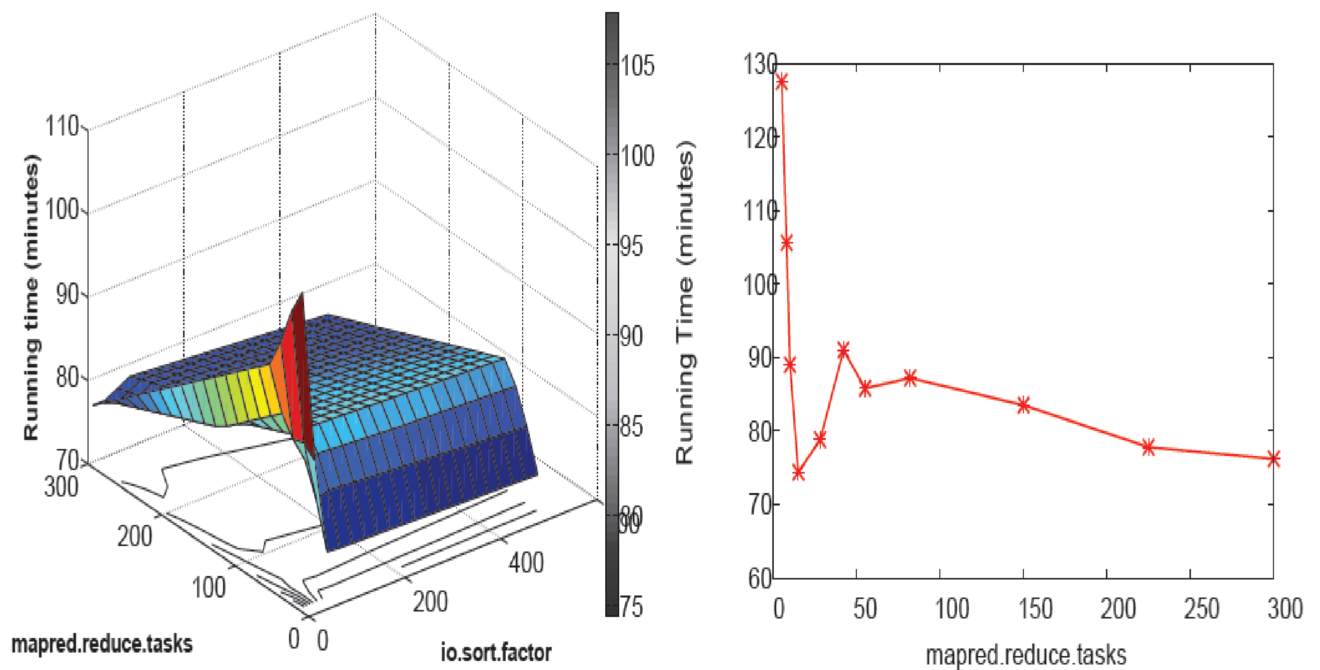


Table 1 - Performance of Hadoop 50GB TeraSort when the mapred.reduce.tasks, io.sort.factor, and io.sort.record.percent parameters are varied [4].

Row#	mapred. reduce.tasks	io.sort. factor	io.sort.record. percent	Job Running Time
1	10	10	0.10	1hr, 25mins, 25sec
2	10	10	0.15	1hr, 14mins, 54sec
3	10	500	0.10	1hr, 7mins, 11sec
4	10	500	0.15	1hr, 1mins, 1sec
5	28	10	0.10	1hr, 22mins, 54sec
6	28	10	0.15	1hr, 4mins, 57sec
7	28	500	0.10	1hr, 22mins, 24sec
8	28	500	0.15	1hr, 3mins, 46sec
9	300	10	0.10	45mins, 22sec
10	300	10	0.15	35mins, 9sec
11	300	500	0.10	44mins, 38sec
12	300	500	0.15	35mins, 56sec

It is evident from the above results that the row#10 gets the best results,

The empirical evidence from above results suggests that the performance of a MapReduce job *J* is some complex function of the job configuration parameter settings. In addition, the properties of the data as well as the resources allocated to the job in the cluster will impact its performance. There exists some function F_J such that:

$$y = F_J(\vec{p} \in \vec{P}, \vec{r} \in \vec{R}, \vec{d} \in \vec{D}) \quad \text{-- Eq(1)}$$

Here, y represents a performance metric of interest for *J* (e.g., J's running time). pˆ represents the setting of the job configuration parameters for a run of J. rˆ represents the resources allocated for the run, and dˆ represents the statistical properties of the data processed

by J during the run. The response surfaces shown in Figure 8 and Figure 9 are partial projections of Eq(1) for the TeraSort MapReduce program when run on our cluster.

The major problem here is when given an input dataset and resource allocation for running a MapReduce job, we can think of pˆ which represents the setting of the job configuration parameters for a run of *J* - as specifying an execution plan for J. Different choices of the execution plan give rise to potentially different values for the performance metric of interest for J. The problem addressed in the paper is to automatically and efficiently choose a good execution plan for J given an input dataset and resource allocation. For this they suggested a few approaches in the paper. The following is the brief description of the approaches [4].

Database query-optimizer style approach:

Query Optimizers in database management systems also focus on identifying an efficient execution plan for declarative SQL queries. This is usually achieved by query optimizers by maintaining statistics about input data provided and using them in conjunction with cost models for cost estimation of these plans. These execution plans have operators such as index scan, sort, hash join, etc. that represent various performance metrics of interest for the SQL query. A good execution plan that is supported by the database's search engine is identified using a search algorithm. This approach is extended to Map Reduce paradigm. The disadvantage of this approach is that the complete MapReduce functionality is written in languages such as java, c++, python that are not restrictive or declarative like SQL. Next the lack of Statistics about the input data before hand and finally, it is tedious to translate algorithms from SQL query optimizers for optimizing Map Reduce programs as the execution plan space for SQL is different as compared to parameters for Map Reduce Programming.

Dynamic Profiling:

Missing data required for optimization can be gathered before the execution of the actual job to overcome the issues faced in the above approach. The idea of Dynamic Profiling can be applied to parameter configuration of MapReduce jobs. But this in turn has its own drawbacks:

- An assumption made for sampling based partitioning is that data type and distribution of key in the input data does not differ from the data generated by the map function which may not hold for all Map Reduce Programs

- Certain restrictions are placed on how the input data is accessed in the Map reduce paradigm whereas sample sort and probabilistic splitting assume that input data can be assembled by sampling at various levels

Designing the perfect sampling method for dynamic profiling is the crucial tasks in this approach as there is always a chance of generating deviating estimates that degrade the performance. Hence a sampling technique should be adopted that gives highly desirable probabilistic error guarantees.

Reacting Through Late Binding:

The dependence on cost models can be reduced through late binding approaches that delay the setting of one or more parameters until after some part of the execution has been observed. This approach can be applied for parameters whose value can either be changed during job execution, or does not have to be set until part of the execution is complete. Hadoop uses a late-binding approach to decide whether to partially aggregate the output of a mapper task. The concept of speculative execution can be useful here if some part of the execution can be done and then undone efficiently, then reactive setting can be applied to a larger class of parameters while keeping the performance impact low. However, there is a risk of thrashing. Currently, MapReduce frameworks have support for speculative execution only at the granularity of full mapper and reducer tasks (which is aimed at lowering the chances of some slow or failed tasks increasing the overall job completion time).

Competition-based Approaches:

The idea here is (i) to start multiple instances of the same task concurrently, with each instance having a different setting of the job configuration parameters; (ii) to quickly identify the best instance; and (iii) to kill all the other instances. Similar ideas have been used for adaptive processing of queries in database systems. The speculative execution feature of MapReduce frameworks is potentially useful here. Currently, the speculative instances are run with the same parameter configuration as the main instance. While the reactive and competitive approaches reduce or eliminate the need for cost models, the reality is more subtle. The problem with reactive and competitive approaches is that they have to predict the performance of an entire job run by observing only an initial part of the execution [4].

Chapter 4 - IMPLEMENTATION

We have seen in the previous section how the job configuration parameters impact the performance of a MapReduce program. Here we implement a project that is aimed at optimizing the MapReduce program performance by setting the parameters that would make the program execute in less or equal amount of time than the default job configuration parameter setting.

The first step in the project is to select the job configuration parameters and see how each parameter is affecting the performance of the MapReduce program. The configuration parameters that were considered are:

1) Number of mappers
2) Number of reducers
3) mapreduce.task.io.sort.factor
4) mapreduce.task.io.sort.mb
5) io.file.buffer.size
6) io.sort.record.percent

For testing the parameter impact we have considered four algorithms which are belonging to different applications, namely

1) Vector Space Model (Indexing)
2) PageRank (Graph)
3) Word Count (Text Processing)
4) Distributed Sort (Sorting)

Each of the algorithms is executed over parameter combinations in which only one parameter is varied each time and the rest of parameter combinations are left to default value. The results and evaluation of these executions are dealt in the next chapter.

The idea behind the project is to gather the parameter combination that would give a better or equal execution time compared to the default combination. For this, I considered six above mentioned parameters with ranges specified in figure below.

Table 2 – Job configuration parameters considered for the project along with their ranges.

Parameter Name	*Description & Use*	*Default*	*Considered Range*
mapred.map.tasks	Number of Mappers	2	[2, 5,10,25,50]
mapred.reduce.tasks	Number of Reducers	1	[1, 5,10,25,50]
mapreduce.task.io.sort.factor	Number of sorted streams to merge at once during sorting	10	[10, 20, 50]
mapreduce.task.io.sort.mb	Size in MegaBytes of map-side buffer for sorting key/value pairs	100	[100, 150, 200]
io.file.buffer.size	Buffer size used to read/write (intermediate) sequence files	4K	[4K,8K]
io.sort.record.percent	Fraction of io.sort.mb dedicated to metadata storage	0.05	[0.05, 0.10, 0.15]

This project consists of **two** parts:

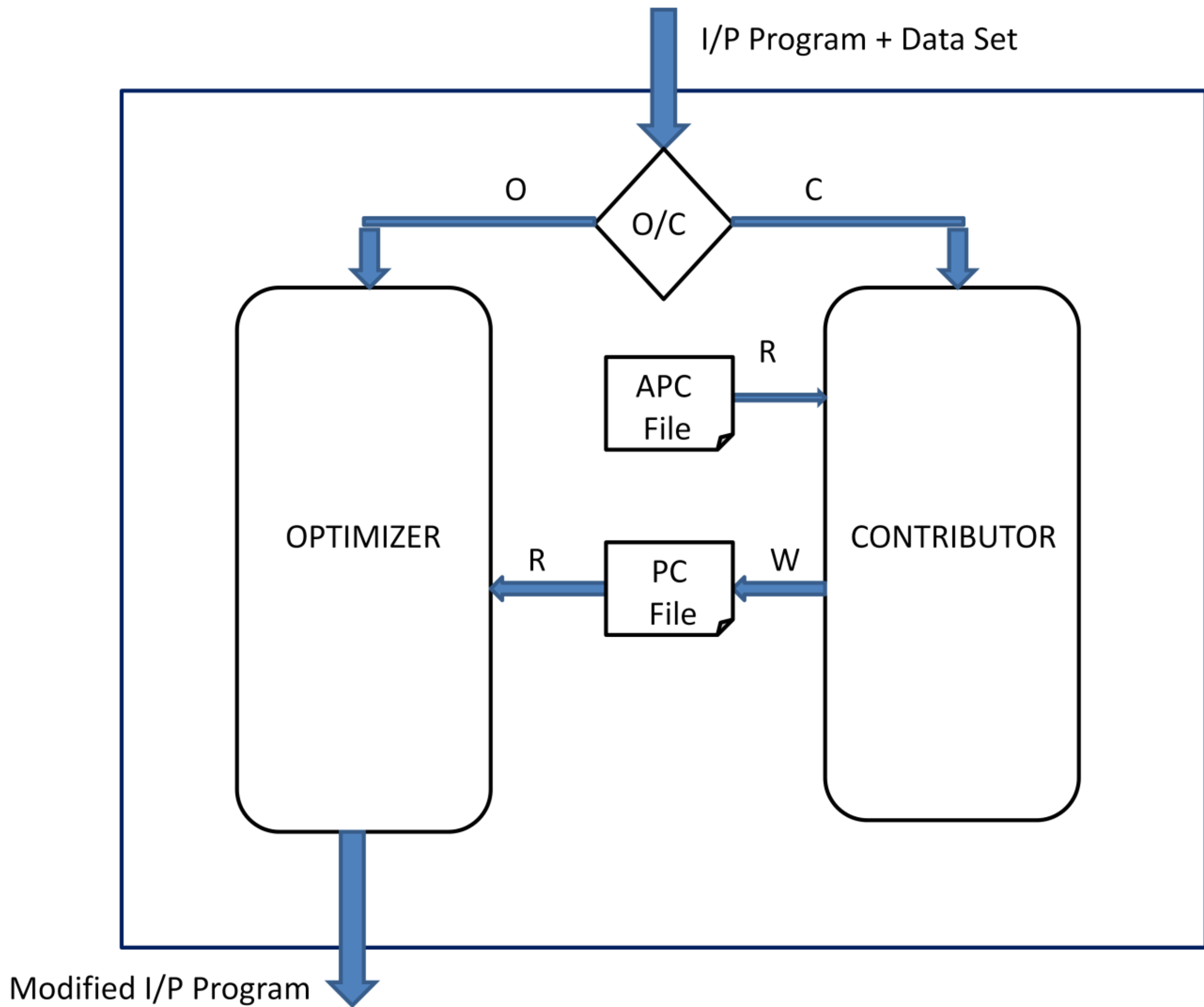
- **Optimizer** – optimizes the MapReduce Program by setting job configuration parameters
- **Contributor** – helps to train the project to suit the domain.

Before going into details about the parts, first we need to do some preliminary executions to help the optimizer to be ready for different kind of applications.

First, I generated all possible combinations of parameters combinations within the specified ranges and stored in a file. This file is called allParameterCombinationsFile. The four above mentioned algorithms are individually executed iteratively for each of the parameter combination in the allParameterCombinationFile. The execution times for each algorithm are sorted and the two best parameter ranges of each of the algorithm are stored in a file. This file is called the parameterCombinationFile. This file has a fixed number of parameter combinations. For this project, eight combinations is the fixed limit. This is limitation is incorporated to ensure that the optimizer doesn't have to run more than eight time so as to see that this optimizer

overhead is not dampening the overall efficiency of the project. The file is modified by the contributor if the user wants to train the system. This file will be used by the optimizer. Below is the overall diagram of the system.

Figure 10– Architecture of the Project.



C	-	CONTRIBUTOR
O	-	OPTIMIZER
W	-	WRITES TO
R	-	READS FROM
APC	-	ALL PARAMETER COMBINATIONS
PC	-	PARAMETER COMBINATIONS
I/P	-	INPUT

Let us see how the project works with detailed description of the two parts.

4.1 Optimizer

The optimizer module as its name suggests takes care about optimizing the MapReduce program. The working in the optimizer module is split up into three phases,

1) Execution
2) Sorting
3) Setting

In the execution phase the module takes the input MapReduce program and data set (partial) provided by the user and executes it over each and every parameter combination from the parameterCombinationFile that was obtained in the preliminary executions of the algorithms or through training from the contributor.

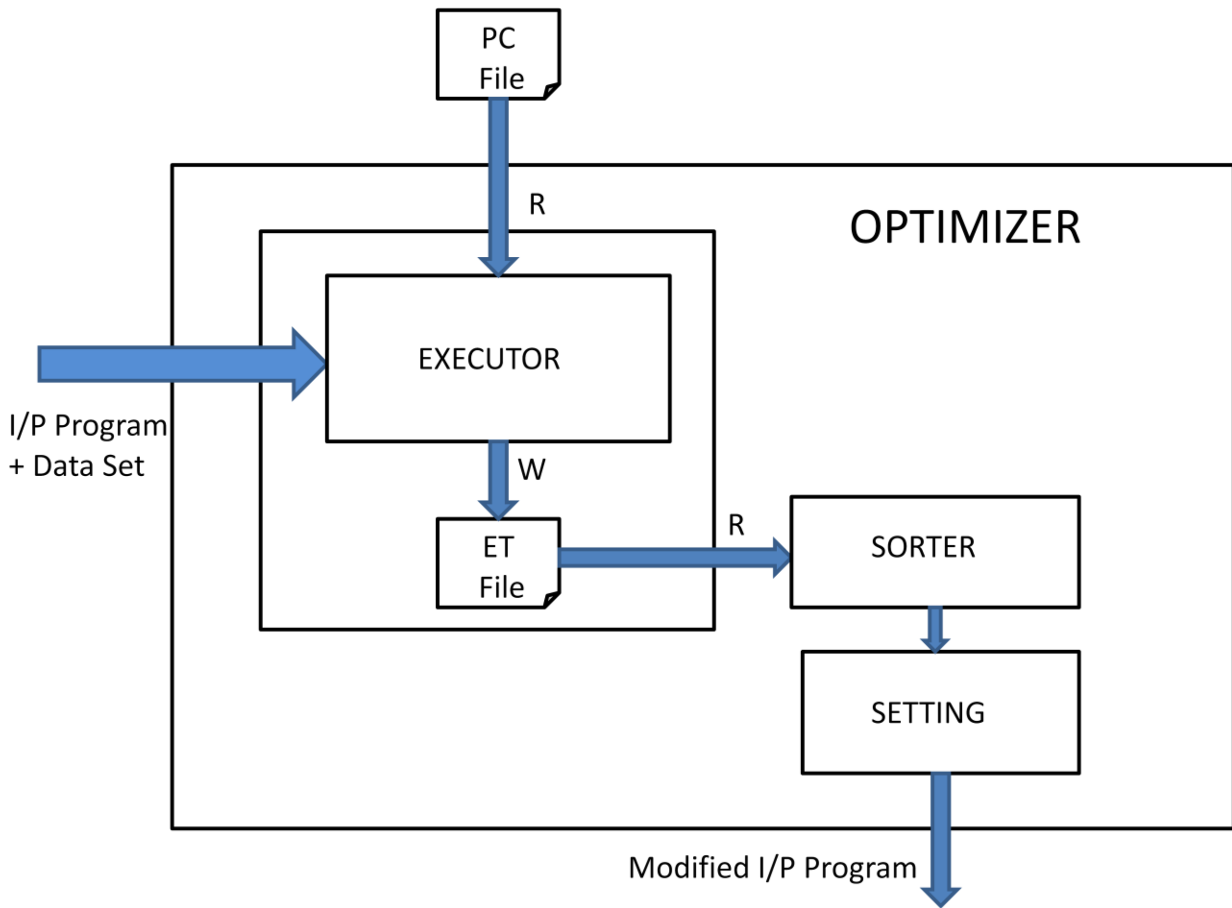
The execution times of each of the parameter combinations are stored in a file. Each parameter combination and its corresponding line number are stored in a hashtable where key is the line number and the value is the parameter combination. The execution times file is forwarded to the sorting phase after finishing the execution of all the parameter combinations.

In the sorting phase, the execution times file is read for execution time of each parameter combination. The minimum of the execution times is obtained and the corresponding line number for minimum execution in other words, the efficient execution is obtained. This line number is forwarded to the setting phase.

In the setting phase, the line number of the efficient execution time is used to get the corresponding parameter combination which is stored in the hashtable. This combination is set to the input program and the program is outputted.

Now, after setting the parameter combination to the input program, this program can be used to execute over a large dataset. The results of this can be seen in the next chapter. Below is the optimizer module diagram.

Figure 11– Design of the optimizer.



W	-	WRITES TO
R	-	READS FROM
PC	-	PARAMETER COMBINATIONS
I/P	-	INPUT
ET	-	EXECUTION TIME

4.2 Contributor

The contributor part in this project helps to train the project towards the domain. As mentioned earlier, many MapReduce programs are used over and over again in applications like daily weather analysis, log analysis, daily report generation etc. So, if we make the optimizer more suitable to the domain, the more efficiently the parameters can be set to the MapReduce parameters. This can be attained by executing few domain specific programs on the contributor. Let us see in detail how the contributor works.

The contributor is split up into three modules. They are

1) Execution
2) Sorting
3) Replacement

In the execution phase the given input MapReduce program and data set is executed over all the parameter combinations from the allParemeterCombinationsFile. Each parameter combination and its corresponding line number are stored in a hashtable where key is the line number and the value is the parameter combination. The execution times file is forwarded to the sorting phase after finishing the execution of all the parameter combinations.

In the sorting phase, the execution times file is read for execution time of each parameter combination. The minimum of the execution times is obtained and the corresponding line number for minimum execution in other words, the efficient execution is obtained. This line number is forwarded to the replacement phase.

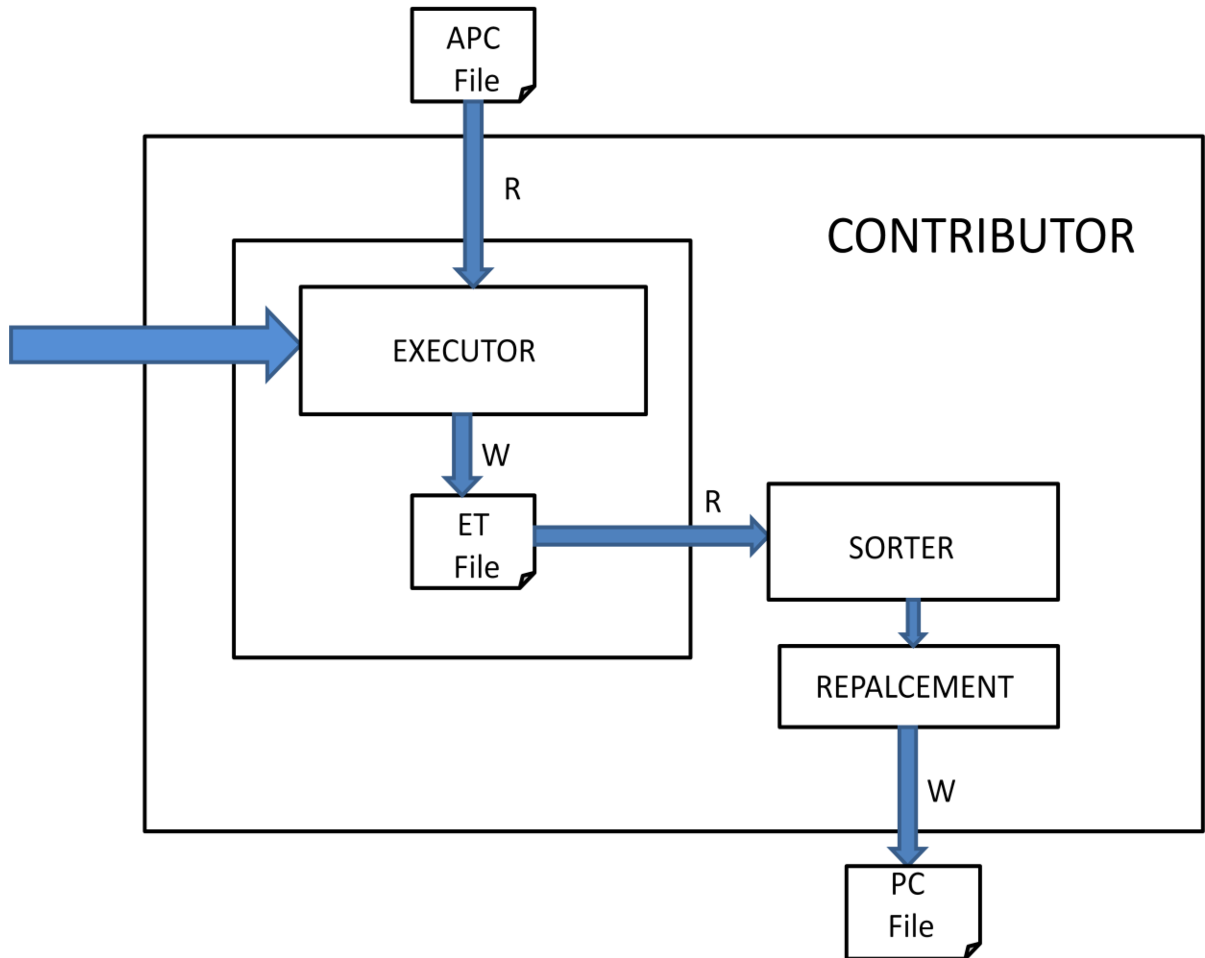
Here in the replacement phase, the line number of the efficient execution time is used to get the corresponding parameter combination which is stored in the hashtable. This combination is written in the parameterCombinationFile used in the optimizer by replacing one of the parameter combinations.

For the replacement of parameter combination in the parameterCombinationFile, many algorithms are considered. The first one was the LRU (Least Recently Used). For this whenever the optimizer uses the parameterCombinationFile and sets a parameter combination to an input MapReduce program, that particular combination is shifted to the top of the file. Now, if a contributor is run with an algorithm, then the last parameter combination will be replaced by the new combination from the contributor. The problem arises when we run the contributor consecutively. Here when we run consecutively, the last parameter combination (which has been replaced by contributor before) in the parameterCombinationFile is replaced with the new one from contributor thereby making the previous combination from the contributor deleted. So, the main objective of the contributor, i .e, to make the project more domain specific fails.

To avoid the drawback, the Least Frequently Used replacement algorithm is considered. Every parameter combination in the parameterCombinationFile is given a counter. Whenever the optimizer sets a particular combination, that combination's counter is increased. Now, if a contributor is run with an algorithm, then the parameter combination with least count will be replaced by the new combination from the contributor. The problem arises when we run the contributor consecutively. Here when we run consecutively, the parameter combination with least count in the parameterCombinationFile is replaced with the new one from contributor thereby making the previous combination from the contributor deleted because it hasn't been used once. So, the count will be zero and it would be replaced. Even this method fails.

So, to stick to the objective of the contributor, i. e, to make the project more domain specific, another hybrid method has been considered. This is a mixture of the above two methods would be suggested. Every parameter combination in the parameterCombinationFile is given a counter. Whenever the optimizer sets a particular combination, that combination's counter is increased. Here whenever the contributor is run with an algorithm, then the new parameter combination is put above the parameter combinations that are not used before. So, it would replace a parameter combination that is either lesser count or equal count. Even this has limitations, but lesser than LRU or LFU methods.

Figure 12– Design of the contributor.



W - WRITES TO
R - READS FROM
PC - PARAMETER COMBINATIONS
APC - ALL PARAMETER COMBINATIONS
I/P - INPUT
ET - EXECUTION TIME

Chapter 5 - RESULTS AND EVALUATION

The four algorithms mentioned earlier are executed with one varying parameter and rest others are kept to default values. The parameter ranges for this experiment are listed in the figure below.

Table 3 – Job configuration parameters considered for making an empirical study of parameter impact on MapReduce program performance.

Parameter Name	*Description & Use*	*Default*	*Considered Range*
mapred.map.tasks	Number of Mappers	2	[2, 5, 10, 15, 25, 50, 75, 100, 150, 200]
mapred.reduce.tasks	Number of Reducers	1	[1, 5,10, 15, 25,50, 75, 100, 150, 200]
mapreduce.task.io.sort.factor	Number of sorted streams to merge at once during sorting	10	[10, 20, 50, 100, 150, 200]
mapreduce.task.io.sort.mb	Size in MegaBytes of map-side buffer for sorting key/value pairs	100	[100, 125,150, 175, 200]
io.file.buffer.size	Buffer size used to read/write (intermediate) sequence files	4K	[4K,8K,16K,32K]
io.sort.record.percent	Fraction of io.sort.mb dedicated to metadata storage	0.05	[0.05, 0.075, 0.10, 0.125, 0.15]

Table 4 – System Configuration used for running the project.

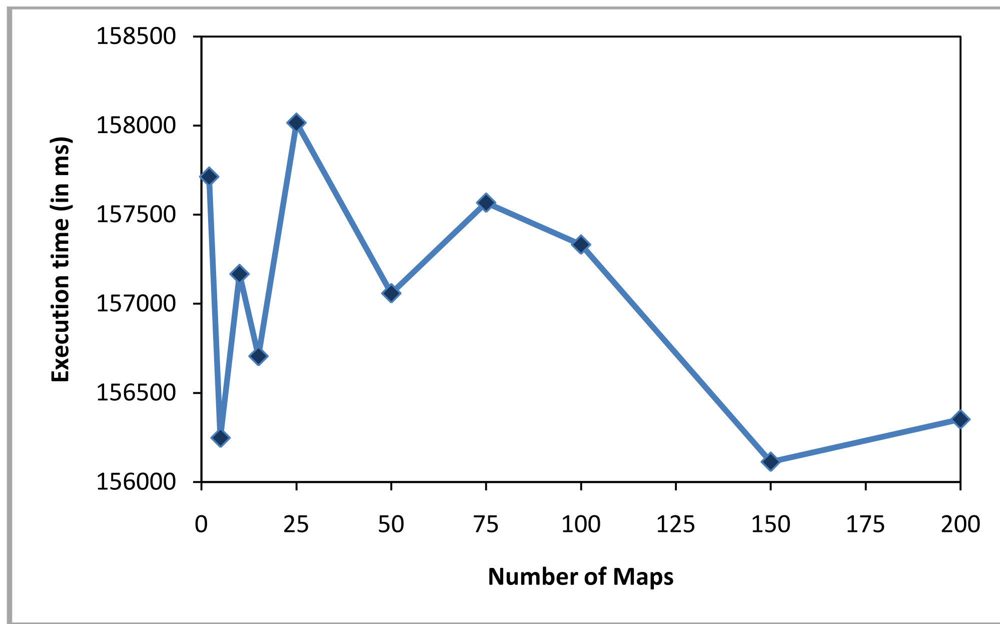
Operating System	Microsoft Windows XP
Number of Cores	2 (Intel(R) Core(TM) 2 Duo)
RAM	3.21 GB
Speed	3.16 Ghz
Hard Disk	297 GB
Hadoop Version	Hadoop-0.20.2
Eclipse	Eclipse- Helios

5.1 Vector Space Model Evaluation

For the experiment, we have considered the Cranfield Data Set (5.48 MB). The preprocessing and indexing is done using MapReduce. The results of varying parameters are as follows.

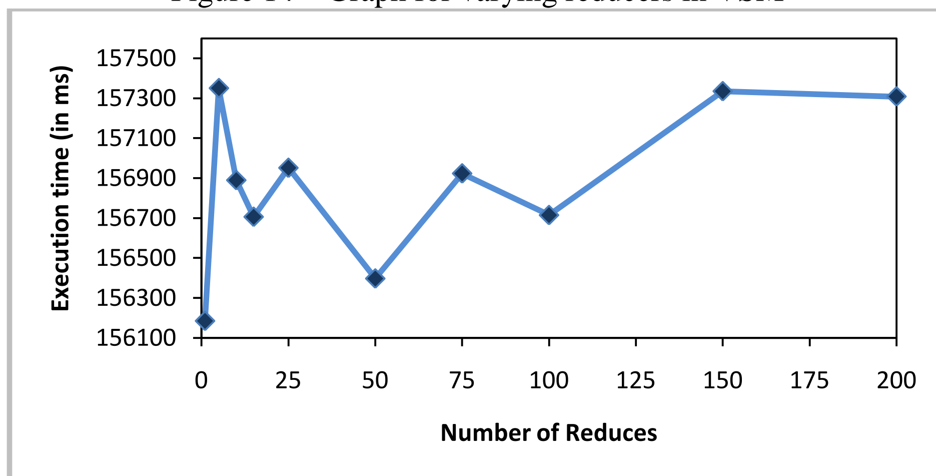
Varying Number of mappers:

Figure 13 – Graph for varying maps in VSM.



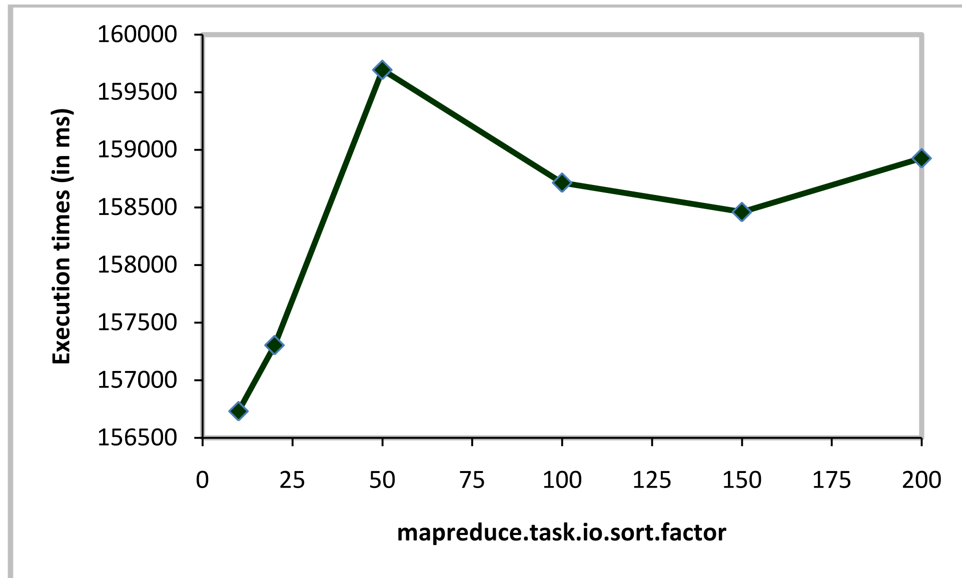
Varying Number of Reducers:

Figure 14 – Graph for varying reducers in VSM



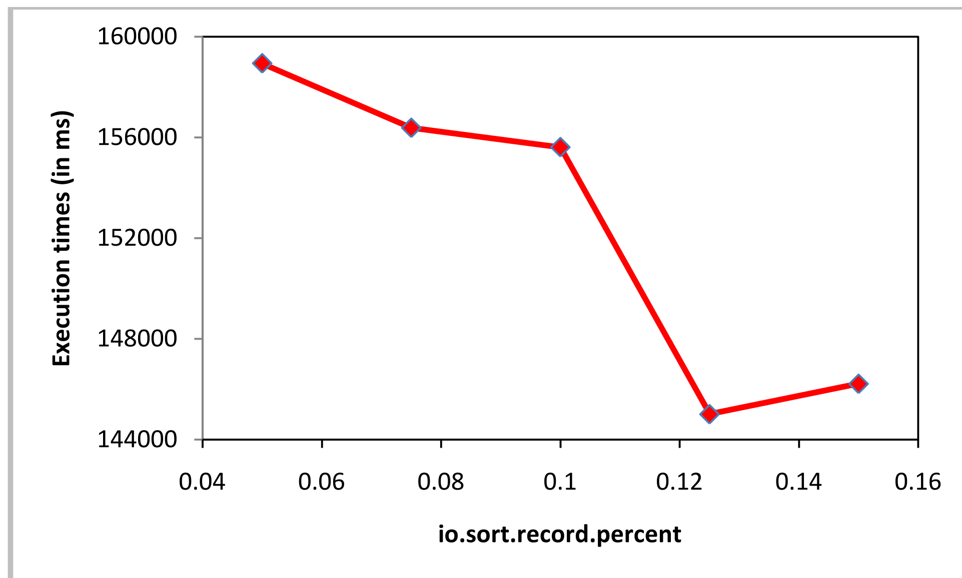
io.sort.factor:

Figure 15 – Graph for varying io.sort.factor in VSM



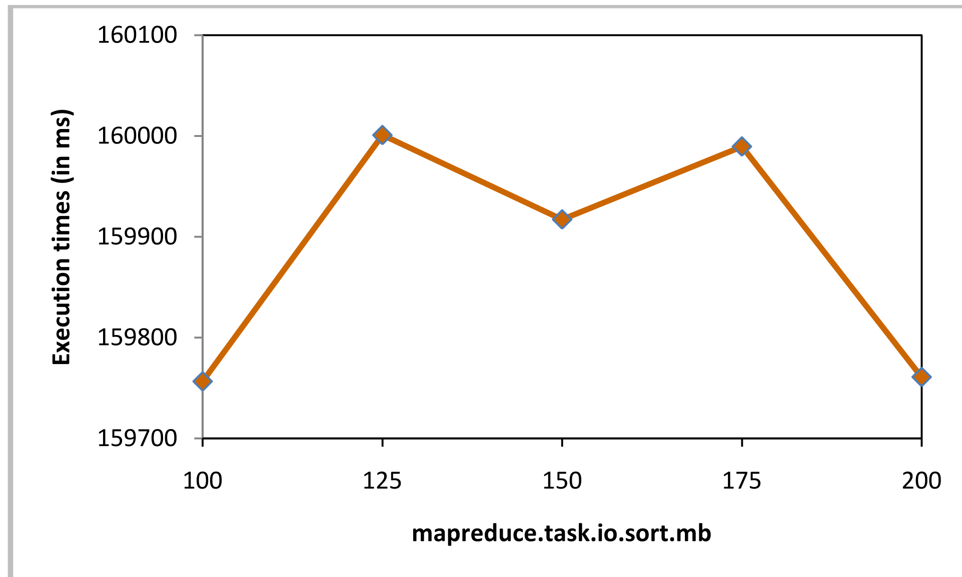
io.sort.record.percent:

Figure 16 – Graph for varying io.sort.record.percent in VSM



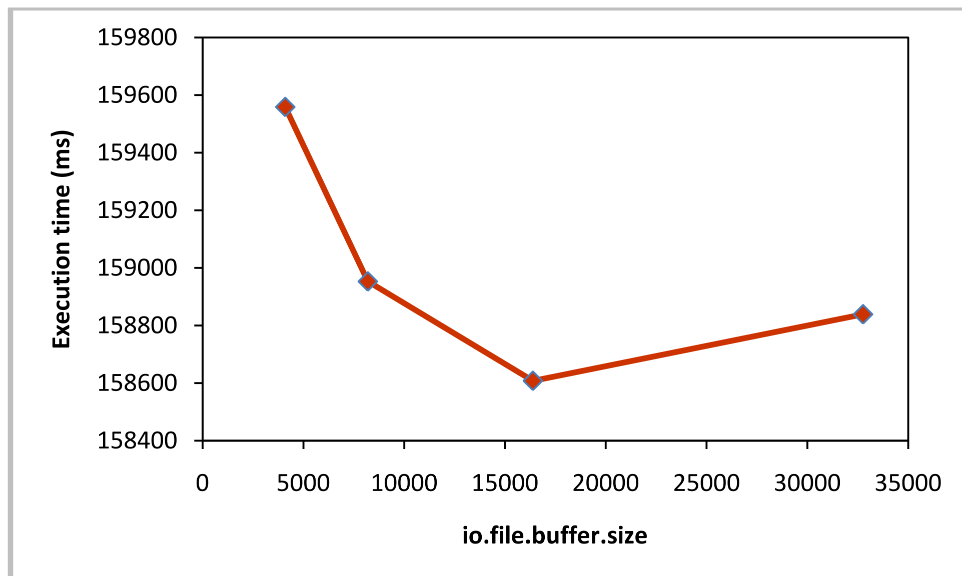
mapreduce.task.io.sort.mb:

Figure 17 – Graph for varying mapreduce.task.io.sort.mb in VSM



io.file.buffer.size:

Figure 18 – Graph for varying io.file.buffer.size in VSM

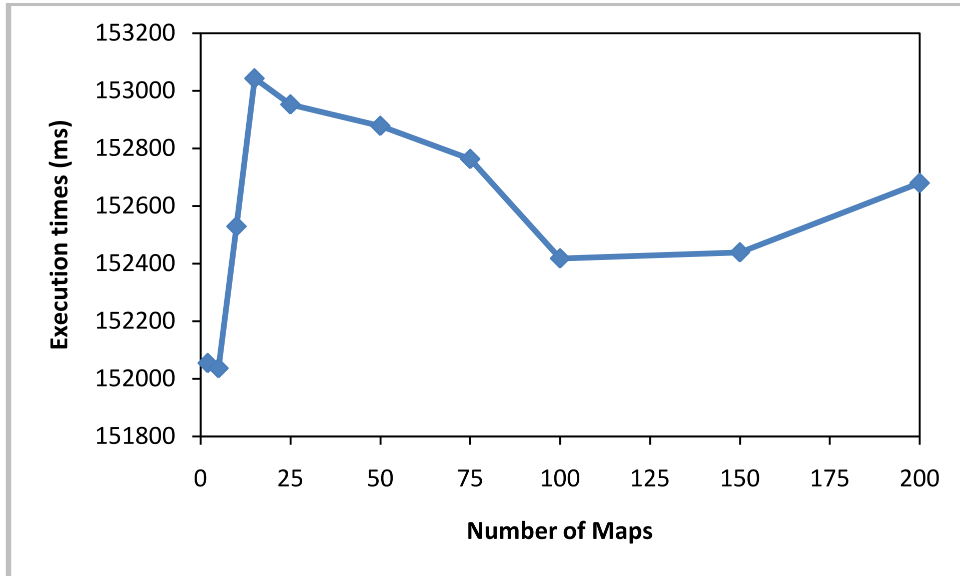


5.2 PageRank Evaluation

For the experiment, we have considered the afwiki data set (65 MB). The link graph generation and page rank calculation is done using MapReduce. The results of varying parameters are as follows.

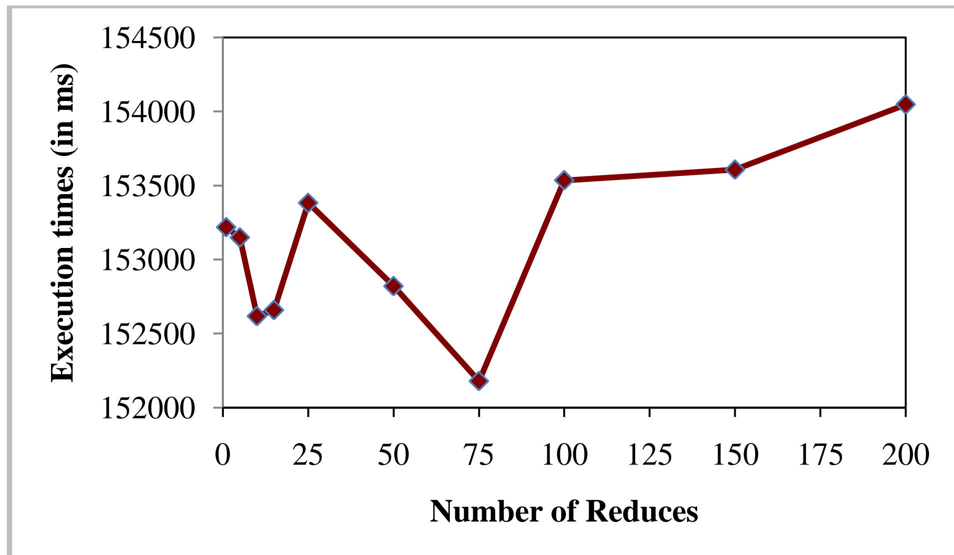
Varying Number of mappers:

Figure 19 – Graph for varying maps in PageRank.



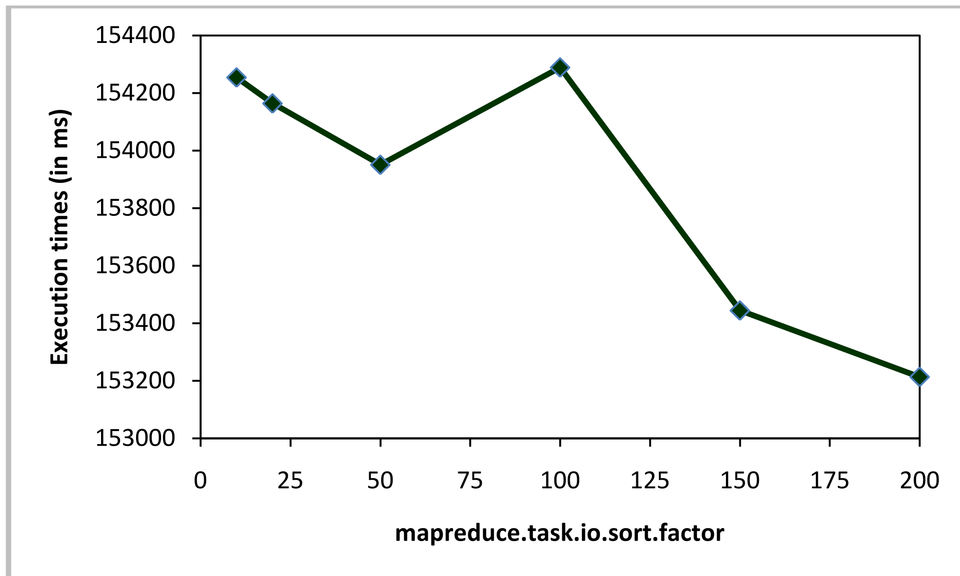
Varying Number of reducers:

Figure 20 – Graph for varying reducers in PageRank



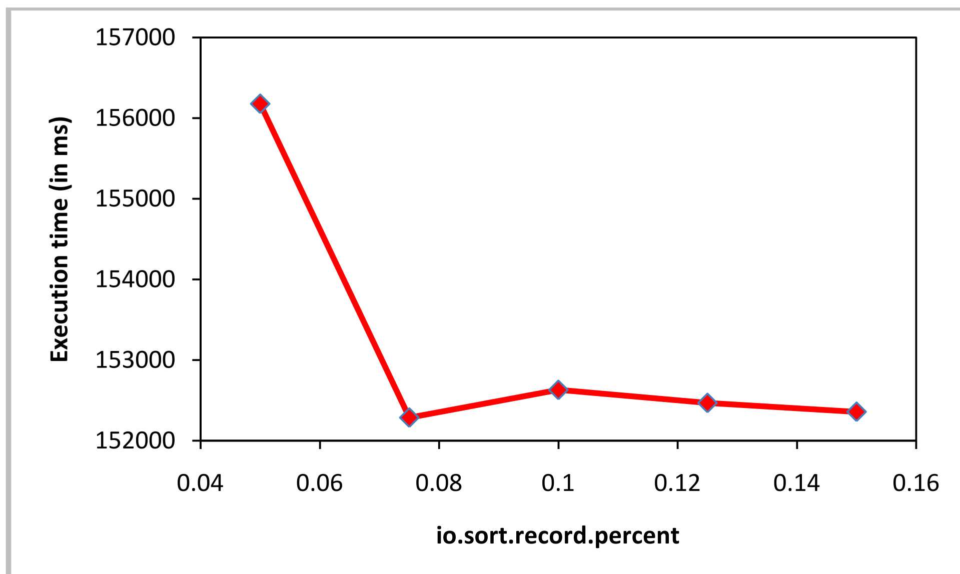
io.sort.factor:

Figure 21 – Graph for varying io.sort.factor in PageRank



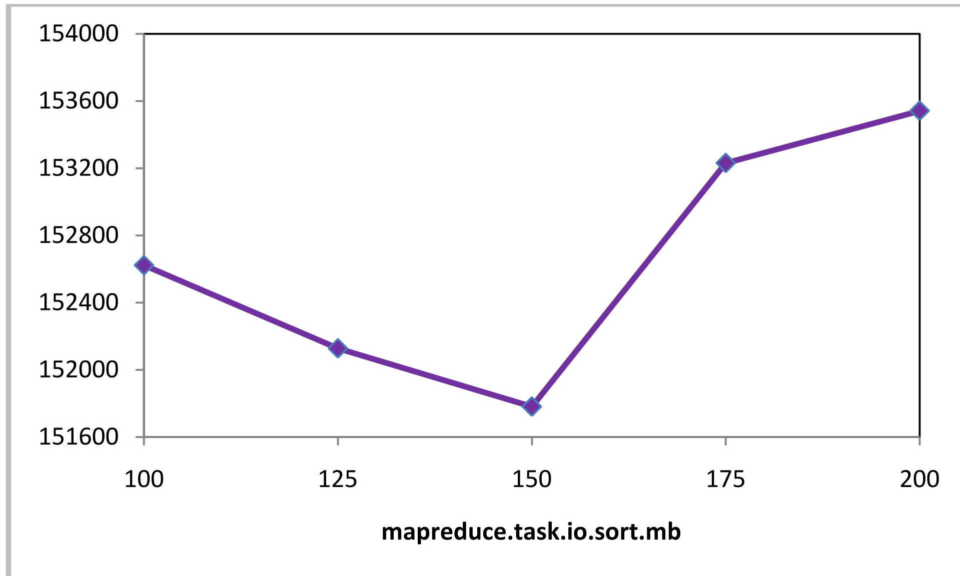
io.sort.record.percent:

Figure 22 – Graph for varying io.sort.record.percent in PageRank



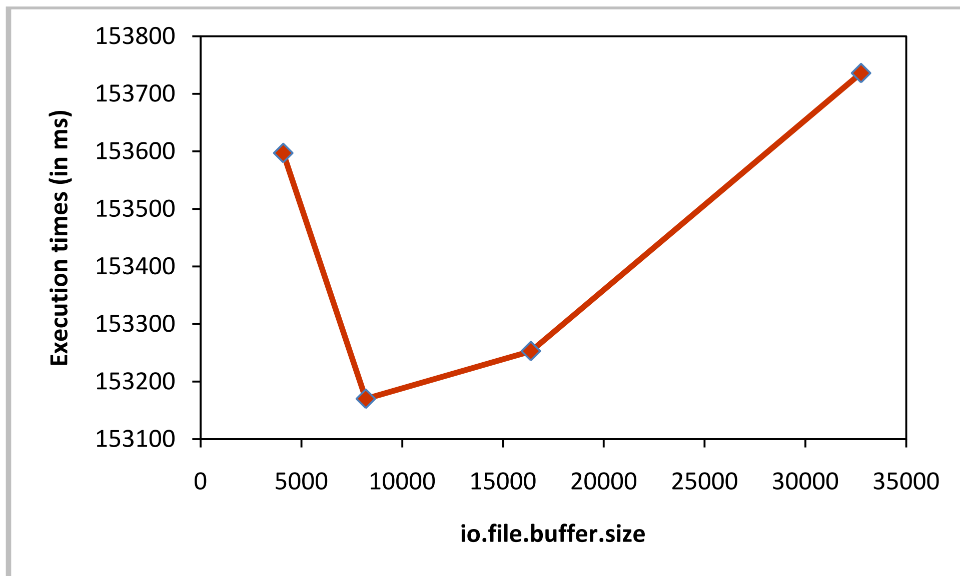
mapreduce.task.io.sort.mb:

Figure 23 – Graph for varying mapreduce.task.io.sort.mb in PageRank



io.file.buffer.size:

Figure 24 – Graph for varying io.file.buffer.size in PageRank

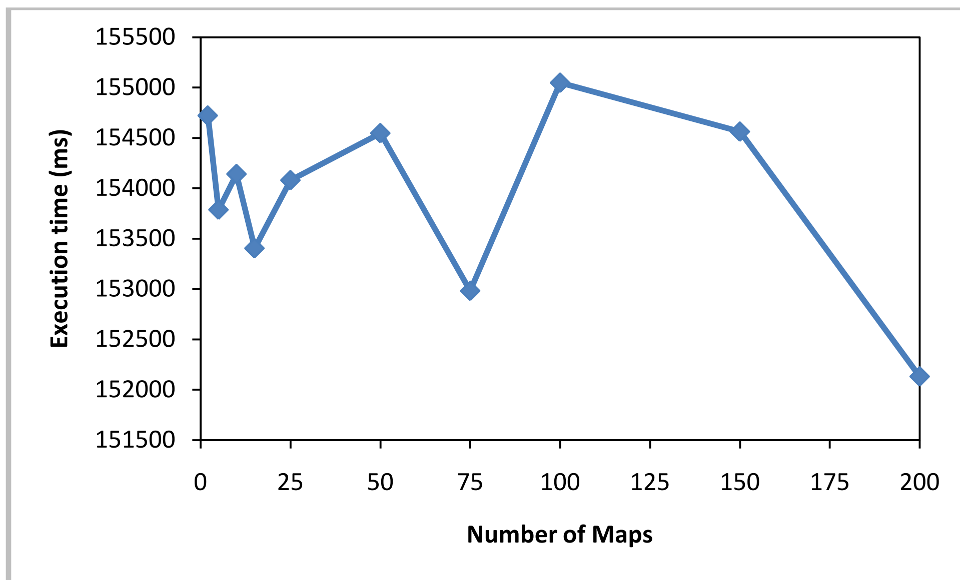


5.3 Word Count Evaluation

For the experiment, we have considered the Cranfield Data Set (5.48 MB). Tokenizing and counting the words is done using MapReduce. The results of varying parameters are as follows

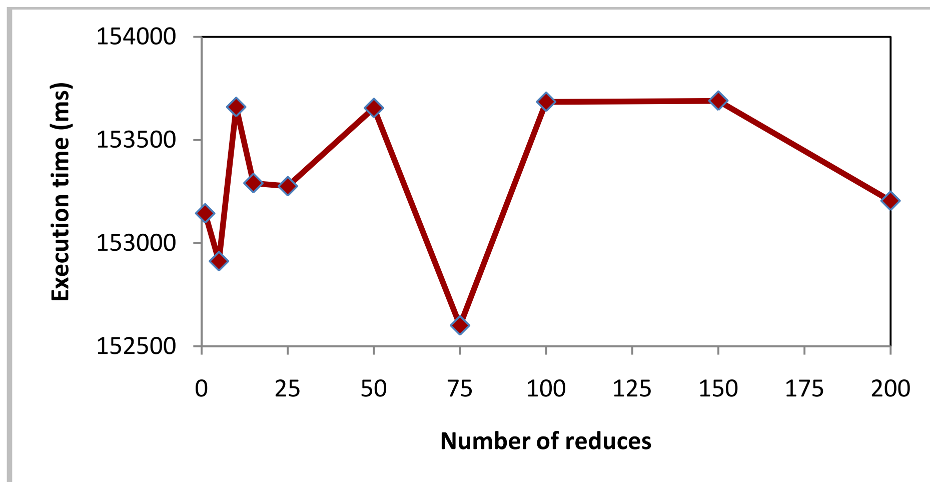
Varying Number of mappers:

Figure 25 – Graph for varying maps in Word Count.



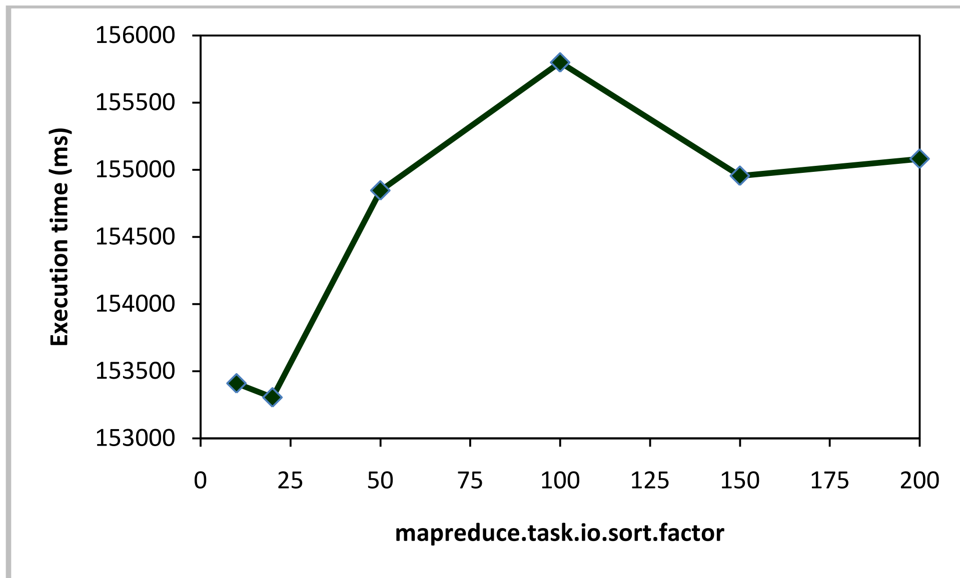
Varying Number of Reducers:

Figure 26 – Graph for varying reducers in Word Count.



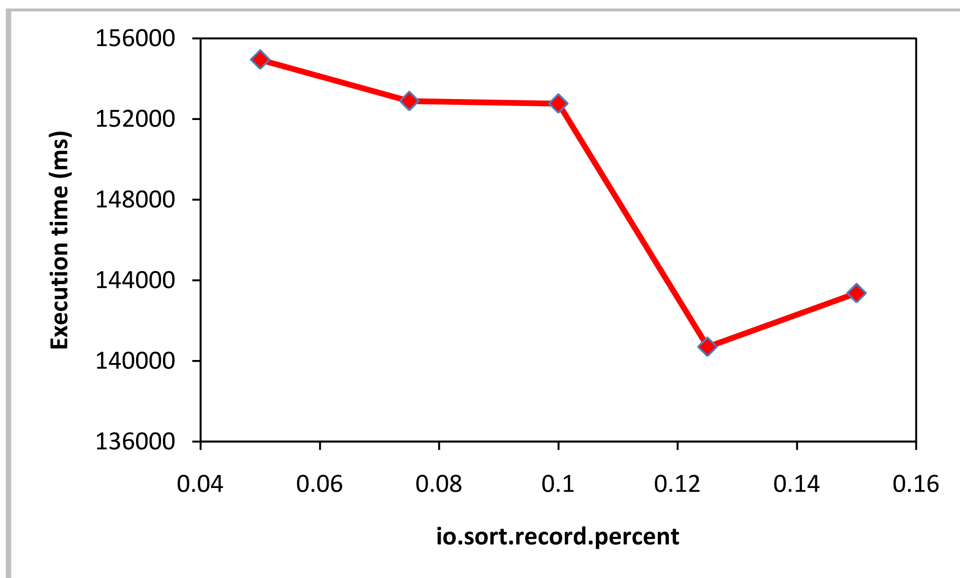
io.sort.factor:

Figure 27 – Graph for varying io.sort.factor in Word Count.



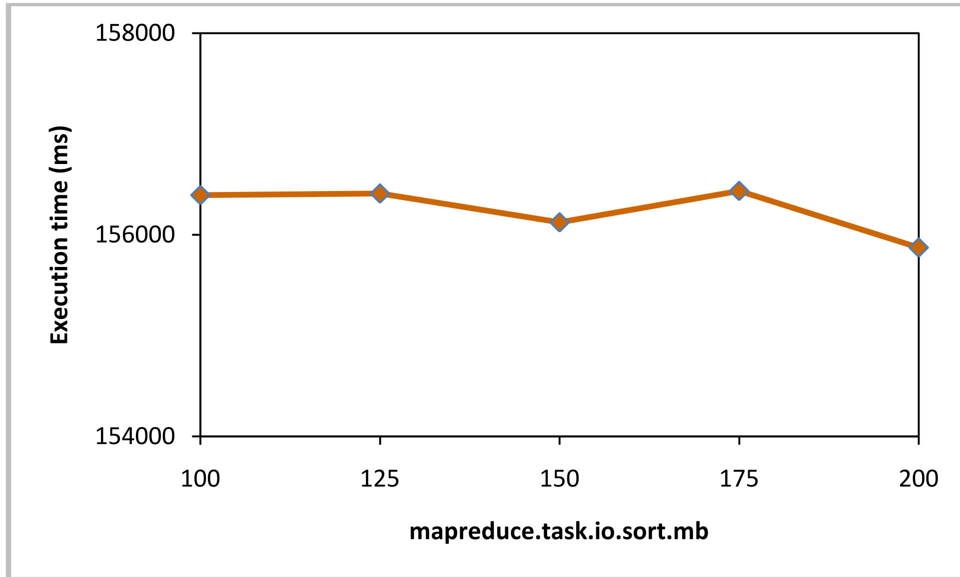
io.sort.record.percent:

Figure 28 – Graph for varying io.sort.record.percent in Word Count



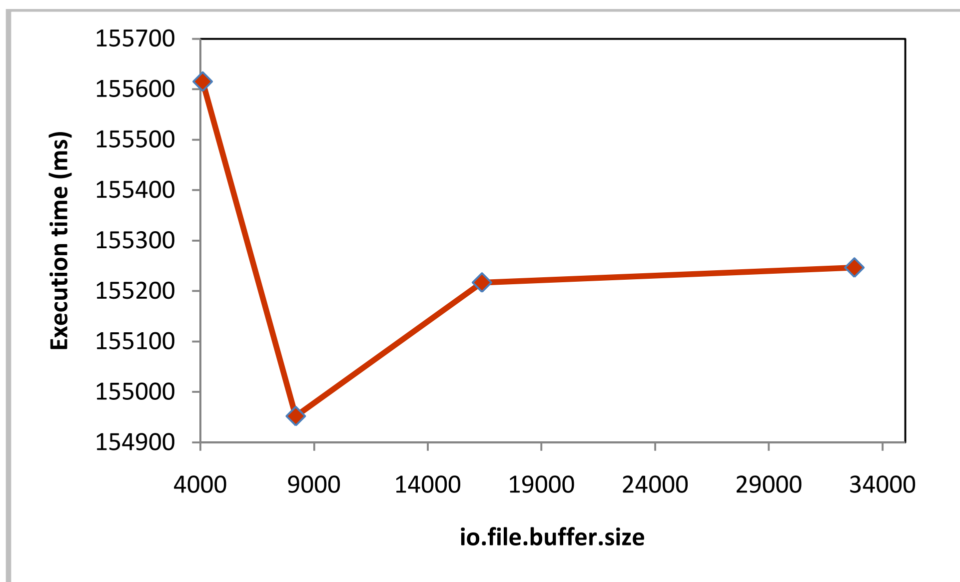
mapreduce.task.io.sort.mb:

Figure 29 – Graph for varying mapreduce.task.io.sort.mb in Word Count



io.file.buffer.size:

Figure 30 – Graph for varying io.file.buffer.size in Word Count

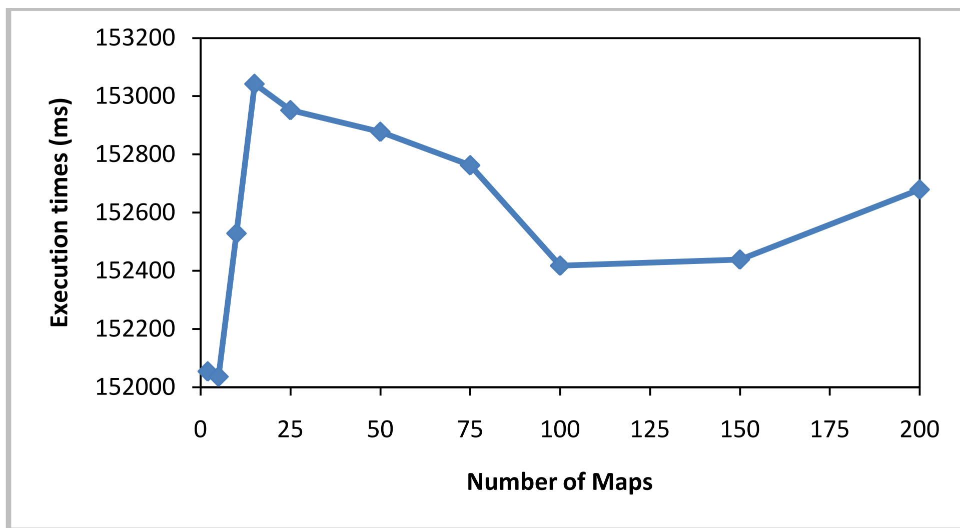


5.4 Sort Evaluation

For the experiment, we have considered the Cranfield Data Set (5.48 MB). Tokenizing, counting the words and sorting the words are done using MapReduce. The results of varying parameters are as follows.

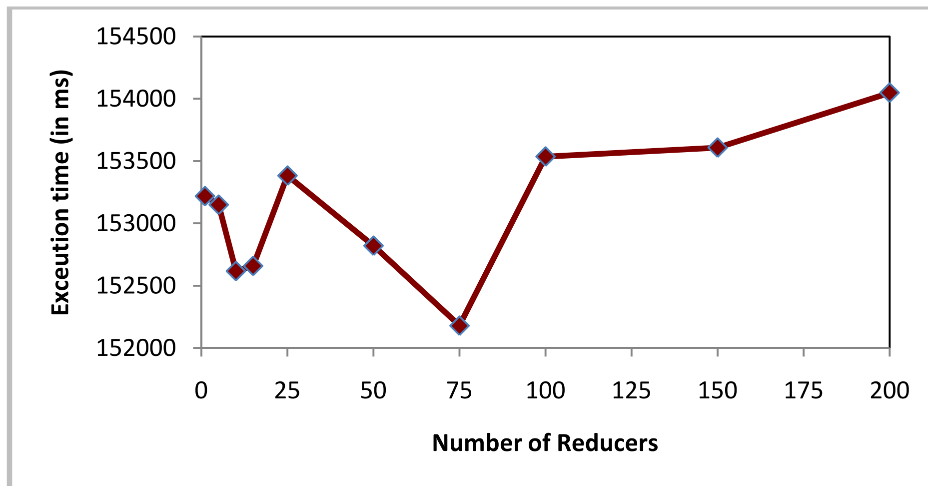
Varying Number of mappers:

Figure 31 – Graph for varying maps in Sort



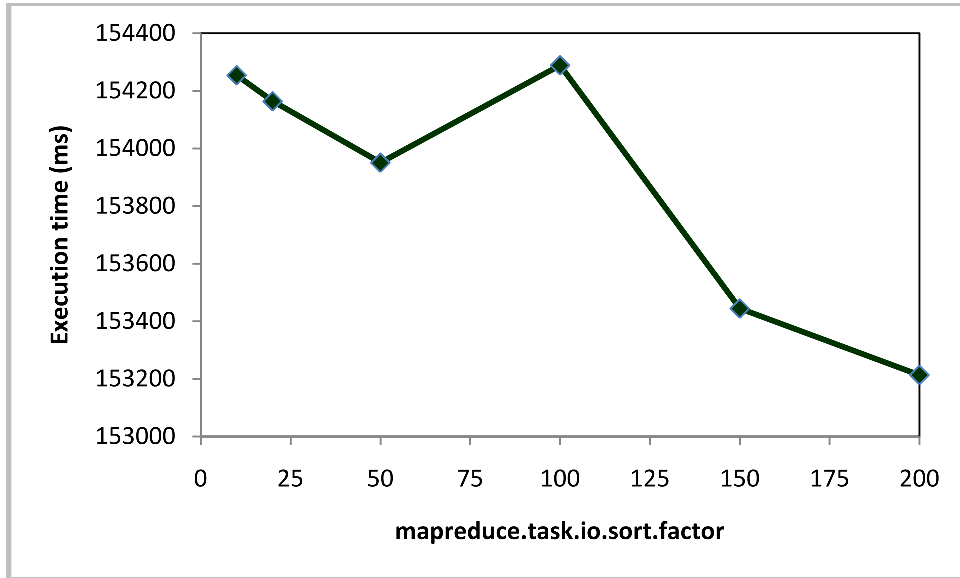
Varying Number of Reducers:

Figure 32– Graph for varying reducers in Sort



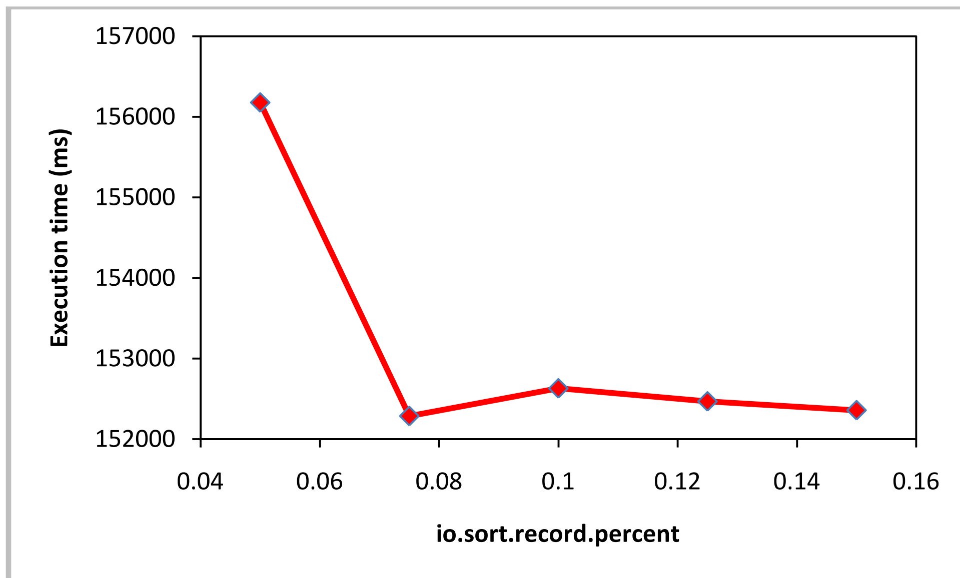
io.sort.factor:

Figure 33 – Graph for varying io.sort.factor in Sort



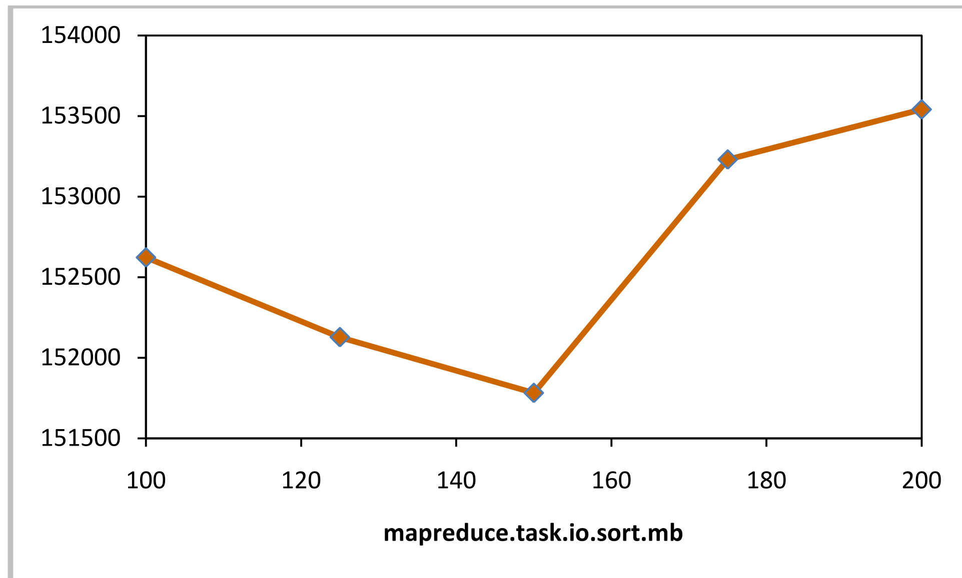
io.sort.record.percent:

Figure 34 – Graph for varying io.sort.record.percent in Sort



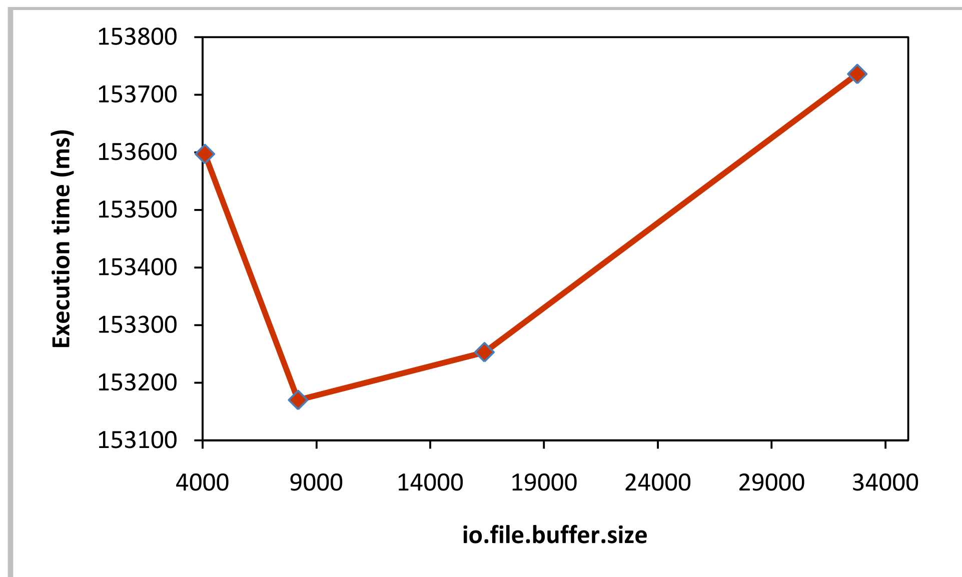
mapreduce.task.io.sort.mb:

Figure 35– Graph for varying mapreduce.task.io.sort.mb in Sort



io.file.buffer.size:

Figure 36 – Graph for varying io.file.buffer.size in Sort



From the above graphs we can see that the performance of the MapReduce programs vary based on the parameter combination. Along with this it is evident that each of the algorithms behaves differently for different parameters based on the type of work and dataset they work on. Now let us see the results of MapReduce programs that have been optimized by the optimizer.

5.5 Optimizer Evaluation

5.5.1 Anagram Evaluation

Anagrams are nothing but words obtained by rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters. We have taken a MapReduce program to find out all the anagrams in the given data corpus. For this we considered the Cranfield Data Set. The algorithm is sent to the optimizer to get the efficient parameter combination and after setting that parameter combination to the algorithm, it is run three times and the execution times are aggregated to dampen the effect of scheduling overhead and other external factors. The results of the executions are as follows.

Figure 37 – Anagram Execution time on Cranfield Data Set comparison graph.

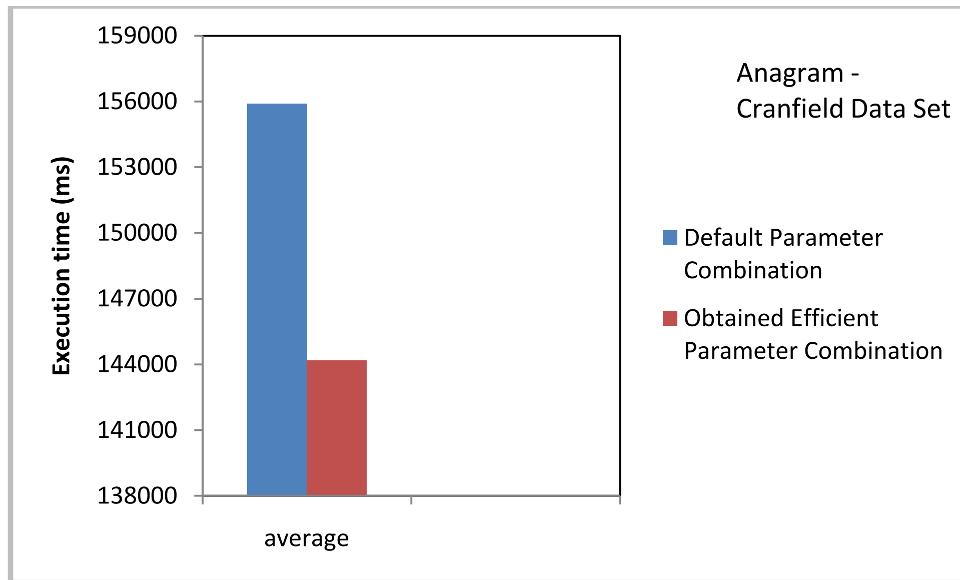


Table 5 – Execution Times of Anagram MapReduce Program on Cranfield Data Set

Execution	Default Parameter Combination Execution time (ms)	Obtained Efficient Parameter Combination Execution time (ms)
Execution 1	156622	144309
Execution 2	155502	144159
Execution 3	155541	144050
Average Execution	155888.3	144172.6

Trained on 2% Cranfield Data Set.

Tested data set: Cranfield Data Set (5.48 MB).

Execution time of optimizer: ~ 24 sec

ExecTime of Anagram (Default) = 155888.3 ms

ExecTime of Anagram (Obtained) = 144172.6 ms

Time gained on each execution = 11715.7 ms

Num of Exec to overcome optimizer over head (24000/11715.7) = 2.048 ~ 3 iterations

We have seen from the above results that the obtained parameter combination gives better results. The time taken to execute the optimizer for this algorithm is ~ 24 seconds. So, to cover up the overhead caused by the optimizer the algorithm has to run for ~ 3. As we know that many MapReduce algorithms we be executed over and over again on daily basis, this is an efficient solution.

Figure 38 – Anagram Execution time on NSF Data Set comparison graph.

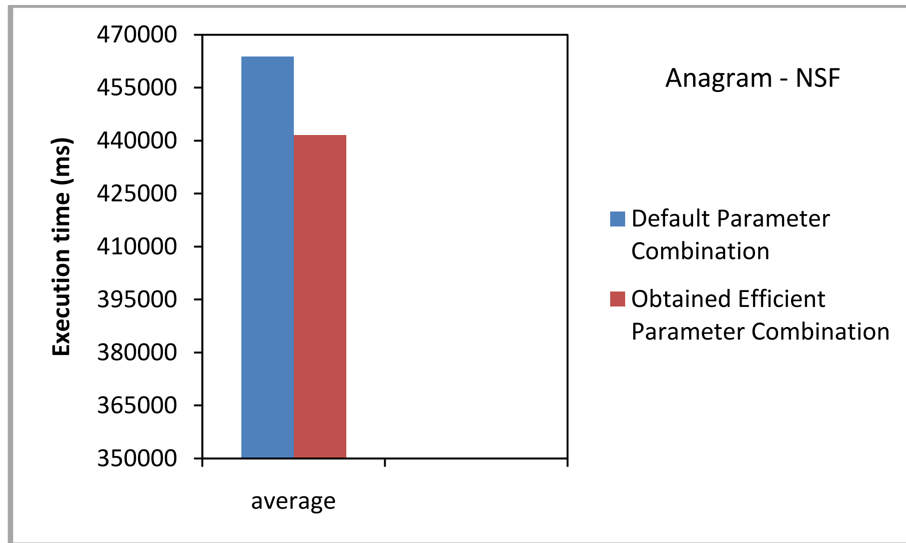


Table 6 – Execution Times of Anagram MapReduce Program on NSF Data Set

Execution	Default Parameter Combination Execution time (ms)	Obtained Efficient Parameter Combination Execution time (ms)
Execution 1	465906	443735
Execution 2	464600	440531
Execution 3	460969	440125
Average Execution	463825	441463.67

Optimized on 2% Cranfield Data Set.

Tested data set: NSF Abstracts (16 MB).

Execution time of optimizer: ~ 24 sec

ExecTime of Anagram (Default) = 463825 ms

ExecTime of Anagram (Obtained) = 441463.67 ms

Time gained on each execution = 22361.33 ms

Num of Exec to overcome optimizer over head (24000/ 22361.33) = 1.07 ~ 2 iterations

We have seen from the above results that the obtained parameter combination gives better results. The time taken to execute the optimizer for this algorithm is ~ 24 seconds. So, to come over the overhead caused by the optimizer the algorithm has to run for ~ 2. As we know that many MapReduce programs are written once and executed over and over again it is an efficient solution.

5.5.2 Distributed Grep Evaluation

The distributed Grep program has the same general functionality as the well-known command line program. It tests every line of the input files with a given pattern which is defined by a regular expression. The map function emits a line if it matches a given pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output. For this we considered the Cranfield Data Set. The algorithm is sent to the optimizer to get the efficient parameter combination. The efficient parameter combination is set to the MapReduce program and executed three times to dampen the effect of external sources. The results of the executions are as follows.

Figure 39 – Grep Execution time on Cranfield Data Set comparison graph.

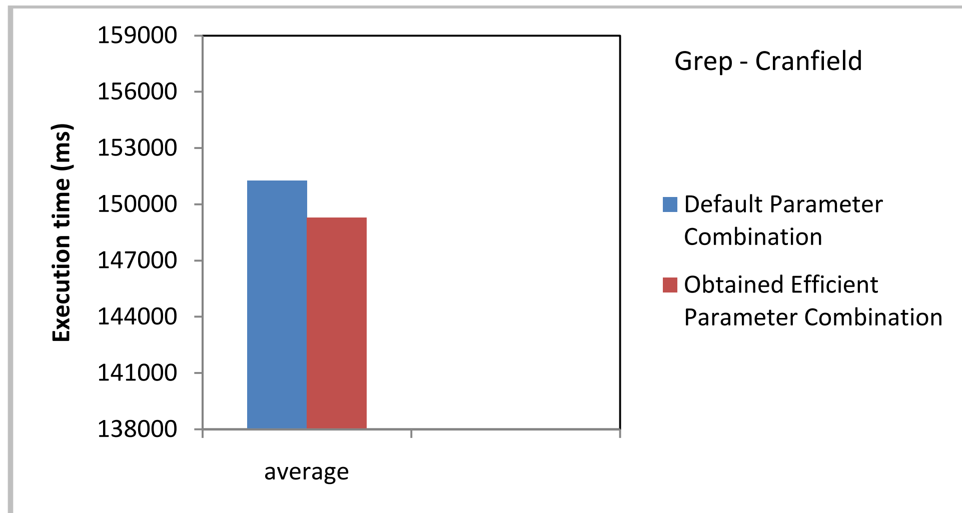


Table 7 – Execution Times of Grep MapReduce Program on Cranfield Data Set

Execution	Default Parameter Combination Execution time (ms)	Obtained Efficient Parameter Combination Execution time (ms)
Execution 1	150638	150013
Execution 2	153060	148435
Execution 3	150153	149419
Average Execution	151283.7	149289

Optimized on 3% Cranfield Data Set.

Tested data set: Cranfield Data Set (5.48 MB).

Execution time of optimizer: ~ 34 sec

ExecTime of Grep (Default) = 151283.7 ms

ExecTime of Grep (Obtained) = 149289 ms

Time gained on each execution = 1194.7 ms

Num of Exec to overcome optimizer over head (34000/ 1194.7) = 28.46 ~ 29 iterations

We have seen from the above results that the obtained parameter combination gives slightly better results. The time taken to execute the optimizer for this algorithm is ~ 34 seconds. To overcome the overhead from the optimizer, the Grep program needs to run for ~ 29 times (34000/1194.7 = 28.46). Grep is very much used for searching in systems, so this above parameter combination can be said as an efficient one.

Figure 40 – Grep Execution time on NSF Data Set comparison graph.

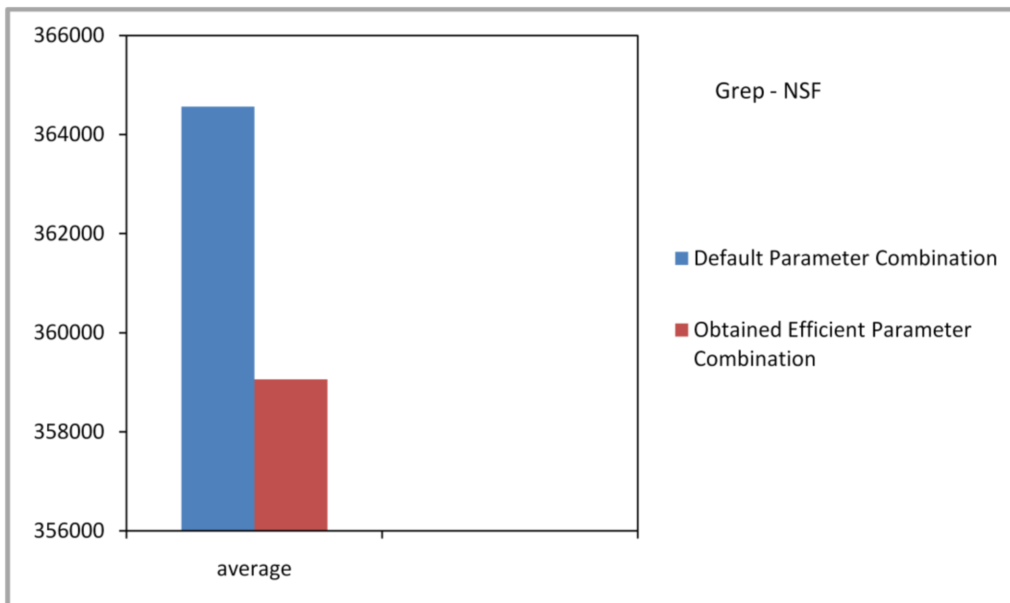


Table 8 – Execution Times of Grep MapReduce Program on NSF

Execution	Default Parameter Combination Execution time (ms)	Obtained Efficient Parameter Combination Execution time (ms)
Execution 1	363703	358782
Execution 2	364344	359719
Execution 3	365640	358672
Average Execution	364562.33	359057.66

Optimized on 3% Cranfield Data Set.

Tested data set: NSF Abstracts (16 MB).

Execution time of optimizer: ~ 34 sec

ExecTime of Grep (Default) = 364562.33 ms

ExecTime of Grep (Obtained) = 359057.67 ms

Time gained on each execution = 5504.66 ms

Num of Exec to overcome optimizer over head (34000/ 5504.66) = 6.17 ~ 7 iterations

We have seen from the above results that the obtained parameter combination gives slightly better results. The time taken to execute the optimizer for this algorithm is ~ 34 seconds. To overcome the overhead from the optimizer, the Grep program needs to run for ~ 7 times. As we know that many MapReduce algorithms we be executed over and over again on daily basis, this is an efficient solution.

5.6 Testing

The project has been developed in stages and at each stage there are new set of requirements that were being handled. These requirements are developed and tested and integrated with the main program. So, we can say that the project has been developed in a spiral model.

The testing done in this project can be grouped under three different kinds of testing. They are

1) Unit Testing.
2) Functional Testing (Black-box)
3) Integration Testing.

Each of the modules that are developed has been tested as a unit and then the functionality of the module is tested by giving test cases. This module is then integrated with the main program and the main program after integration is tested for integration errors as well as for functionality with sample test plans.

5.6.1 Unit Testing

“Unit testing refers to those tests that verify the functionality of a specific section of the code, usually at the function level. In an object oriented environment this is usually at the class level and the unit tests include the constructors and the destructors. The primary goal is to take the smallest piece of testable software, isolate it from the rest of the code and check if it is behaving exactly as expected. Each unit is tested separately before integrating them into modules to test the interfaces between modules.” [17]

Each module of the project has been tested for errors and functionality. Each phase in optimizer, contributor have been tested. In the optimizer, we have the execution module which is intended to read the parameters from the parameterCombinationFile execute the given MapReduce program over the parameters and write the execution times on to the execution times

file. This has three units, reading, executing and writing. Each of the unit is tested for functionality manually by seeing the outputs from each of the unit. The next phase consists of the sorting. Here the execution time file is sorted with keeping track of the parameter combination that it belongs to.

For example, the first two execution times of the file after sorting are:

- Execution time: 33489 ms
- Execution time: 33631 ms

We can see that the execution times are sorted. Each of the units is tested and checked for outputs in the above manner.

5.6.2 Functional Testing (Black Box)

"Functional testing is a type of black box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered."[18]

Each of the modules is tested for its functionality but executing the module with sample inputs and then checking the output manually.

For example, the comparisons of the execution times from the optimizer are:

- Anagram model output (Default): 155888.3 ms
- Anagram model output (Obtained): 144172.6 ms

The Anagram program is executed over the Cranfield Data Set with the default parameter combination and the obtained parameter combination from the optimizer. We can see that the obtained parameter combination has given an efficient solution. So, we can say that the optimizer is functionally working.

5.6.3 Integration testing:

"Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before system testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system."[19]

As mentioned earlier in the implementation, optimizer and contributor have three phases each. Each of the phases is developed individually and then integrated with the other phases to form the optimizer and contributor. So, after each of the integrations the outputted phases have been tested with sample inputs. The output is checked manually for the required output.

For example, after integration the optimizer output looks like:

- Grep output (Default): Execution time: 3484 ms
- Grep output (obtained): Execution time: 3360 ms

The Grep program is executed over the 3% of Cranfield Data Set with the default parameter combination and the obtained parameter combination from the optimizer. Here we can see that an optimized output has resulted from the optimizer. So, the integration is successful and the output is as desired.

Chapter 6 - CONCLUSION and FUTURE WORK

CONCLUSION

A framework for optimizing the MapReduce has been discussed in this report. This framework is a partial brute force approach and helps to work towards automatic optimization of MapReduce programs. The evaluation of various algorithms has been done to show empirical evidence on how the parameters affect the performance of the MapReduce programs. The evaluation results from the optimizer set parameter combinations for the Grep and anagram programs show that this approach gives efficient solutions. So, we can say the this approach gives a better solution but not the best as the parameter combinations in the optimizer are limited and these combinations may give not be the best combinations for a given MapReduce program.

If the optimizer sets the default parameter combination as the efficient combination for a program from the parameterCombinationFile, then the approach fails. So, to write it in the form of an equation:

Execution Time (E_{op}) <= Execution Time ($E_{default}$)

Here E_{op} is the execution time from obtained efficient parameter combination and $E_{default}$ is the execution time from the default parameter combination.

As mentioned earlier that the MapReduce programs are used over and over again on different data sets for various applications helped to lay the objective for this project. The repetitive use of a MapReduce program helps to eliminate the overhead caused by the optimizer and then save the execution time of the application from there onwards except in the case where the efficient combination is the default combination.

LIMITATIONS

As this is a partial brute force approach, there are few limitations in this project. They are as follows:

- As mentioned earlier, if the efficient combination is the default combination then the overhead of optimizer execution will be wastage of time and resources.
- The initial parameterCombinationFile doesn't contain parameter combinations from machine learning and Data base related MapReduce programs.
- This project only considered a subset of job level configuration parameters. But there are many other configuration parameters to be considered at the cluster and file system level.
- The user needs to provide the partial data set to the optimizer by himself rather than the optimizer obtaining it directly from the complete dataset.

FUTURE WORK

- Incorporating parameter combinations in parameterCombinationFile from machine learning and Data base related MapReduce programs.
- Implementing the project with larger data sets and on cluster with various algorithms.
- Providing the users an interface to specify the range of the parameter values for the parameters as different users have different cluster configurations. For example, number of mappers and reducers play differently by the number of nodes available for execution.
- Making the optimizer to obtain the input data from the given complete data set.
- Fine graining the optimizer by considering the execution times of mappers and reducers individually.

Chapter 7 - REFERENCES

[1] J. Dean and S. Ghemawat. **MapReduce: Simplified Data Processing on Large clusters**. In Proc. of OSDI, 2004.

[2] Jason Venner**, Pro Hadoop**, Apress Publications, 2009.

[3] Tom White, **Hadoop: The Definite Guide**, O'Reilly and Yahoo! Press, 2009.

[4] Shivnath Babu, **Towards Automatic Optimization of MapReduce Programs**, SoCC'10, Indianapolis, Indiana, USA, 2010.

[5] Guanying Wang, Ali R. Butt, Prashant Pandey, Karan Gupta, **Using Realistic Simulation for Performance Analysis of MapReduce Setups**, LSAP '09 Munich, Germany. 2009.

[6] Hadoop Map/Reduce tutorial, *http://hadoop.apache.org/core/docs/current/mapred tutorial.html*

[7] Apache Hadoop, http://hadoop.apache.org/

[8] Ricky Ho, Pragmatic Programming Techniques, http://horicky.blogspot.com/2010/08/designing-algorithmis-for-map-reduce.html

[9] Dan Gillick, Arlo Faria, John DeNero, **MapReduce: Distributed Computing for Machine Learning**, December 18, 2006.

[10] Jimmy Lin and Chris Dyer, **Data-IntensiveText Processing with MapReduce**, Morgan & Claypool publishers, 2010.

[11] Eaman Jahani, Michael J. Cafarella, Christopher R´e, **Automatic Optimization for MapReduce Programs**, Proceedings of the VLDB Endowment, Vol. 4, No. 6 . Copyright 2011.

[12] Todd Lipcon, Cloudera, 7 Tips for Improving MapReduce Performance , http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/

[13] Tech Musings! AN ELEGANT TAPESTRY OF TECHNOLOGY , http://thetechmusings.wordpress.com/2011/02/28/hadoop-an-inspirational-clone-of-gfs/

[14] Mitesh's Blog, It's all about Basics, http://cleanclouds.wordpress.com/author/cleanclouds/

[15] Dawei Jiang, Beng Chin Ooi, Lei Shi, Sai Wu, **The Performance of MapReduce: An Indepth Study,** 36th International Conference on Very Large Data Bases, Singapore. 2010.

[16] A. Thusoo et al. Hive **- A Warehousing Solution Over a Map-Reduce Framework**. PVLDB, 2(2):1626-1629, 2009.

[17] Unit Testing - http://en.wikipedia.org/wiki/Unit_testing , July 2011.

[18] Functional Testing - http://en.wikipedia.org/wiki/Functional_testing , July 2011.

[19] Integration Testing - http://en.wikipedia.org/wiki/Integration_testing , April 2011.