Understanding Spark System Performance for Image Processing in a Heterogeneous Commodity Cluster

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Owolabi Adekoya

©Owolabi Adekoya, July/2018. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Head of the Department of Computer Science
University of Saskatchewan
176 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

In recent years, Apache Spark has seen a widespread adoption in industries and institutions due to its cache mechanism for faster Big Data analytics. However, the speed advantage Spark provides, especially in a heterogeneous cluster environment, is not obtainable out-of-the-box; it requires the right combination of configuration parameters from the myriads of parameters provided by Spark developers. Recognizing this challenge, this thesis undertakes a study to provide insight on Spark performance particularly, regarding the impact of choice parameter settings. These are parameters that are critical to fast job completion and effective utilization of resources.

To this end, the study focuses on two specific example applications namely, *flowerCounter* and *imageClustering*, for processing still image datasets of Canola plants collected during the Summer of 2016 from selected plot fields using timelapse cameras in a heterogeneous Spark-clustered environments. These applications were of initial interest to the Plant Phenotyping and Imaging Research Centre (P^2IRC) at the University of Saskatchewan. The P^2IRC is responsible for developing systems that will aid fast analysis of large-scale seed breeding to ensure global food security. The *flowerCounter* application estimates the count of flowers from the images while the *imageClustering* application clusters images based on physical plant attributes. Two clusters are used for the experiments: a 12-node and 3-node cluster (including a master node), with Hadoop Distributed File System (HDFS) as the storage medium for the image datasets.

Experiments with the two case study applications demonstrate that increasing the number of tasks does not always speed-up job processing due to increased communication overheads. Findings from other experiments show that numerous tasks with one core per executor and small allocated memory limits parallelism within an executor and result in inefficient use of cluster resources. Executors with large CPU and memory, on the other hand, do not speed-up analytics due to processing delays and threads concurrency. Further experimental results indicate that application processing time depends on input data storage in conjunction with locality levels and executor run time is largely dominated by the disk I/O time especially, the read time cost. With respect to horizontal node scaling, Spark scales with increasing homogeneous computing nodes but the speed-up degrades with heterogeneous nodes. Finally, this study shows that the effectiveness of speculative tasks execution in mitigating the impact of slow nodes varies for the applications.

Acknowledgements

There are lot of persons that were instrumental to the success of this study. First, I would like to thank the Almighty God for the grace and wisdom given to me to start and complete this program successfully. I would also like to thank my family for their encouragement and support throughout this program. My profound gratitude goes to the graduate committee of the Department of Computer Science and the Graduate College at large for the financial assistantship given to me towards the completion of this work. Thank you for the privilege of exploring the frontiers of research at the University of Saskatchewan. My sincere appreciation also goes to my supervisors Dr Dwight Makaroff and Dr Derek Eager for their reception, mentorship, guidance, support (both professionally and financially), patience and understanding towards the completion of this study. I will be forever grateful for your kind gesture and assistance towards this research pursuit. I would also like to acknowledge my advisory committee members Dr Kevin Stanley and Dr Ian Stavness for their guidance and support towards this ideal. I would not forget to recognize the efforts of my instructors whose knowledge greatly contributed to my success namely Dr Nathaniel Osgood, Dr Ralph Deters, Dr Michael Horsch & Dr Kevin Stanley. I am also hugely indebted to some Departmental staff including Gwen Lacaster, Brittany Melnyk, Greg Oster, Cary Bernath, Raouf Ajami, Merlin Hansen, Seth Shacter & Jeff Long for their tremendous support all through this study. I also want to appreciate the efforts of Javier Garcia Gonzalez and Amit Mondal for providing the sequential version of the applications used in this study. I also acknowledge the support and encouragement of my research colleagues like Tunde Olabenjo, Habib Ado, Philip Dueck, Winchell Qian, Mohammed Rashid Chowdhury, Fadi Mobayed, Lin Pin & Faheem Abrar. I also appreciate the support of my Spiritual mentors Pastor & Mrs Titus Adedapo, Mr & Mrs Funso Oderinde, Brother Hugo Clarke, Mr Jamiu Sanni, Edward & Priscilla Bam, and especially my Spiritual father Pastor Kola Adebari. Lastly, I would like to thank my covenant brothers Tunde Onafeso and Elijah Ericmoore Jossou for being there in all ramifications during this research adventure and a lot of persons that I can not possibly remember. Thank you all for your support and encouragement. I am deeply grateful.

This work is dedicated to the Almighty God, my family, friends and supervisors

Contents

Permission to Use i

Abstract ii

Acknowledgements iii

Contents v

List of Tables vii

List of Figures viii

List of Abbreviations ix

1 Introduction 1
- 1.1 Thesis Motivation . . . 6
- 1.2 Thesis Statement . . . 6
- 1.3 Thesis Findings . . . 7
- 1.4 Thesis Organization . . . 8

2 Background and Related Work 9
- 2.1 Big Data Analytics Frameworks . . . 9
- 2.2 Spark Overview . . . 10
  - 2.2.1 Spark Resource Managers . . . 13
  - 2.2.2 Spark Storage Systems . . . 13
- 2.3 Spark Operations and Parameters . . . 13
- 2.4 Prior Performance Studies of Spark . . . 20
- 2.5 Chapter Summary . . . 22

3 Experimental Methodology and Design 24
- 3.1 Experimental Methodology . . . 24
  - 3.1.1 Applications Benchmark Characteristics . . . 24
  - 3.1.2 Input Datasets . . . 27
  - 3.1.3 Cluster Setup . . . 28
  - 3.1.4 Performance Characteristics of the Cluster Nodes . . . 31
  - 3.1.5 Measurement Tools and Metrics . . . 32
- 3.2 Experimental Design . . . 36
  - 3.2.1 Experiments to investigate the influence of *spark.files.maxPartitionBytes* on application performance . . . 36
  - 3.2.2 Experiments to understand the influence of *spark.executor.cores* on application performance . . . 36
  - 3.2.3 Experiments to investigate the influence of *spark.executor.memory* parameter on application performance . . . 39
  - 3.2.4 Experiments to study the scalability of the Spark system with increasing number of computing nodes . . . 39
  - 3.2.5 Experiments to investigate the influence of *spark.speculation* configuration parameter on straggling tasks . . . 40

4 Performance Analysis and Evaluation 42
- 4.1 Effect of Partition Size . . . 42

4.2 JVM Executor Scaling . . . 58
4.3 Impact of Caching . . . 71
4.4 Compute Node Scaling . . . 76
4.5 Speculative Task Execution . . . 79
4.6 Chapter Summary . . . 83

5 Conclusion and Future Work **85**
5.1 Summary . . . 85
5.2 Contributions . . . 87
5.3 Future Work . . . 88

References **90**

List of Tables

2.1 Spark Configuration Parameters 19

3.1 Applications Characteristics 27
3.2 Dataset Description 28
3.3 Specification of Each Node of the Larger Cluster 29
3.4 Specification of Each Node of the Smaller Cluster 30
3.5 Specification of the Benchmark Machine (*Onomi*) 30
3.6 Spark Configuration Parameters for Experimental Design 30
3.7 Performance Characteristics of Nodes 33
3.8 Configuration Settings for Experiments in §3.2.1 37
3.9 Configuration Settings for Experiments in §3.2.2 38
3.10 Configuration Settings for Experiments in §3.2.3 40
3.11 Configuration Settings for Experiments in §3.2.5 41

4.1 *flowerCounter*: Influence of *spark.files.maxPartitionBytes* (July, 34.8 GB Dataset) 44
4.2 *flowerCounter*: Influence of *spark.files.maxPartitionBytes* (July, 34.8 GB Dataset) 49
4.3 *flowerCounter*: Influence of *spark.executor.cores* (August, 19.3 GB Dataset) 59
4.4 Execution Time Summary of 9 JVM Executors for *imageClustering* (128 MB) 63
4.5 Execution Time Summary of 18 JVM Executors for *imageClustering* (128 MB) 63
4.6 Average Cluster Resource Usage Metrics for *flowerCounter* (64 MB) 63
4.7 Average Cluster Resource Usage Metrics for *imageClustering* (128 MB) 65
4.8 *flowerCounter*: Influence of *spark.executor.cores* 67
4.9 *imageClustering*: Influence of *spark.executor.cores* (July, 34.8 GB Dataset) 67
4.10 *flowerCounter*: Influence of Caching (July, 34.8 GB Dataset) 74
4.11 *imageClustering*: Influence of Caching (September, 5.7 GB Dataset) 75
4.12 *flowerCounter*: Locality Level Summary (July, 34.8 GB Dataset) 75
4.13 *imageClustering*: Locality Level Summary (September, 5.7 GB Dataset) 75
4.14 *flowerCounter*: Tasks Execution Time Summary on all Nodes (July, 34.8 GB Dataset) 78
4.15 Runtime Comparison between the Sequential & Spark *flowerCounter* applications 79
4.16 *flowerCounter*: Tasks Execution Time Summary with Speculation (July, 34.8 GB Dataset) . 82
4.17 *imageClustering*: Tasks Execution Time Summary without Speculation (July, 34.8 GB Dataset) 82
4.18 *imageClustering*: Tasks Execution Time Summary with Speculation (July, 34.8 GB Dataset) 83

List of Figures

2.1 The Berkeley Data Analytics Stack (BDAS) [3] 11
2.2 Components of a Spark Application [49] 12
2.3 The Spark workflow 17
2.4 The Shuffle Phase 18

4.1 Partition Size vs Runtime 43
4.2 Stage Time Summary 46
4.3 Task Progress Summary 47
4.4 Task Progress Summary 48
4.5 CPU Utilization 50
4.6 CPU HeatMap 52
4.7 RAM Usage 53
4.8 Disk Throughput 54
4.9 Network Throughput 55
4.10 HDFS IO 57
4.11 Tasks Completion Progress Summary 60
4.12 Tasks Completion Progress Summary 64
4.13 Task Completion Progress Summary 65
4.14 Execution Time of Core Scaling 69
4.15 Speedup of Core Scaling 70
4.16 Influence of *spark.executor.memory* Parameter 71
4.17 Runtime vs Storage Level 73
4.18 Effect of Node Scaling 77
4.19 Influence of Slow Nodes 78
4.20 Speculative and non-Speculative *flowerCounter* appilcation execution 80

List of Abbreviations

IoT	Internet of Things
AMPlab	Algorithms, Machines, & People Lab
TPC-DS	Transaction Processing Performance Council Decision-Support
P^2IRC	Plant Phenotyping and Imaging Research Centre
JVM	Java Virtual Machine
UI	User Interface
HDFS	Hadoop Distributed File System
RDD	Resilient Distributed Dataset
BDAS	Berkeley Data Analytics Stack
API	Application Programming Interfaces
DAG	Directed Acyclic Graph
YARN	Yet Another Resource Negotiator
SSD	Solid-State Drive
SciDB	Scientific Database
HAT	History-Based Auto-tuning
PIL	Python Image Processing Library
MLlib	Spark Machine Learning Library
SIFT	Scale Invariant Feature Transform
KVM	Kernel-based Virtual Machines
HT	Hyper-Threading
FSB	Front-Side Bus
QPI	Quick Path Interconnect
DMI	Direct Media Interface
TBT	Turbo Boost Technology
JSON	JavaScript Object Notation
GC	Garbage Collection

Chapter 1

Introduction

The MapReduce [17] programming paradigm has been effective for Big Data processing in a fast and scalable fashion. The execution model in MapReduce works by decomposing Big Data processing jobs into Map and Reduce sub tasks and then assigning those tasks to cluster computing nodes. Hadoop MapReduce was developed as an open-sourced alternative of the MapReduce programming paradigm. It is the parallel processing model of Hadoop [24] for processing and analyzing Big Data workloads. However, owing to the continued evolution of Big Data due to innovative exploits in computing domains such as mobile and social computing, Internet of Things (IoT) amongst other new domains, researchers and industry practitioners have found that MapReduce is no longer effective for some new emerging low latency and data-intensive applications such as iterative, data mining and graphical applications due to expensive I/O overheads and data serialization. This has led researchers and other concerned stakeholders to developing variant arrays of computing frameworks to meet these new emerging application scenarios. Apache Spark [62] is one of the new emerging in-memory analytics frameworks for Big Data processing designed ground up with the MapReduce programming paradigm. Spark, developed in 2009 at UC Berkeley AMPLab,[1] open-sourced in 2010 became a top-level Apache project in 2014, is a data analytics engine designed to handle this new set of emerging applications.

Identifying performance bottlenecks in analytics frameworks such as Spark is difficult due to its in-memory and end-to-end design paradigm [40]. Substantial efforts have been directed towards understanding and identifying performance bottlenecks [8, 40, 51] in analytics frameworks as well as improving their performance [2, 4, 45]. However, most of these works involved using transactional and production traces such as the Transaction Processing Performance Council's decision-support benchmark (TPC-DS) [15] and not real applications. Only recently did Zhang *et al.* [64] investigate Spark's performance for real scientific data analysis in comparison with the traditional disk-based SciDB [9] software systems. Nonetheless, there has not been much work done in understanding Spark's performance with respect to specific categories of scientifc applications with binary data such as image processing applications with real image datasets. Therefore, this work focuses on understanding the performance of Spark system for two image processing applications (*flowerCounter* and *imageClustering* developed using the python APIs of the Spark System - Pyspark[2]) in a

[1] https://amplab.cs.berkeley.edu/(Accessed: April 21, 2018)
[2] http://spark.apache.org/docs/latest/api/python/(Accessed: April 21, 2018)

heterogeneous commodity cluster. The larger cluster consists of 12 commodity personal computers with varying configurations while the other smaller cluster consists of 3 larger servers with more computing resources respectively.

The two applications (*flowerCounter* and *imageClustering*) were selected because of their importance to the analytics pipeline of the global research project on food security currently being undertaken by the University of Saskatchewan Plant Phenotyping and Imaging Research Centre (P^2IRC).[3] The overarching goal of P^2IRC is to developed systems that will provide information about plants traits for transforming large-scale seed breeding to support global food security.

The *flowerCounter* application is one of the applications being developed by the Data Analysis for Rapid Plant Phenotyping[4] project group under the Computational Informatics of Crop Phenotype Data research theme. Its purpose is to analyze the rate at which plant flowers are growing on plot fields using timelapse images collected from stationary cameras.

On the other hand, the *imageClustering* application categorizes plant images based on observable physical attributes such as colour, brightness or size. The different categories can then be further analyzed for more insights. These applications are particularly selected to meet future scalability demands.

Processing the massive data with the sequential versions of these applications will be computationally expensive and last for several hours or days. Cluster operation with Spark will permit appropriate parallel processing. Furthermore, this provides opportunity to understand how Spark can be efficiently tuned in order to aid the processing of images. Thus, this thesis contributes to the investigation of distributed frameworks for image analysis applications within the context of P^2IRC project, but can also be more generally applicable to configuration requirements for general Spark jobs.

These two sequential applications are selected not because of their inherent affinity to the Spark processing model but primarily for the purpose of investigating whether their performance can be improved by the sharing of cluster resources using the Spark programming model. The applications are not particularly suited to the Spark programming framework because they are based on the manipulation of large numpy arrays of binary data. The processing of large binary data by Spark is still an experimental feature and thus may not take advantage of Spark's unique features. In particular, the Spark programming model favours applications that produce intermediate data output which can be persisted in memory for subsequent processing in the pipeline. This is the cache design paradigm inherent in the Spark processing engine for optimal performance. However, the *flowerCounter* application, for example, differs from this memory abstraction design as the application involves very little in-memory intermediate data results.

Moreover, the applications (*flowerCounter* and *imageClustering*) used in this study are particularly important for subsequent image processing operation analyses as they will help to provide information about the growth of flowers on respective farm plots. The information obtained from the processing of the image

[3]http://p2irc.usask.ca/index.php(Accessed: February 5, 2018)
[4]http://p2irc.usask.ca/themes/computing/project-3-2.php#ProjectGoal(Accessed: February 5, 2018)

datasets with these applications can aid further analyses for example, to determine the cause(s) of the patterns observed which could be due to genetics, soil content and structure, environmental factors, weather conditions or any other cause(s) whatsoever. These applications may be particularly amenable to features of Spark and thus having a common processing paradigm could be useful for development and deployment.

To this end, this thesis investigates the effect(s) that the configurable engine parameters would have on the application processing time as well as how the underlying cluster computing resources are being utilized with the view of identifying performance bottlenecks in these applications. The sequential version of these applications is used as a baseline for resource consumption and processing time comparisons with the corresponding Spark versions.

A Spark application is expressed as a series of jobs depending on the inherent logic of the application. A *job* consists of a series of stages which are defined by high level distributed operations (Spark transformations and actions) such as *map*, *flatMap*, *filter*, *coalesce*, *reduceByKey*, *repartition*, *collect* and *saveAsTextFile*. Some of these operations such as *coalesce*, *repartition*, and *reduceBykey* involve the movement of data across the nodes in the cluster over the network. This transfer of data across the nodes is otherwise referred to as a shuffling operation. A *stage* is made up of parallel tasks that are executed by JVM executor processes running on the physical machines in the cluster. A *task* is essentially the unit of work corresponding to one data partition within a stage. The number of data partitions in a stage is determined by the size of the input data and the specified data block size. The default block size in Spark (a data partition size) is 128 MB.

Data skew has been identified as one of the causes of straggling tasks; its cause is mis-configured data partition sizes. Mis-configured partition sizes increase CPU computation time especially for shuffle operations and thus prolong job completion times [40, 51]. Spark calculates the number of tasks based on the data block size. Hence, if the number of tasks is smaller than the number of available CPU cores, CPU usage will be sub-optimal. Therefore, knowing the right partition size for Spark jobs (especially shuffle-heavy applications) will help to achieve optimal parallelism. However, determining the correct partition size for different workloads is not a trivial task; it involves using heuristic approaches to finding an optimal configurable size for different application scenarios [51, 64]. As a result, one aspect of this work investigates the effect of data partition size on the performance of the image processing applications.

Effective tuning of Java Virtual Machine (JVM) parameters is key to Spark application performance as the engine runs on the JVM. Badly configured Spark applications for JVM parameters such as the heap size and threads will degrade their performance due to increased garbage collection overhead, thread contention and context switching among other causes. Thus, the performance of Spark jobs is largely dependent on the right configuration of the JVM heap size as well as the number of JVM executors.

Chiba and Onodera [12] observed that using a single JVM executor is not an optimal solution. This might be due to the fact that Hadoop Distributed File System (HDFS) [47] client does not handle concurrent threads created by an executor with many cores well, thus leading to bad HDFS I/O throughput. This is because the number of concurrent operations that HDFS can handle is dependent on the number of data

blocks stored on the node.

As a result, performance might be improved by increasing the number of executors, as the data being processed is partitioned across many machines in the cluster, rather than having a single executor with many cores reading the same fraction of data on a single node [21]. On the other hand, running many small executors with a single core defeats the advantage of parallelism within an executor and subsequently degrade performance due to JVMs communication overhead [12, 13]. Also, having many tiny executors with fewer cores can lead to unnecessary duplication of data [21]. Evidently, this work also considers the effect of increasing JVM executor while keeping the total heap space constant on job performance.

Owing to the heterogeneous cluster nature, slow tasks are not unusual. Spark mitigates straggler tasks/nodes by *speculative execution*. This works by rescheduling tasks whose run time exceeds the median time of tasks that has already been completed to the scheduling queue. This might not be effective as tasks might still be resubmitted to the slow nodes as there is no way of identifying such nodes to prevent the task scheduler from submitting tasks to them [60]. Also, the effect of slow tasks on application performance could also be mitigated by increasing the number of cores allocated to a single task [44]. Similarly, this study investigates the effect of speculative task execution on the application processing time.

Identifying the key configuration settings for fast job execution (particularly for the applications being considered) as well as for optimal resource utilization is nontrivial because Spark provides a vast number of configuration settings regarding *application properties, execution environments, resource utilization, tasks scheduling, memory management, compression and serialization, shuffle behaviour and Spark UI.* The right combination of configuration parameters with respect to the different aspects of the programming framework varies depending on the different applications Spark supports. The operational modes represented in the *flowerCounter* and *imageClustering* applications requre optimal use of cluster resources especially CPU and memory. This informs the choice of configuration parameters considered in the study.

Motivated by the objectives highlighted above and the work presented by Nguyen *et al.* [38], the key configuration settings investigated in this work are as follows:

- **spark.files.maxPartitionBytes** - This is a type of execution parameter that largely affects CPU usage. It is used to control the size of data in each partition and thus to increase/decrease the number of tasks (Equation 1.1) for effective parallelism and optimal CPU usage. The number of tasks is calculated based on the input data size and the block size provided by this parameter. The default block size of 128 MB is the same as the block size provided by the distributed data storage. The distributed file system used in this work is the Hadoop Distributed File System (HDFS).

$$Number_of_Tasks = \frac{Input_Size}{Block_Size} \tag{1.1}$$

- **spark.executor.cores** - This is also an execution parameter type that controls the number of cores allocated to each executor and thus determines the number of executors in each of the cluster nodes.

The number of cores per executor (Equation 1.2) provided by this parameter is calculated by dividing the total number of cores on a cluster node minus one (as it is mandatory to reserve at least one core for the operating system) by the number of executors desired on the particular node.

$$Number_of_Cores_Per_Executor = \frac{Total_Number_of_Cores_on_Cluster_Node - 1}{Number_of_Executors_on_Cluster_Node} \tag{1.2}$$

- **spark.executor.memory -** This is an application specific parameter which is used to control the size of heap allocated for Spark applications. The size (Equation 1.3) given to each executor is obtained by subtracting 1 GB (as it is mandatory to reserve at least 1 GB for the operating system) from each node's memory and then divide by the number of executors desired on each cluster node. This parameter as well as the different storage levels (discussed in Chapter 2) helps to investigate the caching mechanism of the Spark engine.

$$Size_of_Memory_Per_Executor = \frac{Total_Memory_Size_on_Cluster_Node - 1024}{Number_of_Executors_on_Cluster_Node} \tag{1.3}$$

- **spark.speculation -** This is a pervasive scheduling parameter type that impacts all cluster resources (CPU & Memory amongst others) [38]. This helps to mitigate the impact of slow tasks on application completion especially in a heterogeneous cluster such as the one used in this study.

In summary, this work provides answers to the following questions:

- How does Spark perform in comparison to the sequential versions of these applications?
- What is the effect of partition size on job completion determined by the *spark.files.maxPartitionBytes* parameter?
- What is the impact of JVM executor scaling on job performance controlled by the *spark.executor.cores* configuration setting?
- Does increasing memory allocated to Spark by adjusting the *spark.executor.memory* parameter improve performance or what is the influence of using the different storage levels to cache the input data on application execution speed?
- How does Spark scale with increasing computing nodes in this context?
- What effect does speculative execution (triggered by the *spark.speculation* parameter) due to straggling nodes has on job completion time?

1.1 Thesis Motivation

The processing capability of cluster computing resources is an important and critical component for the processing of large scale data as well as the robustness and effectiveness of the distributed software systems that run on these cluster resources. The speed and efficiency of data processing computations is largely dependent on the processing power of the cluster resources both software and hardware alike. The demand for information and actionable insights from data by practitioners and other stakeholders make having a fast and robust data analytics framework necessary for continuous business growth, innovation and increased revenue to meet the escalating low-latency applications. This is the key premise upon which the motivations for this study solidly rested.

The first motivation for this work was the need to investigate the performance of scale-out analytics frameworks such as Spark for image processing applications as opposed to the single-node scale-up system currently being used for processing. In a scale-out systems, upgrades are done by adding more nodes to the cluster (horizontal scaling) while in a scale-up systems upgrades necessitate increasing computing resources (CPU, Memory, Disk & Network) in a single-node system (vertical scaling) [31, 18].

Apache Spark is a fast analytic engine designed to address some of the growing application use cases. However, the need to reduce the impact of computation bottlenecks for better processing time and data locality has led to a number of Spark improvements because that the CPU has been identified as the major impediment to faster analytics by the Spark engine compared to other factors [40]. The latest version of Spark[5] (the Spark optimizer and the execution engine), has been claimed to have undergone effective performance improvement techniques for better utilization of the CPU and other resources to reduce computation time [16].

Evidently, the performance improvement implemented in Spark for fast analytics provided another impetus for this work as image processing applications are computationally expensive and time-consuming to run sequentially. It is believed that this work would provide meaningful justification in terms of computation cost/time and scalability concerns for processing images in a Spark cluster of commodity computing machines.

1.2 Thesis Statement

The default configuration settings of Spark do not always yield optimal performance for different workloads and determining the right combination of configuration parameters for workloads can be very confusing due to myriads of configuration parameters available [5]. For optimal workload performance, especially in a heterogeneous cluster environment, knowing the influence of each of the configuration parameters on job performance metrics is key in selecting the right combination of settings for fast job completion. Therefore,

[5]http://spark.apache.org/news/spark-2-1-0-released.html (Accessed: April 22, 2018)

the goal of this work was to be able to provide answers to the following questions:

- Would the use of a selection of configuration parameters permit the reduction of job execution time in a predictable manner?
- Would the inspection of low-level performance characteristics and job metrics help guide system architects in identifying bottlenecks as well as developing optimization strategies to improve resource utilization and job execution time?
- Would it be possible to mitigate stragglers effect(s) on job execution time?

In order to address the statements above, this work provides insight into the influence of some Spark configuration parameters on the chosen applications.

The first contribution of this study is to determine the influence of data partition size on the completion speed of Spark jobs. Choosing the right data partition is important for fast execution in ensuring that tasks are evenly distributed across all the cluster nodes and thus helps to achieve efficient usage of resources as well as optimal parallelism. Apart from the contribution regarding data partition size, the impact of running tasks with the Java Virtual Machine (JVM) executor process on application speed was investigated. This is necessary as the Spark engine itself runs on the JVM and thus having too little or much executor processes could adversely affect job completion time. The third contribution of this study is to determine the importance of memory allocated for running Spark applications on job completion. Spark is built around memory abstraction, that is, the ability to take advantage of distributed memory and persist computation in it for subsequent processing. This work also studies the scale-out property of Spark with increasing number of computing nodes and compares performance with the sequential version of the application executed on a single machine of similar resource specification(s). The last contribution of this work helps to understand the effectiveness of Spark in mitigating the impact of slow tasks on application performance with speculative execution.

1.3 Thesis Findings

Based on the experiments conducted, here is a summary of the findings discovered:

- Smaller data partition sizes (thus large number of tasks) less than 64 MB did not result in fast job completion time due to communication overheads. For all the data partition sizes considered, 64 MB data size yielded the fastest execution time.
- Numerous tiny tasks limits parallelism within an executor and thus result in inefficient use of cluster resources. On the other hand, tasks with large CPU and memory do not yield fast completion time due to processing delays and threads concurrency. Eleven JVM executors with 4 cores each exhibited better performance than using a single executor with large number of cores for the *flowerCounter* application.

Similarly, for the *imageClustering* application, two JVM executors with 16 cores each outperformed other configurations considered. Also, for both applications, using large number of JVM executors with one core each exhibited the worst performance due to delays caused by communication overhead.

- Spark scales with increasing homogeneous nodes but degrades with heterogeneous nodes for the *flowerCounter* application only on the larger cluster. The scale-out property of spark showed a quasi-linear behaviour up to 9 cluster nodes with the *flowerCounter* application but decreased with 11 nodes, due to the addition of two nodes that have CPU frequency that is half of the other nodes.

- Increasing memory allocated for Spark applications (for example the *imageClustering* application) similarly increased job completion time. The impact of caching the input dataset using the different storage levels shows that the MEMORY_AND_DISK storage level has the fastest execution speed due to tasks with locality levels that made data and the processing code close enough to facilitate speedy execution.

- For the sequential vs Spark applications (*flowerCounter*), the Spark application executed on both the small & large clusters shows 4x and 3x the speed of the sequential application executed on the single machine with all datasets considered, despite similar or better processing and storage resources on the single machine.

- Finally, experiments show that speculative task execution mitigates the impact of slow nodes on job completion time for the *flowerCounter* application but slightly increases completion time for the *imageClustering* application.

1.4 Thesis Organization

The remainder of this thesis begins by describing the background information and related work in Chapter 2. This is followed with details of the experimental methodology used for this work as well as its design in Chapter 3. Next, the discussion of experimental results and their implications embodies Chapter 4. Finally, Chapter 5 concludes this thesis with a summary and a few recommendations for future research.

Chapter 2

Background and Related Work

This chapter contains a review of some existing works to provide a relatively comprehensive understanding of the Data Analytics Frameworks. The review begins with a brief discussion of Big Data Analytics Frameworks as well as an overview of Spark according to the layered Berkeley Data Analytics Stack (BDAS)[1] architecture from the AMPLab in UC Berkeley. This is closely followed by a detailed description of Spark operations and key configuration parameters used in this study. This chapter concludes with a review of previous performance studies of Spark.

2.1 Big Data Analytics Frameworks

Big Data refers to data sets beyond the processing capacity of usual database software systems.[2] *Big Data* is increasing in forms (consisting of raw, structured, semi-structured and even unstructured data), size (data is estimated to increase to Zettabytes (10^{21}) in the nearby future), velocity (speed of the incoming data and its flows), variability, complexity and value [30]. Today, enterprises and institutions are exploring ways of discovering facts and actionable insights from the data to enhance critical business decisions for increased productivity and to provide the needed impetus for birthing innovations. There is a strong need for developing systems that can handle the storing, processing and analysis of Big Data [28]. This has motivated domain experts to discover ways by which new and improved analytics frameworks can be developed to enhance the prevailing demands of deriving value from data. Notable examples of data analytics frameworks include Apache Spark [62], MapReduce [17], Dryad [27], Apache Flink [10] (an open source platform for distributed computation over streams of data), Apache Storm [50] (a distributed real-time directed acyclic graphs computation system), Apache Samza [39] (a stream processing engine that uses Apache Kafka [58] and Apache Hadoop YARN [53]) and DataFlow [1] among others. These frameworks allow the expression of parallel computations with a set of high-level operators without expensive data movement in a fault tolerant manner [63].

[1]https://amplab.cs.berkeley.edu/software/ (Accessed: April 21, 2018)
[2]https://en.wikipedia.org/wiki/Big_data (Accessed: April 21, 2018)

2.2 Spark Overview

Spark is a general unified analytics engine designed to be fast and easy to use. It leverages distributed memory abstraction to store intermediate data results for later reuse by employing a data abstraction model called Resilient Distributed Datasets (RDDs) [62, 63]. These are read-only data objects that are distributed, scalable and fault-tolerant. They are created through operations (either lazy transformations or actions) on data in persistent storage or from other RDDs. The ability of the Spark engine to persist intermediate computational results in memory makes it particularly apt for iterative applications such as machine learning and graph algorithms. These algorithms work on filtered data sets that can easily be incorporated into the memory of cluster commodity servers.

Most current data analytics frameworks like Hadoop MapReduce [17] have not performed well with these new classes of applications. This is because most of the frameworks were built using the acyclic data flow model and as such cannot handle applications such as iterative and graph processing well. They are designed specifically for handling applications such as batch processing; for them to be able to handle this new class of applications, job output has to be written to external systems causing expensive I/O overheads due to data replication and serialization ([63, 62]).

The Berkeley Data Analytics Stack (BDAS) provides the following characteristics for Big Data applications:

- Low latency queries: Allows applications to run faster,
- Sophisticated analysis: Handles new classes of applications,
- Generalized analysis: Suited for all kinds of analysis including batch, iterative and interactive analysis, and
- Interoperability: Allows different analytics workloads to inter-operate.

BDAS is a layered analytics software framework consisting of different components built to seamlessly interact from the ground up to make sense of the emerging applications and the demanding use cases emanating from Big Data. The majority of the components within the stack emanated as autonomous projects intertwined by similar technologies to form a complete general solution stack to allow different frameworks to inter-operate [20]. The architecture is shown in Figure 2.1. The solution stack adapted for this study includes the in-built Spark standalone manager (resource manager), HDFS (storage medium), Spark core engine (processing engine) and the MLib (K-Means algorithm in the Machine Learning Library built on top of the Spark core).

At the core of the BDAS framework is the Spark in-memory data analytics engine for faster data computation analysis and low latency applications. It is an in-memory general-purpose analytic engine that supports iterative machine learning algorithms, interactive data analysis as well as the conventional batch processing operations in a scalable and efficient manner. Spark is a fast cluster computing paradigm that

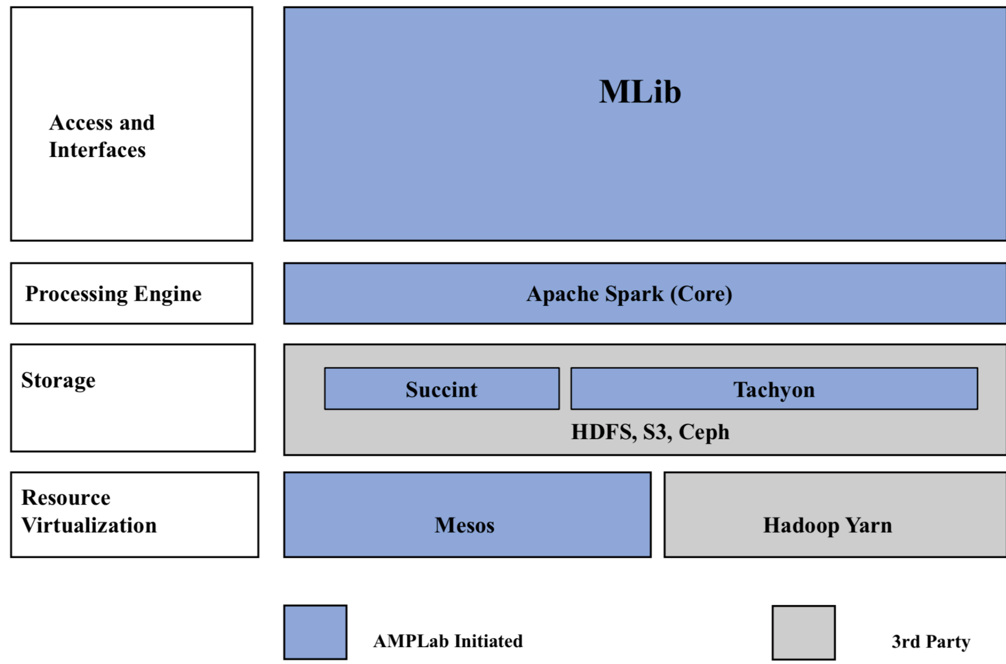


Figure 2.1: The Berkeley Data Analytics Stack (BDAS) [3]

exposes rich Scala and Python Application Pragramming Interfaces (APIs) for programming. These APIs facilitate programming at a much higher level of abstraction compared to traditional approaches. One of the main motivations for developing Spark was to allow distributed programming of Scala collections or sequences seamlessly. Spark supports a variety of data sources such as the HDFS [47], HBase [56], Casandra [32], Amazon S3[3] and other Hadoop-supported data sources.

A Spark application consists of five major components: a driver program, a cluster manager, workers, executors and tasks as shown in Figure 2.2. These components are briefly discussed as follows:

- Driver Program - A driver program (Spark application) is an application written using any of the supported Spark APIs - Scala, Python, Java or R. At a high level, a spark application consists of SparkContext (the main entry point to a Spark application) and the user code. RDDs are created by the interactions between the SparkContext and the user code which are then translated into a *Directed Acyclic Graph (DAG)*. The *DAG* of actions and transformations represents (the execution plan) the different RDD operations and the dependencies between them. There are two types of dependencies: *narrow* and *wide* dependencies. In *narrow* dependencies (for example map, filter and union transformations) each parent RDD partition is needed by at most one child RDD while in *wide* dependencies (for example join and groupByKey transformations) the parent RDD partition is needed by multiple child partitions. The RDDs translated into DAG are then submitted to the *DAGScheduler*. The *DAGScheduler* splits the DAG into stages of tasks and then submit the tasks to the *TaskScheduler*.

[3]https://en.wikipedia.org/wiki/Amazon_S3 (Accessed: April 23, 2018)

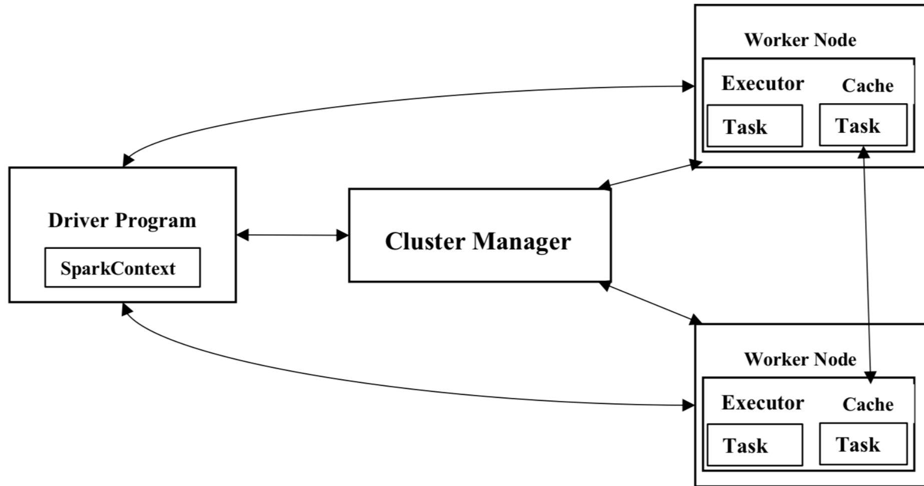


Figure 2.2: Components of a Spark Application [49]

The *DAGScheduler* also determines the preferred location for tasks. Pipelined transformations (that is the transformations that do not require shuffling - narrow dependencies) are grouped into one stage. The boundary of each stage is a shuffle operation. The *TaskScheduler* is responsible for sending tasks to the cluster, retry them if there are failures and also mitigate straggling tasks. The driver program also contains a scheduler backend mechanism that is an interface for plugging in different implementations of cluster managers such as Apache Mesos, Hadoop YARN, Spark standalone and Spark local.

- Cluster Manager - A Cluster manager is the resource Spark uses to acquire resources for executing the driver program jobs. The three cluster managers designed to work with Spark are the Mesos, YARN and the Spark standalone manager.

- Worker - A worker provides compute resources such as the CPU, memory and storage resources for executing a Spark application. Each worker node manages one or multiple ExecutorBackend process(es) and each ExecutorBackend process launches and manages executor instance.

- Executor - An executor is a Java Virtual Machine (JVM) process created by Spark on each worker node for an application. Each executor maintains a thread pool in which each task runs as a thread.

- Tasks - This is the smallest unit of work sent by the *TaskScheduler* to an executor which performs computations and returns results either to a driver program or divides its output for shuffle. Tasks in Spark include the Shuffle Map Task (Shuffle Write Operation), the Shuffle Reduce Task (Shuffle Read Operation) and the Result Task. The Shuffle Map Task writes its output in a shuffle file to disk (SPARK_LOCAL_DIRS). The Shuffle Reduce Task pulls shuffled data over the network and applies reduce logic. The Result task sends output to the driver.

2.2.1 Spark Resource Managers

This is the resource virtualization layer of the BDAS. A cluster resource management system is an important component in a cluster of computing servers for providing analytics frameworks efficient share of resources and scheduling their applications in a distributed fashion. Such a system generally consists of a master process on a node within the cluster that manages slave daemons on the other nodes and frameworks (such as Spark in this scenario) that run tasks on the slave nodes. A framework that runs on any cluster resource manager usually contains two key components: a scheduler to be offered resources by the master process and an executor that runs the framework's tasks on the slave nodes ([22, 25]).

Spark is designed to work with two resource managers namely Mesos [25] and YARN [53], in addition to the built-in Standalone resource managers within the core engine. The Standalone resource manager is used in this study. This is to avoid unnecessary overhead that could be introduced by using resource managers that were not inherently built in the Spark engine.

2.2.2 Spark Storage Systems

Batch, iterative and interactive operations on Spark requires that data should be accessible to all the nodes within a cluster to ensure data locality based on the BDAS. The distributed nature of these operations require that data must be stored using systems that allow data sharing across a cluster of commodity servers without costly overheads due to data movement. Such systems are referred to as Distributed File Systems. They are designed to store and provide fast access to large datasets achieving scalability and fault tolerance by replicating data across cluster nodes to mitigate possible data loss [46].

Spark works with Distributed File Systems such as Hadoop Distributed File System (HDFS) [47], Cassandra [32], and Alluxio (Tachyon) [33]. This study uses only the HDFS as the distributed file storage system because additional database functionality is not needed.

2.3 Spark Operations and Parameters

To meet the challenges imposed by the new set of emerging applications, a new abstraction data called the Resilient Distributed Datasets (RDDs) was implemented in Spark to enable data persistence in memory [62, 63]. An RDD is a collection of immutable and fault-tolerant data objects which can be divided into multiple partitions for parallel operations on different machines within a cluster. RDDs are fault-tolerant parallel data structures that allow users persist data in memory. To achieve fault-tolerance, RDDs provide interface based on coarse-grained transformations which allow them to achieve tolerance by registering RDD transformations rather than the actual data such that RDD lost can be quickly recovered without data movement consequences [62, 63]. RDDs operations are of two types: RDD transformations and actions. The transformations, for example map, filter and flatmap operations, are lazy in that they do not return any value

when executed while the actions, such as count and collect operations, return a value to the driver program when called upon.

RDDs enable the fast processing of workloads by allowing users to persist RDDs for later reuse. RDDs are persisted in memory by default but can be spilled to disk if the available memory is insufficient. Priority levels can also be assigned to RDDs for specifying which RDD should be spilled to disk first. Spark provides different storage levels for persisting or caching RDDs. Storage levels are flags used for controlling how RDDs should be stored. RDDs can be stored in memory or on disk or in memory in serialized format.

Data locality [7] is crucial to application performance. It specifies the proximity of data to the processing code. RDDs also help to achieve effective data locality placement strategies by ensuring that data are placed close enough to where they will be processed with the interface *preferredLocations(p)* based on the number of partitions for better performance. Scheduling design in Spark is built on the notion that computations are usually faster if the data and the operating code are together as it is cheaper to transport serialized code than data chunks. The locality levels based on data location implemented in Spark are highlighted thus (from closest to farthest):

- PROCESS_LOCAL - This level places data in the same JVM process running the code. It's the fastest locality level.

- NODE_LOCAL - This is the level when data is on the node as the processing code. For example, when the Spark node is also an HDFS datanode or when data is on another executor in the same node. It is slightly slower than the PROCESS_LOCAL level as it requires data movement between processes but faster than the other locality levels.

- NO_PREF - This level ensures that data is accessed equally at the same time without any preference to where the data is located. This level gives no preference to data location.

- RACK_LOCAL - This is the level for data that resides on a rack hosting several servers. Thus, it requires data transfer across network via a switch.

- ANY - This means that data is on the network but on a different rack.

Spark usually prefers to schedule tasks with the PROCESS_LOCAL level but when this can not be satisfied, it waits until a busy CPU is free for scheduling or schedules task at a more remote or distant location that requires data movement.

RDDs can be partitioned across cluster nodes to ensure working with reduced data set size. Data partitioning in the context of distributed systems is the dividing of large data sets into logical chunks for parallel processing over a cluster of computing machines. Partitioning is important for distributed data processing to reduce network I/O overhead of distributed operations as Spark achieves locality by placing data and serialized codes close enough to the worker nodes for processing.

Spark uses RDD partitions to store data read from distributed data storage such as HDFS [47] or Cassandra. The degree of parallelism in a Spark application is based on the RDDs partition number. The number of partitions is dependent on how the RDD was created. RDDs are created either over files read from distributed or local storage or through a parallel collection of data objects.

For RDDs created over files stored in HDFS for example, the number of partitions will be equal to the HDFS blocks (a block is 128 MB by default in HDFS). The number of partitions is equal to the number of tasks that will be computed by the worker machines. There are two key RDD operations as stated above - transformations (these create a new dataset from an existing one for example *map, filter, mapPartitions, groupByKey and reduceByKey)* and actions (these return the result of computed pipelined transformations such as *collect, count, saveAsTextFile and takeSample*). Here are some key terms in the physical execution model of a Spark application with respect to the RDD.

- Jobs - These are parallel computation of tasks that are materialized by Spark action RDD operations. For example, assuming a Spark's application logic contains the following RDD operations:

  1. Load a file from HDFS containing words into RDD1 (RDD1 created with partitions automatically based on the default HDFS block size)
  2. Load another file from HDFS with words into RDD2 (RDD2 creation)
  3. *Join* RDD1 and RDD2 to form RDD3 (a transformation operation)
  4. Split up RDD3 into distinct words with *flatMap* to form RDD4
  5. Transform each word in RDD4 and count unique occurrence with *map* and *reduceByKey* transformations to get each word frequency - RDD5.
  6. Save RDD5 to a text file using *saveAsTextFile* transformation.
  7. Split up RDD1 with *flatMap* to get RDD6
  8. *Count* the number of words in RDD6
  9. Split up RDD2 with *flatMap* to get RDD7
  10. Transform RDD7 and count unique word occurrence with *map* and *reduceByKey* operations - RDD8
  11. *Collect* RDD8 and print the result to stdout.

  The entire set of steps form the operations represented in a Spark application. Saving the frequency of occurrence of each word to file in step 5 constitutes a job. The same is true for count and collect operations in steps 8 and 11 respectively.

- Stages - These are series of work within a job corresponding to one or more pipelined RDDs. As shown in the example above, steps 1 through 4 are stages containing pipelined RDDs (RDD1, RDD2, RDD3, RDD4) that must occur before the *saveAsTextFile* Job can be materialized. Similarly, stages 1 and 7

are necessary for the *count* job in step 8 to be materialized and also stages 2 and 9 are needed for the *collect* job in step 11. They are primarily computations that produce intermediate results that can be persisted.

- Tasks - A task is the piece of data within a stage that corresponds to one RDD partition, computed on a given executor JVM.
- Worker Nodes - These are the physical machines that run the executors and the tasks.
- Executors - These are JVM processes that run on the worker nodes for the execution of the tasks.
- Shuffle - This is the transfer of data between stages. **Join** operations such as the *join* in step 3 of the example above, **ByKey** operations such as *reduceByKey* and *groupByKey* and **repartition** operations such as *repartition* and *coalesce* can cause a shuffle and thus disrupt the locality of data resulting in more than one stages.

In other words, a Spark application defines a Directed Acyclic Graph (DAG) of RDDs which are operated upon to create new RDDs that refer back to their parents, thereby generating a lineage graph [62, 63]. Action operations force the translation of the lineage graph starting from the final action RDD and works backwards into stages which are further broken down to tasks. These tasks are submitted to the TaskScheduler for onward execution on the cluster machines by fetching input data either from a data storage or using an existing cached RDD. A typical execution flow of the Spark engine is as shown in Figure 2.3. An efficient Spark application requires setting an optimal number of partitions to achieve better parallelism and effective usage of cluster resources. Two key issues might arise when the number of tasks spawned by jobs is smaller than the number of CPU cores available [21]:

- Reduced benefit from entire cluster computing power, and
- Inconsistent data partition sizes resulting in memory pressure and increased garbage collection due to frequent pauses in computation, thus slowing down data processing.

Data within a partition would be spilled to disk often if its size exceeds memory capacity to avoid out-of-memory exceptions. Spilling data to disk results in expensive overheads due to data sorting and disk I/O, thus stalling job's progress. Therefore, in order to get optimal benefit from cluster resources, the number of partitions should be at least equal to the number of CPUs available or greater by a multiple of 2 to 3 [21].

The number of partitions can be increased, for example if files are being read from HDFS, by the use of the *repartition* transformation which triggers shuffle, configuring HDFS *InputFormat* to create more splits or by writing files to HDFS with a smaller block size [13]. Partitions can also be increased by adjusting the Spark configuration parameter *spark.files.maxPartitionBytes*. The upper limit, however, is that tasks should take at least *100 ms* to execute because having too many tasks can increase the overhead of scheduling tasks [55].

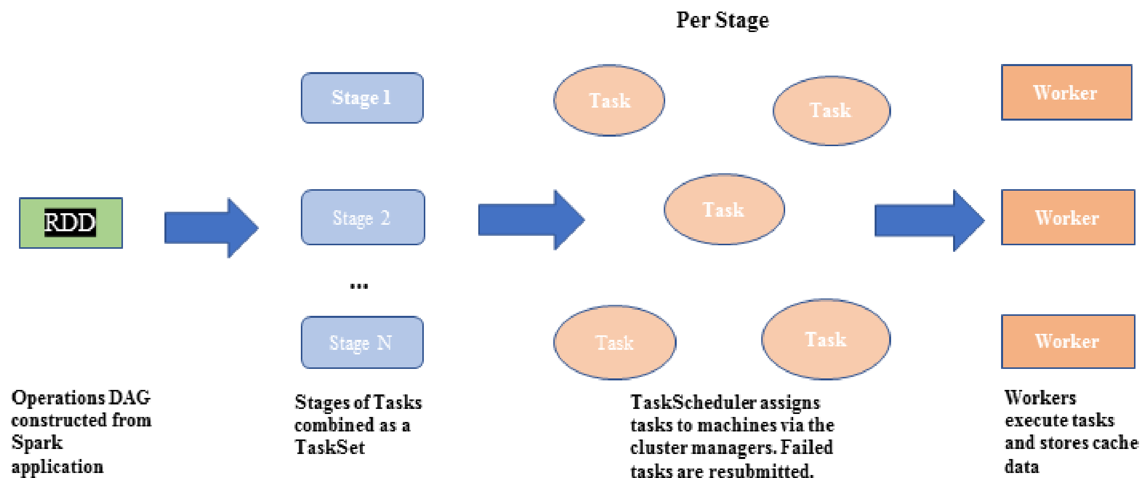


Figure 2.3: The Spark workflow

Data files are scattered in partitions across the cluster by the use of partitioners. In Spark, there are two types of partitioners implemented, namely the *HashPartitioner* (this is the default in Spark used for pair RDDs and used in this work) and the *RangePartitioner* (this partitioned sortable records equally by range). Data shuffling can be avoided if RDDs are partitioned appropriately [21]. Shuffling [6] of data is very expensive as it involves data sorting, repartitioning, serialization and deserialization, compression to reduce I/O bandwidth and disk I/O operations.

In Spark, unlike Hadoop, the reduce phase does not start until all map tasks have finished. Each map task writes its output in a shuffle file to disk as shown in Figure 2.4. This is the shuffle write operation. As there might be a lot of map tasks, the number of shuffle files can be substantially large. The effect of the files on computation can be reduced by enabling the Spark configuration parameter *spark.shuffle.compress* for compression. The files written to disk are then read via the network in the reduce phase during the shuffle read operation [21].

Another importance of RDDs is that they help to mitigate the effect of slow nodes in clusters by running backup tasks. This benefit is achieved by a technique called speculative execution. The following steps show how the speculative algorithm works:

- First looks if the amount of finished tasks in a stage is greater than speculation quantile multiplied by the number of total tasks in the particular stage. If this is true, speculation execution will take place. Speculation quantile is the percentage of tasks which must be completed before speculation can be enabled in a particular stage.

- A scan of all successful tasks in the stage will be done to calculate the median time of tasks execution.

- A threshold for relaunching of slow tasks is then calculated by multiplying the speculation multiplier with the median time previously calculated. Speculation multiplier defines how many times slower a

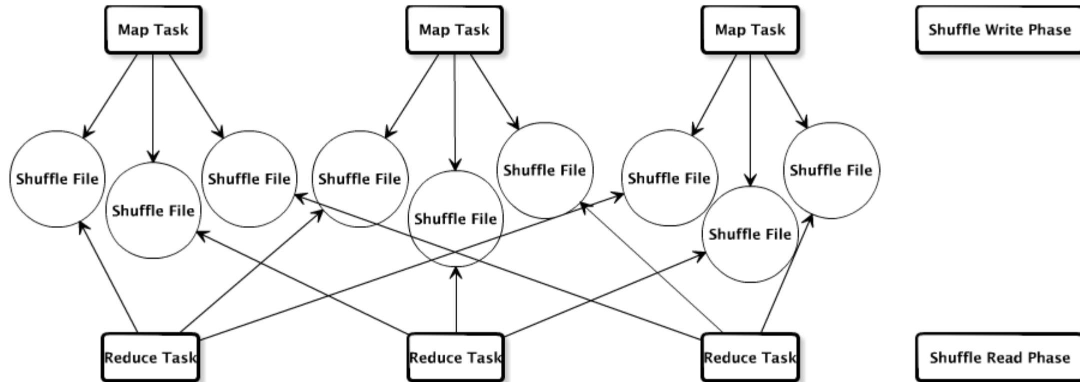


Figure 2.4: The Shuffle Phase

task is, with respect to time, than the median time to be considered for speculation.

- Then relaunch tasks whose running times are greater than the threshold.

Spark speculative execution, rather than helping to reduce the effects of straggling tasks, might sometimes increase job execution time. This is because the speculative execution is based on the median time without taking into consideration that slow tasks might be resubmitted to the straggling worker nodes [60].

Spark provides a number of configuration parameters regarding application properties, execution environments and resource utilization among other parameters as exemplified in Chapter 1. These parameters are used in controlling application behaviours for optimal performance. The configuration parameters used in this study are shown in Table 2.1 together with a short explanation of each parameter alongside the corresponding default values. The detailed description of the key parameters (including *spark.executor.cores, spark.executor.memory, spark.files.maxPartitionBytes & spark.speculation*) investigated is discussed in Chapter 3.

Table 2.1: Spark Configuration Parameters

Parameter Name	Description	Default
spark.driver.cores	Number of cores to use for the driver process; cluster mode only	1
spark.driver.memory	Amount of memory for the driver process	1 GB
spark.executor.memory	Amount of memory per executor process	1 GB
spark.executor.cores	Number of cores to use on each executor; allows an application to run multiple executors provided there are enough cores on that worker	1 in YARN mode and all available cores in standalone and Mesos
spark.local.dir	Directory on disk for storing map output files and RDDs	/tmp
spark.default.parallelism	Default number of partitions to use for shuffle operations such as reduceByKey when not set by user	The value depends on the cluster manager
spark.rdd.compress	Whether to compress serialized RDD partitions	false
spark.shuffle.compress	Whether to compress map output files	true
spark.shuffle.file.buffer	Size of the in-memory buffer for each shuffle output stream; reduces the number of disk seeks and system calls	32 KB
spark.io.compression.codec	Codec used to compress internal data such as RDD partitions, broadcast variables and shuffle outputs	lz4
spark.broadcast.compress	Whether to compress broadcast variables before sending them	true
spark.files.maxPartitionBytes	The maximum number of bytes to pack into a single partition when reading files	134217728 (128 MB)
spark.eventLog.enabled	Whether to log Spark events, useful for reconstructing Web UI after the application has finished	false
spark.eventLog.dir	Directory for logging Spark events	file:///tmp/spark-events
spark.speculation	Specifies whether tasks should be relaunched if they are running slow	false

2.4 Prior Performance Studies of Spark

A number of performance evaluation studies of iterative operations and other operations on data analytics frameworks have been done. Among these studies, Nguyen *et al.* [38] investigated the influence of configuration settings on application performance using different application workloads and thus developed a framework capable of identifying key configuration parameters that affect job performance.

Ousterhout *et al.* [42] argued that the architecture of existing analytics frameworks (such as Spark) makes reasoning about performance very challenging. This is due to the fine-grained pipelining paradigm inherent in such analytics frameworks. Pipelining ensures better performance by utilizing cluster resources for task execution but still requires efficient tuning for concurrent use of cluster resources and thus results in problems such as non-uniform resource usage by tasks, resource contention by concurrent tasks among others. To address these problems, the authors proposed a new architecture for analytics frameworks that makes performance bottlenecks easier to understand and identify and the resulting implications of the frameworks' behaviours. The new architecture requires that jobs are partitioned into executable units called *monotasks* with dedicated resource scheduler that ensures that each *monotask* uses a single cluster resource fully and thus helps to avoid contention by queuing *monotasks.*

Due to the fast adoption and rapidly growing community of Spark enthusiasts, researchers and developers, big data analytics benchmarks have been developed to provide guidance and deployment strategies on how best to take advantage of the frameworks such as Spark. Notably among the benchmarks, specifically designed with the Spark programming paradigm, is SparkBench [34]. SparkBench is a suite of comprehensive workloads capable of evaluating (by providing configuration settings for optimal performance across different workload types as well as resource usage patterns) all the access and interfaces components built upon the Spark engine raging from iterative workloads to graphical applications as well as interactive and streaming workloads.

Veiga *et al.* [54] compared the performance of Hadoop, Spark and Flink using workload benchmarks including PageRank [19] and K-Means clustering among others and revealed that Spark outperformed Hadoop and Flink with respect to scalability in general across all the benchmarks considered especially for K-Means. The work also showed that HDFS block size (partition size) most suited for Spark workloads is 64 MB. The authors again revealed that one big executor/worker with 8 cores is the best configuration for Spark workloads except for PageRank. This might not be absolute because the maximum number of cores in the machine used for the study was 8. Therefore, more study is required to fully understand the influence of cores per executor on Spark applications. This finding that one big executor per node is the optimal configuration for Spark applications is also in resonance with the study done by Li *et al.* [34]. However, this finding contradicts Chiba *et al.* [12] as that suggests that workloads with two or four executors per node achieve the better performance.

Container technologies, such as Docker [37], are gaining traction nowadays, largely due to the adoption in large scale cloud computing. This increasing use in cloud computing is as a result of fast boot/start up

time and very low memory requirement that Docker containers provide. However, their performance vis-a-vis big data applications such as Spark still remains unclear. Ye and Ji [61] made effort to understand the performance of Spark applications in Docker containers and identified that the non-linearity performance behaviour exhibited by the applications is due two key problems: *configuration parameters* and *resource contention between multiple containers*. The authors thus created a performance prediction model for Spark applications in Docker containers with predictability of over 90%. A similar work provided a proof-of-concept architecture for scaled-up (the addition of more compute resources to a single-node system) Spark-based servers by dynamic partitioning of Docker-based containers into logical volumes to improve scalability performance of Spark applications [31].

Ousterhout *et al.* [40] investigated the veracity of the belief that the major bottlenecks in data analytics frameworks are the network, the disk and stragglers (stragglers are slow nodes that substantially prolong job execution time) were the three notions investigated. The study was done using the Spark core analytics engine with a developed Blocked Time Analysis methodology that measures how long jobs spent blocked on cluster resources, with two benchmarks and industry workloads. Results obtained contradict the aforementioned claims about bottlenecks in data analytics frameworks. The results revealed that network improvements and disk I/O do not substantially impact performance in analytics frameworks but, rather most jobs are blocked in the CPU. According to the researchers, the study is relatively inconclusive as the work does not cover a wide range of workloads; the work showed that the widely held belief about performance in data analytics frameworks is absolutely untrue and thus requires that much work to be done before the global information community can claim to understand performance in data analytics frameworks holistically.

Hadoop and Spark were assessed using PageRank iterative workloads for memory utilization and speed using both real and synthetic datasets [23]. Experimental results show that for time-sensitive applications, Spark usually outperforms Hadoop as long as there is enough memory for the computation. This is because Spark is memory-intensive and as the number of iterations increases, intermediate results can quickly fill up the entire memory space which would eventually thwart the secure speed advantage of the Spark engine. This is challenging because it is difficult to ascertain the amount of memory that would be required for a particular iterative operation. The amount of memory that would be required is dependent on the particular iterative algorithm and the size of the input dataset. On the other hand, Hadoop is the preferred choice for less-memory-sensitive applications or when there is insufficient amount of memory for storing intermediate data results, provided there is enough disk space to take in the original dataset and the intermediate results [23].

A performance prediction model [59] was developed recently to evaluate Spark workloads with respect to the execution time, the memory utilization and the I/O cost using both iterative and non-iterative applications including the WordCount, Logistic Regression, K-Means clustering and the PageRank on a cluster of 13 nodes. The prediction accuracy from the model was found to be relatively high for the job execution time and the memory utilization but varied depending on the different applications for the I/O cost. According to the

paper, this might be due to the fact that the prediction performance model was simulated on a small scale.

Work-load driven performance measurement [8] of the Spark engine implemented on a modern-scale up commodity cluster of servers showed that the work time inflation (extra time plus CPU time spent by job threads) and threads load imbalance (when one or few threads require more CPU time than other threads) are the major factors inhibiting workload scalability. At the micro-architecture level, the DRAM latency (memory bound latency) was the main cause of work time inflation. In particular, scalability analysis was done with increasing number of executor pool threads.

The effect of data partition size and executor core scaling to job completion time in data analytics frameworks have been investigated [51]. However, the characteristics of the applications considered in those works are different from the applications studied here. The impact of memory, caching, serialization, local file systems and SSDs forms the core of the study done in Zhang *et al.* [60] to understand the Spark's system in comparison with SciDB for in-memory scientific data analysis. Generational garbage collection, multi-threading and executor scaling influence on TPC-H queries using the Spark engine have also been studied for their effects on job performance [12]. Straggler tasks effect in a heterogeneous environment has also been studied using improved version of the Spark's speculative execution algorithm [60]. Speculative execution of Spark tasks was shown to be ineffective as it does not accurately determine straggling tasks due to slow nodes. HAT [11] is an optimized MapReduce Scheduler that mitigates the impact of slow tasks in heterogeneous environments by using historical information of already concluded tasks to detect slow tasks and then scheduled them accordingly on respective map and reduce slow nodes.

2.5 Chapter Summary

It is evident from the related works reviewed that more study is required for a comprehensive understanding of the influence of configuration parameters on Spark workloads. This is because most of these works provided conflicting results that are insufficient to be generally applicable for all Spark related workloads. It therefore implies that performance results from the applications considered in this study might not yield similar results as in the related studies reviewed.

However, these works are quite representative in that they provided insights on the influence of Spark configuration parameters as well as their effects on different application workloads. These studies provided guidance on the critical configuration parameters to investigate for their influence on applications processing and helped to reduce the time spent on designing experimental methodologies. The findings garnered from these studies also revealed the key performance metrics to monitor as well as their calculations for comprehensive understanding of Spark's performance regarding the applications considered in this study.

Prior performance studies of Spark are crucial to this study because they provided useful insights about the performance of analytics jobs in frameworks such as Spark. The studies revealed insights on performance studies of processing different applications on analytics frameworks including Spark by identifying perfor-

mance bottlenecks and optimization strategies for mitigating their effects. The studies also offered ideas on areas where future works can be concentrated.

Chapter 3

Experimental Methodology and Design

3.1 Experimental Methodology

This section outlines the characteristics of the applications considered for this study with brief discussion on how each of the applications works. It then further describes the nature of the datasets used. A description of the experimental set up and the performance characteristics of the cluster nodes follow. Finally, the section discusses the metrics of interest together with the tools and techniques that were used for collecting them.

3.1.1 Applications Benchmark Characteristics

The two applications selected are *flowerCounter* and *imageClustering*. These applications are written with the Python APIs (Pyspark) implemented in Spark. They also use other external libraries including OpenCV [43], numpy [57], scikit-image [52], and Python Image Processing (PIL) [35] library. The *flowerCounter* application, which estimates the number of flowers on images collected from plot fields, was selected to explore how it can be adapted to run faster in a scale-out Spark server cluster and then compare its performance with the sequential version of the application run on a single machine. Another reason for choosing the application is to study how it scales with increasing computing machines of varying capacity and datasets for future analyses. At a high level description, the *imageClustering* application clusters images using the k-Means clustering algorithm implemented in MLlib [36] library built on top of the Spark core engine based on features in the images with varying brightness or colour. Again, the choice of this application is to investigate the speed and scalability properties of the Spark engine for the clustering of the images relative to the performance of the sequential version of the same application processed on a single machine.

The *imageClustering* application is more amenable to the Spark programming framework than the *flowerCounter* application. This is because the application is iterative in nature and uses the k-Means Machine Learning Library (MLlib) clustering algorithm of Spark as opposed to the external libraries used in the *flowerCounter* application. Also, the *imageClustering* application uses Spark's read-only broadcast variables which help prevent transferring data objects multiple times to the executors for processing. This is particularly computationally efficient for applications whose tasks require large values to prevent unnecessary network overhead due to data transfer.

Apart from the application properties described above, the applications are also relatively representative in

that they cover a diverse set of Spark lazy transformations and actions with both narrow - where each parent RDD partition is needed by at most one child RDD (for example map and filter transformations) and wide - where the parent RDD is needed by multiple child partitions (for example reduceByKey transformation) lineage dependencies. Table 3.1 shows the applications together with the inherent transformations, actions and the dependencies. The meaning of the transformation and action operations contained in the applications is explained briefly as follows:

- map - Returns a new image RDD by passing each source image through a function such as the *computeHistogram*, and the *computeHistogramShifts*. These functions are described in below.
- coalesce - Used for reducing the number of partitions before writing output to file.
- collect - Returns all the elements in an array for use in subsequent stages in the application pipeline.
- first - Returns the first element in an array for use as a parameter in the *computeHistogram* function.
- saveAsTextFile - Writes the computed flower estimate for each image to a HDFS path.
- flatMap - Used for extracting features from each of the images and returns the extracted features for each image mapped to the corresponding image name as in the input image source.
- filter - Used to select only extracted features of interest and then returns new image features RDD that satisfies the specified condition.
- mapPartitions - Returns new image RDD partition by passing each source image partition through a function.
- reduceByKey - Returns new key-value pairs for each image RDD with the value aggregated using the given reduce function.
- collectAsMap - Returns new key-value pairs for each image RDD to the master as a dictionary
- takeSample - Returns an array with a random sample of the number of images in the source images RDD.

Also, here is a brief description of the details of each of the applications considered:

- **flowerCounter** - This application basically contains four main parallelizable stages defined with the functions namely ***computeHistogram***, ***computeHistogramShifts***, ***computeFlowerPixelPercentage*** **and** ***computeFlowerCount***. The ***computeHistogram*** stage computes the histograms for all images within a defined plot mask using the Lab Colour Space (b-channel component) and the histograms of the Grayscale[1] images. The function returns the computed histograms representing the plot

[1]https://en.wikipedia.org/wiki/Grayscale (Accessed: April 23, 2018)

pixels for both the grayscale image and the b-channel component of the images. The ***computeHistogramShifts*** uses the average histogram of the images b-channel pixel histograms calculated from the ***computeHistogram*** function. The shifts are calculated by correlating the b-channel histograms with the corresponding average histogram values. The returned value is a dictionary containing the key (representing the image filename) and the corresponding histogram shift value of all the images. The ***computeFlowerPixelPercentage*** takes into consideration the histograms and histogram shifts obtained from the previous steps and determines how many pixels contain flower colours. The ***computeFlowerCount*** then computes the estimate of the number of flowers in each of the images by first highlighting the flowers in each respective image with a logistic transformation on each image using the already computed shift pixel value. Finally, the estimate of the number of flowers is obtained by using the scikit-image Determinant of Hessian[2] (DoH) blob detection algorithm. The computed flower count estimate on each image obtained with the *map* transformation is then saved to disk as a key(image name)-value dictionary object with the *coalesce* and *saveAsTextFile* transformations respectively.

- **imageClustering -** This application clusters the images by first extracting features (these are key points in the images with varying degrees of physical attributes such as colour and brightness) with a *map* function that uses the OpenCV Scale Invariant Feature Transform (SIFT)[3] algorithm to compute descriptors from the images in a partition sequentially. The extracted image feature descriptors are then filtered and mapped to their corresponding image filename as a key-value dictionary object. The features obtained are then used to build a clustering model by applying the K-Means clustering algorithm implemented in the Machine Learning Library component (MLlib) built on top of Spark. For the results shown in Chapter 4, the key API functions inherent in the K-Means model built represented as stages are the *collectAsMap* stages specified in §4.2, particularly in Tables 4.4 and 4.5 on page 63. The *takeSample* stages randomly select images from the input RDD and returns an array of images for use in the subsequent *collectAsMap* stages. Cluster centres are then selected from each partition using the model built and initial position of each feature predicted relative to the cluster centres. Iteratively features were assigned to clusters based on the difference between the actual and predicted position of each feature. The final cluster was then selected based on the most common cluster to which features were matched/assigned. And later the *coalesce* and the *saveAsTextFile* stages (two stages in one) then combined all the results from all the partitions for writing as output to HDFS. For each image, the filename and its corresponding assigned cluster value are then saved to disk.

The execution time of the sequential versions is compared with the Spark version run on a single machine (*onomi*). For the sequential *flowerCounter* execution, only the functions necessary for estimating the number of flowers in the images from the plot field, which are represented in the parallel Spark version, were considered.

[2]https://en.wikipedia.org/wiki/Blob_detection (Accessed: April 28, 2018)
[3]http://docs.opencv.org/trunk/da/df5/tutorial_py_sift_intro.html (Accessed: April 23, 2018)

Table 3.1: Applications Characteristics

Application Name	Transformations	Dependencies	Actions
flowerCounter	map coalesce	narrow wide	collect first saveAsTextFile
imageClustering	coalesce filter map flatMap mapPartitions reduceByKey	wide narrow narrow narrow narrow wide	collect collectAsMap takeSample saveAsTextFile

On the other hand, for the sequential *imageClustering* execution, all the functions inherent are represented in the Spark version.

3.1.2 Input Datasets

The input datasets are still images of canola plants collected with timelapse cameras from prepared field throughout the Summer of 2016. The field is divided into a number of plots. The cameras are named based on respective plot numbers namely camera1108, camera1109, camera1122, camera1207, camera1225, camera1237, camera2103 and camera3102. The total images collected throughout the season was about 275 GB. This study however, focused only on the images collected for specific time periods in the months of July, August and September 2016 with detailed analysis done on the July images especially as these contain the highest number of flowers. The July dataset is about 35 GB. These are images collected from July 1 to July 15 of 2016. In August, there were three sets of datasets collected. The first dataset, which is about 50 GB, was collected from July 15 to August 2. The second dataset of about 60 GB was collected from August 2 to August 15 and the last dataset of about 20 GB was collected from August 26 to August 31. This study used the last dataset of August. The September dataset used is about 6 GB. These are images collected from August 31 to September 12. The detailed description of the images is shown in Table 3.2. All images are in the JPEG format.

It is important to emphasize that the datasets are not large enough in terms of size to be considered as *Big Data*. The datasets were considered as an initial step towards understanding the influence of Spark configuration parameters on the applications. It is expected that the datasets will grow as the deployment of the project continues and such the applications can be scaled accordingly. The described datasets used in this study are enumerated below:

- The July datasets were used to investigate the effect of partition size using the *spark.files.maxPartitionBytes* configuration parameter on job completion time for the *flowerCounter* application as described in §3.2.1.

Table 3.2: Dataset Description

Month	Time Period	Images Size	Number of Images	AverageSize/Image
September	Aug 31 - Sep 12	5.7 GB	12686	449 KB
August	Aug 26 - Aug 31	19.3 GB	39770	485 KB
July	July 1 - July 15	34.8 GB	93708	371 KB

- For the executor JVM scaling experiments using the *spark.executor.cores* configuration setting described in §3.2.2, all the datasets (July, August & September images) were used for both application workloads. There were two sets of experiments: an initial set of experiments done with the September images and a detailed set of experiments conducted using all the datasets.
- The experiments described in §3.2.3 to investigate the effect of the *spark.executor.memory* configuration parameter used all the datasets for both applications but only the results for the *imageClustering* application are reported.
- The scale-out experiments in §3.2.4 were also conducted with all the datasets and likewise for the sequential experiments. The sequential experiments were done for only the *flowerCounter* application.
- The speculative execution experiments to investigate the effect of the *spark.speculation* parameter described in §3.2.5 used all the datasets as well for both applications.

3.1.3 Cluster Setup

There were two clusters environment set up. The larger one was made up of twelve physical nodes hosted using the Kernel-based Virtual Machines[4] (KVM); one node served as the master node and the remaining eleven nodes served as the slaves/workers. The nodes were hosted and connected together in a private network of 1 Gbit/s to ensure undisturbed traffic flows. The specification of each cluster node is as shown in Table 3.3. It is a heterogeneous cluster in that nine of the nodes have CPU speeds that's almost twice that of the other two nodes (luigi & mario). It is important to state that there is a difference between the specification of the physical hardware and the virtual hardware. The processors of the machines in the cluster are of the type *i7-2600*, but the VMs see different processors of the type *E312xx*. On the other hand, the smaller cluster was made up of three physical machines hosted using KVM; one node served as the master node and the remaining two nodes served as the slaves/workers. The smaller cluster is also a heterogeneous one in that two of the nodes have the same processor type and clock speed as opposed to the third node. The specification of each node of the smaller cluster is as shown in Table 3.4.

[4]https://en.wikipedia.org/wiki/Kernel-based_Virtual_Machine (Accessed: April 23, 2018)

Table 3.3: Specification of Each Node of the Larger Cluster

Nodes	Role	CPU Type	Cores	Memory	Disk
192.168.1.211 (mario)	Spark Slave Hadoop Slave	Intel(R) Core(R) CPU i7-2600 @ 1.80 GHz	8	34.53 GB	7.79 T
192.168.1.212 (luigi)	Spark Slave Hadoop Slave	Intel(R) Core(R) CPU i7-2600 @ 1.80 GHz	8	34.53 GB	7.72 T
192.168.1.200 (Master)	Spark Master Hadoop Master	Intel(R) Core(R) CPU i7-2600 @ 3.40 GHz	8	11.46 GB	0.4 T
192.168.1.201 (Worker1) 192.168.1.202 (Worker2) 192.168.1.203 (Worker3) 192.168.1.204 (Worker4) 192.168.1.205 (Worker5) 192.168.1.206 (Worker6) 192.168.1.207 (Worker7) 192.168.1.208 (Worker8) 192.168.1.209 (Worker9)	Spark Slaves and Hadoop Slaves	Intel(R) Core(R) CPU i7-2600 @ 3.40 GHz	8	14.34 GB	7.44 T

The rationale for choosing the nodes in the large cluster was to investigate whether cluster operation with large number of small commodity computers would permit the reduction of the applications processing time. On the other hand, the small cluster was considered specifically to understand the influence of large executor size (in terms of allocated cores and memory) on the applications processing time.

For the sequential setup, the specification of the single machine, *onomi*, is as shown in Table 3.5. The operating system used on the master node (the larger cluster) is Ubuntu 16.04.2 LTS (GNU/Linux 4.7.1-040701-generic x86_64) while on the remaining eleven nodes of the larger cluster and the three nodes of the smaller cluster is Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-71-generic x86_64). The Hadoop version used is Hadoop 2.7.2. The Spark version is Spark 2.1.0. The Scala interpreter version is Scala 2.11.8. The Java version used is 1.8.0_121 Java HotSpot(TM) 64-bit Server VM (build 25.121-b13, mixed mode).

As mentioned in the introduction section of Chapter 1, the specific list of Spark parameters investigated in this study is shown in Table 3.6, other parameters are set to the base value. The critical parameters for each of the experiments in §3.2 are itemized thus:

- *spark.files.maxPartitionBytes* - This is the key parameter that underlies the experiments in §3.2.1.
- *spark.executor.cores* - This parameter is investigated in §3.2.2.
- *spark.executor.memory* - This parameter is the focus of the experiments in §3.2.3.
- *spark.speculation* - The effect of speculative execution on slow tasks is studied in §3.2.4.

Table 3.4: Specification of Each Node of the Smaller Cluster

Nodes	Role	CPU Type	Cores	Memory	Disk
louie	Spark Slave Hadoop Slave	Intel Core Processor Broadwell CPU @ 2.40 GHz	48	377 GB	/dev/vdb 460 GB /hadoop 500 GB
dewey	Spark Slave Hadoop Slave	Intel Core Processor Haswell CPU @ 2.60 GHz	48	377 GB	/dev/vdb 460 GB /hadoop 500 GB
huey	Spark Master Hadoop Master	Intel(R) Core Processor Broadwell CPU @ 2.40 GHz	48	247 GB	/dev/vdb 460 GB /hadoop 500 GB

Table 3.5: Specification of the Benchmark Machine (*Onomi*)

Node	Role	CPU Type	Cores	Memory	Disk
128.233.174.7	Benchmark Machine	Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60 GHz	56	131.92 GB	24.3 T

Table 3.6: Spark Configuration Parameters for Experimental Design

Parameters	Usage
spark.files.maxPartitionBytes	Used to control RDD partition size
spark.executor.memory	Used for controlling executor heap size
spark.executor.cores	Used to vary the number of cores allocated to each executor and also controls the number of executor instances.
spark.speculation	For specifying whether the speculative task algorithm should be applied for slow tasks

3.1.4 Performance Characteristics of the Cluster Nodes

The performance characteristics of the cluster nodes used in this study are discussed here with respect to the following properties:

- Host-bus Speed,
- Cache Size,
- Hyper Threading (HT) Support,
- Turbo Boost Technology (TBT).

The characteristics of each node are as shown in Table 3.7.

Host-bus Speed

There are different types of bus: the Front-Side Bus (FSB), the Direct Media Interface (DMI) bus and the Quick Path Interconnect (QPI). The DMI is a point-to-point interconnection between an integrated memory controller and an I/O controller hub on the motherboard. QPI is also a point-to-point link between the processor and the chipset. All the processors have the same bus speed of 5 GT/s (gigatransfers/second) except for the processor on *onomi* which has a bus speed of 9 GT/s with 2 QPI bus links. Hashwell and Broadwell processors on the small cluster has the same bus link as the processors on the large cluster but with improved DMI bus links [26].

Cache Size

Caching improves performance by persisting frequently accessed data transfers between the processor and the main memory, reducing access latency. Thus, the size of the cache as well as how fast the processor can access the data stored in the cache (the hit rate) is an important factor of processor performance. The size of the L1 (layer 1 - 64 KB) cache is the same for all the processors used in this study. This is because the L1 cache is an intrinsic feature of processor architectural design that cannot be altered. However, the L2 cache can come in different sizes as it is external to the processor. As shown in Table 3.7, all the processors in the small and large clusters have the same L2 cache size of 4 MB. The L2 cache size of the processor in *onomi*, on the other hand, is smaller with a size of 256 KB. Larger caches (for example L2 caches on the small and large clusters and L3 cache on *onomi*) are slower (the speed of the cache reduces down the chain and expensive due to large amount of on-die resources and high power consumption, but can still provide performance benefits especially when there is high cache-miss at the layer 1 (L1) cache level. Larger L2 cache increases processor's performance for applications that operate on small datasets, but only yield marginal benefits for applications involving large datasets [26].

Hyper-Threading Support

Hyper-Threading (HT) is a feature implemented in processors for performing multiple tasks at once. This implementation is advantageous to applications that operate on independent data in parallel as is the case with the applications considered in this study, otherwise it should be disabled. For processors with HT support, one physical core appears as two logical cores to the operating system and thus allowing simultaneous scheduling of two processes per core. All the processors used in this study have HT support including the processor on *onomi*. The processor has two sockets each with 14 cores (28 cores in total). With HT enabled, the 28 physical cores appears as 56 logical cores to the operating system for simultaneous operations in parallel. However, HT (among other factors such as large and slow L3 cache) did not yield much performance benefit. In particular, the *flowerCounter* application, probably, was not written ground-up to take advantage of multi-core processors; it had to be rewritten to allow for multiprocessing. This after-effect application optimization might not have been an effective performance improvement strategy, as many of the processing resources are not heavily utilized. The other machines only have one physical CPU and HT helped to achieve 8 logical cores on respective machines.

Turbo Boost Technology

Intel Turbo Boost Technology (TBT)[5] is an advanced processor's feature that dynamically adjust its operating frequency based on intrinsic demand required by application processes/tasks for optimal performance. This is also another performance determinant factor that might affect application outcome. All the processors (on the physical machines) considered in this study have the TBT capability except for the *Haswell* micro architecture processor in *dewey*.

3.1.5 Measurement Tools and Metrics

The main tool for measuring metrics in this study is the log files as well as through the REST API provided by Spark. The metrics can be accessed either via the Web UI or as JSON for both running applications and applications stored in the History Server. History Server, which extends Spark's Web UI, is the Web UI for completed and running applications. It is used to maintain and visualize event logs of completed and running applications. The Web UI also helps for performance debugging purposes by providing specific information for the applications (statistical information about the execution time of the jobs, stages and tasks) as well as the task progress view for identifying straggler tasks and other purposes. It also provides the amount of input data processed and the amount of output data produced for understanding data distribution on the nodes [48]. However, these tools provided by Spark are not in themselves sufficient for performance study. As a result, an instrumented version of Spark called Sparkoscope[6] as well as some scripts for trace analysis

[5] https://en.wikipedia.org/wiki/Intel_Turbo_Boost (Accessed: March 18, 2018)
[6] https://github.com/ibm-research-ireland/sparkoscope (Accessed: April 23, 2018)

Table 3.7: Performance Characteristics of Nodes

Nodes	Clock Speed	Bus Speed	Cache Size	No of QPI Links	Thread(s)/ Core	Core(s)/ Socket	TBT Support
mario & luigi	1.80	5 GT/s DMI	L1: 64 KB L2: 4 MB	None	1	1	Yes (2.0)
master Worker1 Worker2 Worker3 Worker4 Worker5 Worker6 Worker7 Worker8 Worker9	3.40	5 GT/s DMI	L1:64 KB L2: 4 MB	None	1	1	Yes (2.0)
dewey	2.60	5 GT/s DMI2	L1: 64 KB L2: 4 MB	None	1	1	No
louie & huey	2.4	5 GT/s DMI2	L1: 64 KB L2: 4 MB	None	1	1	Yes (2.0)
onomi	3.5	9.6 GT/s QPI	L1: 64 KB L2: 256 KB L3: 35.8 MB	2	2	14	Yes (2.0)

of Spark jobs were used for collecting CPU, network, memory and disk utilization information in the cluster ([41], [64]).

For all the experiment scenarios, the average metric values are reported from three experimental run instances respectively except for the experiments (with the exception of the partition size vs runtime scenario) discussed in Sections §4.1 and §4.2. Experiments in these sections also involved three run instances but results are reported only for the instance with the least runtime. The three experimental run scenarios considered is to allow Spark's driver to stabilize as the variation between the job completion time remains nearly constant and negligible for the second and third run instances. The large time variation between the first and the last two run instances is due to the start up overhead of the driver process on the master node.

Also, for the experiments in Sections §4.3, §4.4 and §4.5 (with the exception of influence of *spark.executor.parameter* and the comparison between the Sequential and Spark applications), three run scenarios were considered but results presented only for the instance with the lowest job runtime. Thus, the performance metrics of interest across all experiments are as follows:

- Execution Time at the stage and task levels,
- Runtime or job completion time,
- Number Tasks per node,
- Percentage of Average CPU Utilization,
- Percentage of Average RAM Usage,
- Network throughput,
- Disk throughput,
- HDFS I/O.

The execution time (especially the median and the maximum execution time of tasks) is important for identifying slow tasks that might be due to data skew or slow nodes. This behaviour can be further verified by looking at the visual chart that shows the task execution progress for each executor on the Web UI. Apart from the execution time, the percentage of resource utilization (CPU and RAM), the disk and network I/O are metrics that also affect the performance of Spark jobs and thus will be important to understand the impact of these metrics on job completion time.

Furthermore, the time-related metrics (execution and job completion time) highlighted above in addition to other important ones (not stated above) regarding the applications are particularly important for performance evaluation. These other metrics include the *runTime*, *duration* (sum of individual tasks time), *schedulerDelay*, *executorRunTime*, *executorCpuTime*, *executorDeserializeTime*, *resultSerializationTime*, *gettingResultTime*, and *jvmGCTime*. The idea on how to collect these metrics came later in the study and,

therefore are collected only for the experiments (last sets of experiment implemented) investigating storage levels discussed in section §4.3. These metrics are defined as follows to aid their understanding:

- *runTime* (Equation 3.1) - This is obtained as the minimum launch time subtracted from the maximum finish time for all the tasks in all the stages of execution across all cluster nodes given that each task has a start time and a finish time. The *runTime* is given thus:

$$runTime(elapsedTime) = maximumFinishTime - minimumLaunchTime. \tag{3.1}$$

- *duration* (Equation 3.3) - This is the sum total of the *schedulerDelay*, *executorDeserializeTime*, *executorRunTime*, *resultSerializationTime* and *gettingResultTime* for all the tasks in all the stages on all cluster nodes. For a given task t in all the stages of execution on a worker node n, the *duration* or *taskExecutionTime*[7] is calculated thus (Here N is the total number of nodes in the cluster (9 in this scenario) and T is the total number of tasks - 1118 for *flowerCounter* and 7001 for *imageClustering*):

$$\begin{aligned} duration(taskExecutionTime) &= schedulerDelay_{n,t} + executorDeserializeTime_{n,t} \\ &+ executorRunTime_{n,t} + resultSerializationTime_{n,t} + gettingResultTime_{n,t}. \end{aligned} \tag{3.2}$$

$$\begin{aligned} duration(taskExecutionTime) &= \sum_{n=1}^{n=N} (\sum_{t=1}^{t=T} schedulerDelay + \sum_{t=1}^{t=T} executorDeserializeTime \\ &+ \sum_{t=1}^{t=T} executorRunTime + \sum_{t=1}^{t=T} resultSerializationTime + \sum_{t=1}^{t=T} gettingResultTime). \end{aligned} \tag{3.3}$$

- *executorRunTime* (Equation 3.4) - This is the summation of data read/write time from the filesystem (HDFS), CPU execution time and the JVM garbage collection time. The *executorRunTime*[8] across all the nodes is calculated thus:

$$\begin{aligned} executorRunTime = \sum_{n=1}^{n=N} (\sum_{t=1}^{t=T} taskIOReadTime + \sum_{t=1}^{t=T} taskIOWriteTime + \sum_{t=1}^{t=T} executorCpuTime \\ + \sum_{t=1}^{t=T} jvmGCTime). \end{aligned} \tag{3.4}$$

[7]https://www.ibm.com/support/knowledgecenter/en/SSZU2E_2.2.1/performance_tuning/application_spark_parameters.html - (Accessed: March 31, 2018)

[8]https://www.ibm.com/support/knowledgecenter/en/SSZU2E_2.2.1/performance_tuning/application_spark_parameters.html - (Accessed: March 31, 2018)

3.2 Experimental Design

This section highlights the experimental studies that are carried out to meet the objectives of this thesis work. To understand Spark's performance for the image applications, the following experiments were executed:

- Experiments to investigate the influence of *spark.files.maxPartitionBytes* on application performance,
- Experiments to understand the influence of *spark.executor.cores* on application performance,
- Experiments to investigate the influence of *spark.executor.memory* parameter (whether increasing the memory allocated to Spark while the system's physical memory remains constant) on application performance,
- Experiments to study the scalability of Spark system with increasing numbers of computing nodes, that is, to investigate the scale-out property of Spark, and
- Experiments to investigate the influence of *spark.speculation* configuration parameters on straggling tasks and on the overall job performance.

3.2.1 Experiments to investigate the influence of *spark.files.maxPartitionBytes* on application performance

The *spark.files.maxPartitionBytes* configuration parameter is used to control RDD partition size and ultimately for controlling the level of parallelism (the number of tasks). As the right partition size to ensure fair share of data to all the executor is important to avoid data skew and to reduce job completion time, experiments were carried out on the large cluster using the July dataset on the *flowerCounter* application with *spark.files.maxPartitionBytes* parameter varied from the default value (128 MB) up till 2 MB respectively but analysis focused on the 128 MB and 64 MB experimental runs respectively ([51], [64]). The *spark.files.maxPartitionBytes* parameter was varied in multiples of 2 based on previous works to increase the number of tasks (the level of parallelism) and also to reduce tasks size. This was necessary to investigate whether increase in the number of tasks by the parameter would improve resource utilization and reduce application processing time. The *spark.executor.cores* value of 6 was used from the 8 available cores on each node of the large cluster to share evenly for the executors (even number of cores). One core was reserved for the operating system. The configuration settings used for this experiment are as shown in Table 3.8.

3.2.2 Experiments to understand the influence of *spark.executor.cores* on application performance

In addition to Spark being an in-memory computation engine, it is also based on the JVM. Therefore, the effective configuration of the JVM parameters for reduced garbage collection overhead is essential for

Table 3.8: Configuration Settings for Experiments in §3.2.1

Configuration Setting	Base Value							Parameter Type
spark.driver.memory	10 GB							Application
spark.broadcast.compress	true							Compression
spark.rdd.compress	true							Compression
spark.io.compression.codec	lz4							Compression
spark.shuffle.compress	true							Compression
spark.files.maxPartitionBytes	128 MB	64 MB	32 MB	16 MB	8 MB	4 MB	2 MB	Execution
spark.executor.memory	12 GB							Memory Management
spark.executor.cores	6							Execution

optimizing job performance. The recommendation according to Chiba and Onodera is to use either two or four executor JVMs for better job performance as using a single large executor with more than five cores concurrently or many tiny executors could lead to bad HDFS I/O. This is because HDFS does not perform well with many concurrent threads as the number of simultaneous operations it can support is dependent on the number of HDFS blocks stored on the data nodes. Having many concurrent threads could result in cores reading same small fraction of data on a single node and therefore, reduce parallelism due to communication overheads [13, 21].

As a result, experiments were conducted with all the datasets on the small cluster by increasing the number of cores allocated to each Spark's executor JVMs. Thus, varying the number of executors by the *spark.executor.cores* parameter per each run scenario while keeping the heap size constant based on the total memory allocated to the worker machine as shown in Table 3.9. The *spark.executor.cores* parameter, varied in multiples of 2 based on the number of cores available on each node in the cluster, investigated the effect of multiprocessing within an executor on application performance. The *spark.executor.cores* value of 47 (one core left for the operating system) evaluated the influence of executor with many cores on application performance. The *spark.executor.memory* parameter was kept at 7 GB relative to the total memory on each of the nodes in the cluster (377 GB) to prevent halting the applications' execution due to insufficient amount of memory. This is because as the number of executors per node increases by reducing the number of cores per executor, the amount of memory required to execute the applications increases. The configuration settings for these experiments are shown in Table 3.9 with the respective varied values of the *spark.executor.cores* parameter.

For the partition size used for the experiments dictated by the *spark.files.maxPartitionBytes*, the *flowerCounter* application executed on the small cluster used the default 128 MB partition size for all the datasets. However, the *imageClustering* application on the same small cluster used partition size of 8 MB because the execution failed with partition size greater than 8 MB. The failure was due to recurrent HDFS client errors such as *hdfs.DFSClient: Exception in createBlockOutputStream that IOException: Failed to replace a bad datanode on the existing pipeline due to no better datanodes being available to try.* This is because HDFS

Table 3.9: Configuration Settings for Experiments in §3.2.2

<table>
<tr><th>Configuration Setting</th><th colspan="7">Base Value</th><th>Parameter Type</th></tr>
<tr><td>spark.driver.memory</td><td colspan="7">10 GB</td><td>Application</td></tr>
<tr><td>spark.broadcast.compress</td><td colspan="7">true</td><td>Compression</td></tr>
<tr><td>spark.rdd.compress</td><td colspan="7">true</td><td>Compression</td></tr>
<tr><td>spark.io.compression.codec</td><td colspan="7">lz4</td><td>Compression</td></tr>
<tr><td>spark.shuffle.compress</td><td colspan="7">true</td><td>Compression</td></tr>
<tr><td>spark.files.maxPartitionBytes</td><td colspan="4">128 MB - flowerCounter</td><td colspan="3">8 MB - imageClustering</td><td>Execution</td></tr>
<tr><td>spark.executor.memory</td><td colspan="7">7 GB</td><td>Memory Management</td></tr>
<tr><td>spark.executor.cores</td><td>1</td><td>2</td><td>4</td><td>8</td><td>12</td><td>16</td><td>47</td><td>Execution</td></tr>
</table>

cluster with small number of data nodes (3 in this case) may experience failure due to large datasets [14]. The small cluster was only available for a limited amount of time and debugging the failures encountered with the *imageClustering* application was not possible. The same experimental design and explanation regarding partition size hold for the experiments involving *spark.executor.memory* parameter in §3.2.3 with the *imageClustering* application.

Apart from the experiments conducted on the small cluster, preliminary experiments were also performed on the large cluster with the August dataset for the *flowerCounter* application and the September dataset for the *imageClustering* application. September dataset with the default partition size of 128 MB was used for the *imageClustering* application because experiments with the other datasets (July and August) with the same default partition failed due to *out-of-memory* exceptions. Two sets of experiments were performed for the applications. One set of experiments for the *flowerCounter* application used 3 cores with 6 GB of memory per executor on the large cluster with 9 nodes (18 executors, 2 per node). The other experiments used 2 cores with 4 GB of memory per executor on the same large cluster (27 executors, 3 per node). On the other hand, the *imageClustering* application used 6 cores with 12 GB of memory (9 executors, 1 per node) for one set of experiments while the other set of experiments used 3 cores with 6 GB of memory.

For all the experiments conducted here and in other subsections of §3.2, few configuration parameters regarding driver application and data compression were fixed while the other parameters depending on the experiment type were varied accordingly. The fixed parameters include *spark.driver.memory*, *spark.broadcast.variable*, *spark.io.compression.codec*, *spark.rdd.compress* and *spark.shuffle.compress*. The *spark.driver.memory* value of 10 GB was selected for the driver process on the master node where the main entry point, the SparkContext, to the Spark engine was instantiated. This is important for initiating and stopping Spark applications and also for coordinating the interactions between all the processes running on the worker nodes. The *spark.driver.memory* was fixed at 10 GB because large amount of memory is not required by the driver process (the driver process does not participate in the applications processing execution).

The other compression parameters are turned on by default except the *spark.rdd.compress* parameter. Generally, compressing data objects in Spark is important for effective application performance. The *spark.broadcast.compress* helps to compress serialized data objects before they are sent over the network and therefore reduces overhead especially for large data objects. This is particularly important for applications that require broadcast variables such as the *imageClustering* application. Another parameter turned on by default is the *spark.shuffle.compress*, used for compressing intermediate shuffled data objects. This is a key performance parameter for iterative applications (such as the *imageClustering* application) and other applications that make several passes over intermediate data outputs. The RDD partitions were compressed with the *spark.rdd.compress* parameter. This is to reduce the overhead of transferring large numpy arrays of images encapsulated as RDDs. The *spark.io.compression.codec* parameter is the algorithm used for compressing the rdd, broadcast variable and the shuffled data. This parameter is required by the other compression parameters for them to function.

3.2.3 Experiments to investigate the influence of *spark.executor.memory* parameter on application performance

Here, the cache mechanism paradigm of Spark was investigated using the configuration parameter setting *spark.executor.memory*; particularly exploring the impact of increasing executor's memory on the execution speed of the applications. The experiments were performed using the July datasets for both application workloads on the small cluster environment. The execution time with *spark.executor.memory* of 10 GB memory was used as the baseline while the number of cores allocated to each executor kept at 16 (spark.executor.cores=16). In order to be able to experiment with different memory sizes for the *spark.executor.memory* parameter, the number of executors per node was fixed at 2 with 16 cores each. Therefore, the maximum memory capacity used by the executors on each node is 320 GB (total memory capacity was 377 GB). The detailed configuration settings used is as shown in Table 3.10.

To further investigate the impact of caching on application execution speed, the different storage levels implemented in Spark were studied for both applications on the large cluster (with 9 cluster nodes excluding *mario* and *luigi*) using only the July dataset for the *flowerCounter* application, while only the September dataset was employed for the *imageClustering* application.

3.2.4 Experiments to study the scalability of the Spark system with increasing number of computing nodes

Spark is a scale-out analytics system, that is, more nodes can be added to meet scalability demands. This is the focus of the experiments performed here. The scale-out inherent design was put to test by increasing the compute nodes from 1 to 11 (odd number increment) on the larger cluster for the *flowerCounter* application using the July dataset. The configurations used are the same as those used in §3.2.1 with the default

Table 3.10: Configuration Settings for Experiments in §3.2.3

Configuration Setting	Base Value						Parameter Type
spark.driver.memory	10 GB						Application
spark.broadcast.compress	true						Compression
spark.rdd.compress	true						Compression
spark.io.compression.codec	lz4						Compression
spark.shuffle.compress	true						Compression
spark.files.maxPartitionBytes	128 MB - *flowerCounter*			8 MB - *imageClustering*			Execution
spark.executor.memory	10 GB	20 GB	40 GB	80 GB	120 GB	160 GB	MemoryManagement
spark.executor.cores	16						Execution

partition size of 128 MB. The execution time taken on a single node was used as a basis for calculating the execution speedup. The completion time of the sequential version of the *flowerCounter* application executed on *onomi* was also compared with the corresponding parallel versions. The configuration on *onomi* (56 cores) is equivalent to 9 (6 cores per node were used) compute nodes on the larger cluster. Also, the execution time of a similar configuration (64 cores - 32 cores per node) on the smaller cluster was compared with the sequential runtime as well as the larger cluster run scenario.

3.2.5 Experiments to investigate the influence of *spark.speculation* configuration parameter on straggling tasks

Speculative execution in Spark is used to reduce the effect straggling nodes on job completion time. It is controlled by the *spark.speculation* parameter to reschedule slower tasks on the nodes to other worker nodes. In order to investigate the influence of this parameter on job completion time, experiments were performed with the parameter turned on (*spark.speculation=true*) using all the nodes in the larger cluster. There were two experimental scenarios: one with all the 11 nodes, but with speculation disabled and the other with the same number of nodes and speculation enabled. The idea was to investigate how the task scheduler would handle slower tasks on the straggling nodes to mitigate their effects on the job completion time. Speculative execution will not kill the slower tasks, but run the corresponding speculated tasks in parallel. If the speculated tasks finished before the slower tasks, the slower tasks would be killed and vice versa.

The experiments used all the datasets for the *flowerCounter* application with partition size of 128 MB. The *imageClustering* application, on the other hand, used only the July dataset on the large cluster with a partition size of 8 MB determined by the *spark.files.maxPartitionBytes* parameter. This is because the experiments failed with partition size greater than 8 MB for the July dataset due to *out-of-memory* exceptions. The complete configuration parameter settings are shown in Table 3.11.

Apart from the key *spark.speculation* parameter, there are other configuration parameters that are used in conjunction with the speculation parameter by default. These parameters include *spark.speculation.interval*,

spark.speculation.multiplier and *spark.speculation.quantile*. The *spark.speculation.interval* determines the frequency at which Spark will speculate tasks with the default value of 100 milliseconds. The *spark.speculation.multiplier* default value of 1.5 is the factor by which a task is slower than the median time of already completed tasks in a particular stage of execution. The *spark.speculation.quantile* used the default value of 0.75 to determine the segment of tasks that should be completed before speculation can take effect.

The other fixed configuration parameters, including *spark.driver.memory*, *spark.broadcast.compress*, *spark.rdd.compress*, *spark.io.compression.codec* and *spark.shuffle.compress*, apart from the *spark.speculation* parameter, shown in the table were chosen for similar performance reasons given in §3.2.2. In addition to these parameters, the *spark.executor.cores* and *spark.executor.memory* were also fixed at the set values determined by the systems' characteristics of the cluster nodes. Optimal values were set for these parameters with consideration for the operating system.

Table 3.11: Configuration Settings for Experiments in §3.2.5

Configuration Setting	Base Value		Parameter Type
spark.driver.memory	10 GB		Application
spark.broadcast.compress	true		Compression
spark.rdd.compress	true		Compression
spark.io.compression.codec	lz4		Compression
spark.shuffle.compress	true		Compression
spark.files.maxPartitionBytes	128 MB - *flowerCounter*	8 MB - *imageClustering*	Execution
spark.executor.memory	12 GB		Memory Management
spark.executor.cores	6		Execution
spark.speculation	false	true	Scheduling

Chapter 4

Performance Analysis and Evaluation

This chapter outlines the results from the experiments conducted in this study. It begins with an evaluation of the effect of data partition sizes on job completion time with detailed analysis of those stages whose execution time is substantially affected by partition size. The impact of executor JVMs on application performance with different configuration parameter settings on both the small and the large clusters is evaluated next. Execution times of the different configuration settings executed on both clusters and the sequential execution on *onomi* are reported. This chapter also shows the results for the influence of increasing Spark's execution memory on application performance. The results from the scale-out (horizontal node scaling) property experiment of Spark are presented next. The analysis and evaluation of the results obtained from the speculative execution experiments concludes this chapter.

4.1 Effect of Partition Size

The results of the experiments described in §3.2.1 are discussed here. In order to determine the right partition size for the applications, experiments with different data partition sizes were conducted for the *flowerCounter* application. The influence of partition size on the total processing time of the *flowerCounter* application is plotted in Figure 4.1. Experiments were conducted starting with the default partition size of 128 MB and up till 2 MB data size. Partition size of 64 MB reduces processing time from the default data size. Processing time remained almost the same for further reduction in the partition size until the smallest partition size of 2 MB considered. In order to aid adequate understanding of the cause of the patterns observed, for example the processing time reduction from 128 MB to 64 MB partition size, further studies were carried out as discussed in the succeeding discussions. These results indicate that reduction of the partition size from the default partition size of 128 MB to 64 MB reduces processing time and further reductions in the partition size do not provide speed advantage for the processing. This limitation of the Spark engine to scale with smaller partition sizes is due to the bottleneck caused by large number of tasks and thus results in the inefficient utilization of the cluster CPU/IO resources [51]. This finding is further corroborated by the analyses that follow here.

As stated above, detailed analyses were conducted for the 64 MB and 128 MB partition sizes to better understand the underlying cause of the patterns observed. Table 4.1 shows the task summary results

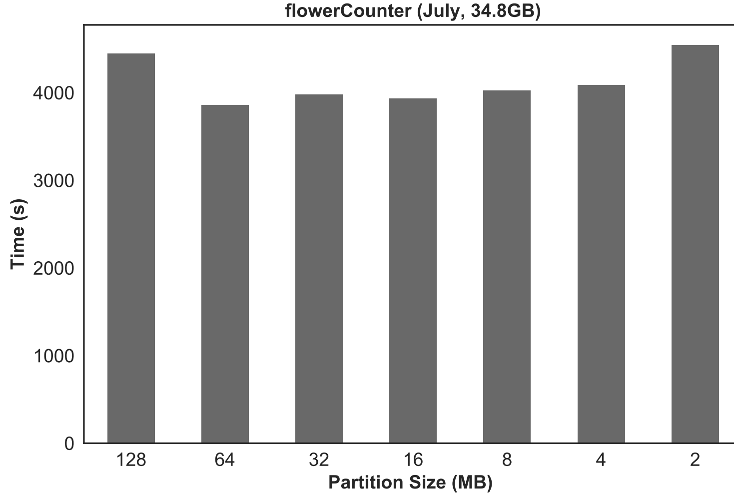


Figure 4.1: Partition Size vs Runtime

from those experiments. The table shows the execution time summary for concurrently running tasks for each stage across all the cluster nodes. As highlighted in the table, the stages whose execution times (with respect to the total elapsed) are substantially affected by the partition size are the *computeHistogram* and *computeHistogramShifts* respectively. Consequently, analysis focuses only on the *computeHistogram* and *computeHistogramShifts* stages because the completion time of these stages substantially impacted the total job completion time. The execution times of the other two stages (*computeFlowerPixelPercentage* and *computeFlowerCount*) are reduced as well but the difference in the total execution times is not as substantial as the other two.

The *computeFlowerCount* stage accounted for about 76% and 68% of the total completion in both run scenarios respectively. Despite this substantial contribution and the large variance in the median and maximum time, the difference in the execution times for the *computeFlowerCount* stage is very similar in the run scenarios. This is because the slow tasks in the 128MB partition size scenario completed almost at the same time as the large number of tasks in the 64 MB partition size scenario.

The results are further exemplified in Figure 4.2. These results indicate that the partition size of 64 MB reduced the median execution time by 66%, 58%, 51% & 50% per task in each of the stages respectively and reduced the total job completion time by about 13% compared to the 128 MB partition size scenario. These results represented also reveal that the variation (the difference between the median and the maximum time) in the tasks' completion time is higher for the *computeHistogram* and *computeHistogramShifts* stages of the 128 MB partition size scenario explained by the large number of outliers in those two stages. This variation (and large number of outliers) is due to straggler tasks that have prolonged execution time in comparison to other concurrent tasks.

In order to understand what might be responsible for the increased completion time (caused by the variation in the tasks' completion time) of the 128 MB partition size scenario, the tasks' completion progress

Table 4.1: *flowerCounter*: Influence of *spark.files.maxPartitionBytes* (July, 34.8 GB Dataset)

64 MB Partition Size				
Stages (Function Name)	Minimum Time(s)	Median Time(s)	Maximum Time(s)	Total Time(s)
collect (computeHistogram)	5	27	55	302
collect (computeHistogramShifts)	5	25	75	280
collect (computeFlowerPixelPercentage)	9	29	124	333
saveAsTextFile (computeFlowerCount)	64	275	440	2947
Total Elapsed Time				3862
128 MB Partition Size				
Stages (Function Name)	Minimum Time(s)	Median Time(s)	Maximum Time(s)	Total Time(s)
collect computeHistogram	14	80	414	624
collect computeHistogramShifts	7	59	220	415
collect computeFlowerPixelPercentage	9	59	144	368
saveAsTextFile computeFlowerCount	91	554	862	3042
Total Elapsed Time				4449

views are shown in Figures 4.3a and 4.3b for the *computeHistogram* stage and similarly in Figures 4.4a and 4.4b for the *computeHistogramShifts* stage. These views show that slow tasks are the cause of the slow completion time for the 128 MB partition size scenario; the time for the 128 MB partition size is almost twice of the 64 MB partition size.

In the *computeHistogram* stage, for example, the task progression is almost linear with the 64 MB partition size scenario unlike the 128 MB partition size scenario. At the completion time of about 300 seconds for the 64 MB (with 556 total tasks) partition size scenario, there are still about 39 tasks that are yet to be completed in the 128 MB (with 279 tasks in total) partition size scenario and thus prolonged the completion time. This is due to the large task sizes of the 128 MB partition size scenario in comparison to the 64 MB partition size scenario.

Also, Table 4.2 shows the execution time summary for each distinct task (these times are the summation of tasks times with each task time added distinctly and not times for concurrent running tasks) added together. These results again show that the average task execution time for the 128 MB partition size scenario is about twice as that of the 64 MB partition size scenario (over 3 minutes for 128 MB as against about 1.5 minutes for 64 MB) across all cluster nodes. This also corroborates findings in the preceding section that the high

completion time of the 128 MB partition size is due to large variation in the tasks' completion time dictated by large task sizes.

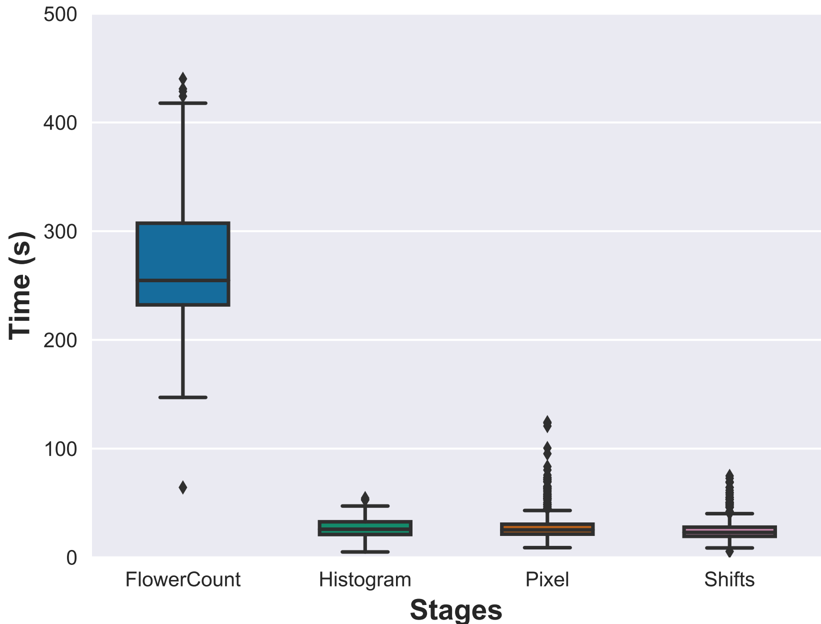
Furthermore, it is important to state that the task progress views in Figures 4.3 and 4.4 reflect the tasks time due to computation only. This is because majority of the tasks' elapsed times were spent in compute (other task time metrics are negligible and as such are not represented). The results shown in Figure 4.3 for the *computeHistogram* stage show that tasks scheduling follows an almost linear fashion with only a few prolonged tasks in the 64 MB partition size scenario. The scheduling of tasks suffers no substantial delays because concurrent tasks completed almost at the same time. The maximum number of concurrent tasks, dictated by the number of cores allocated for the tasks, is 54 (6 cores on each of the 9 nodes in the cluster). The total number of tasks is 556. This means that each node would execute at least 60 tasks in this stage (assuming about 10 tasks per core).

On the other hand, for the 128 MB partition size scenario, the scheduling of tasks is largely affected by straggler tasks throughout the entire tasks' execution flow. The tasks are substantially delayed especially in the latter part of the execution. The total number of tasks in the 128 MB partition size scenario is 279, yielding at least 30 tasks on each node for about 5 tasks per core. It thus appears that the prolonged stage execution time of the 128 MB partition size scenario is due to same large task sizes that were slow to complete and reduced number of tasks in comparison to the 64 MB partition size scenario. The same explanation holds for the *computeHistogramShifts* results depicted in Figure 4.4.

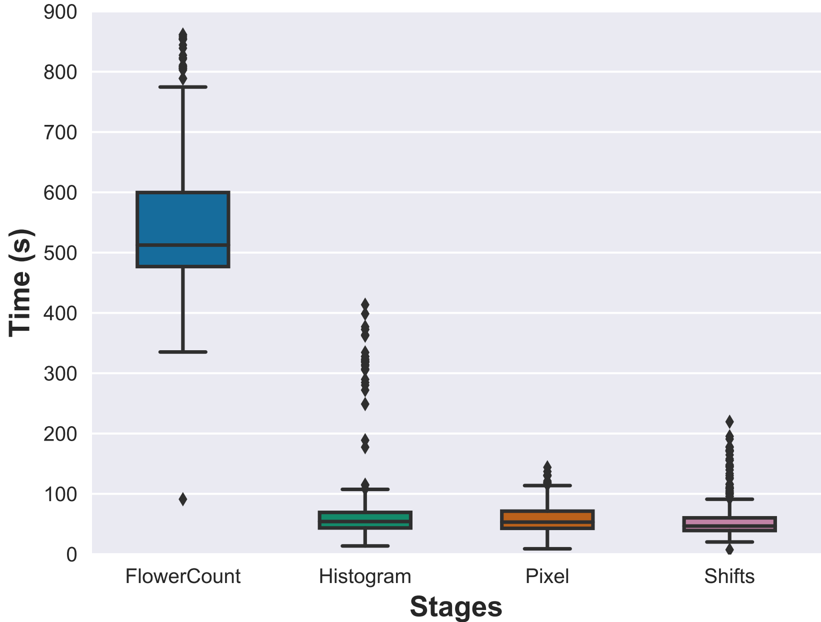
To further investigate the influence of the *spark.files.maxPartitionBytes* parameter on cluster resources, metrics were collected with respect to CPU utilization, memory usage, network and disk throughput as well as HDFS I/O as shown in Figures 4.5, 4.7, 4.6, 4.8, 4.9 and 4.10 respectively. For all the metrics reported, the average values sampled at every 30 seconds are plotted in these figures. In general, these metrics show that the rate at which resource usage changes is more frequent with the 128 MB partition size scenario in comparison to the 64 MB partition size scenario. A detailed discussion of these figures follows in the succeeding paragraphs.

For the CPU usage illustrated in Figure 4.5, the downward spikes represent the interchange from one stage to the next in the application processing pipeline. It can be seen from the result represented that the interchange between the stages happens almost immediately for the 64 MB partition size scenario while in the 128 MB partition size scenario the interchange is prolonged and stretched. The delay in the first interchange (that is, the end of the *computeHistogram* stage in the 128 MB partition size scenario) is more pronounced than all the other stages. This is because some tasks took longer time to complete and thus resulted in the pattern observed. Similarly, the second interchange which represents the completion of the *computeHistogramShifts* stage finishes longer in the 128 MB partition size scenario in comparison to the 64 MB partition size scenario. In general, CPU utilization is similar in both partition size scenarios but with more pronounced volatility in the 128 MB partition size scenario especially in the last *computeFlowerCount* stage.

flowerCounter (July, 34.8 GB)



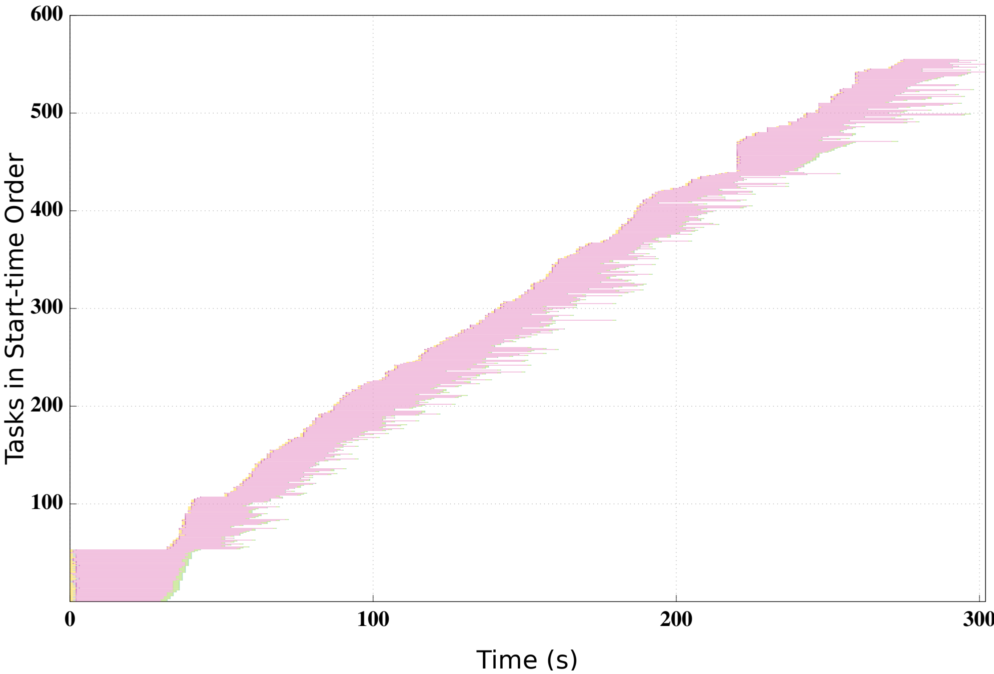
(a) Stage Summary for *flowerCounter* 64 MB



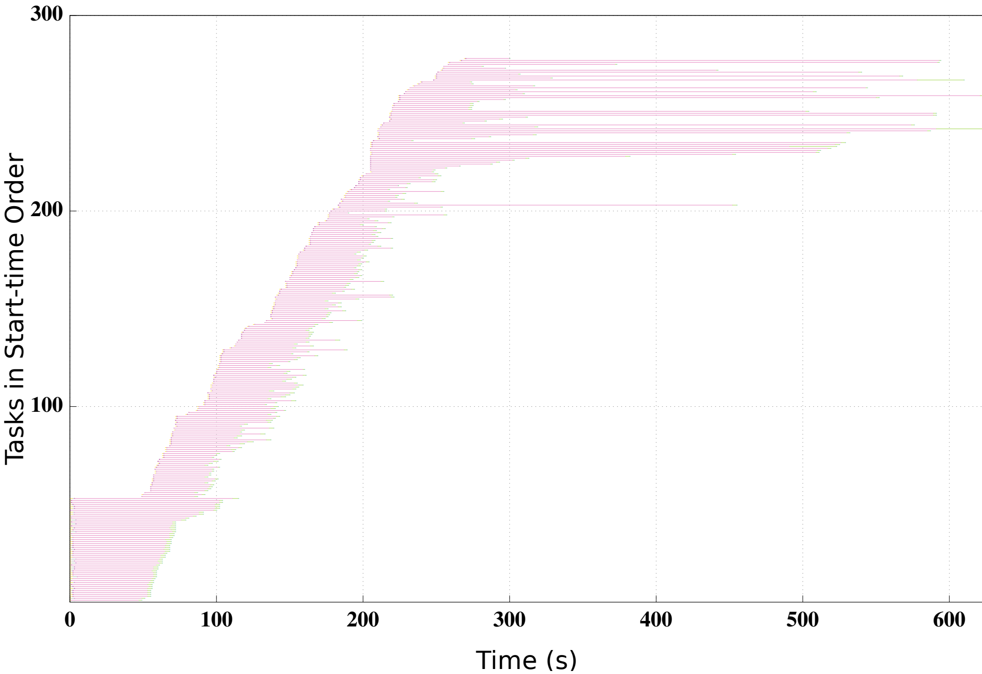
(b) Stage Summary for *flowerCounter* 128 MB

Figure 4.2: Stage Time Summary

flowerCounter (July, 34.8 GB)



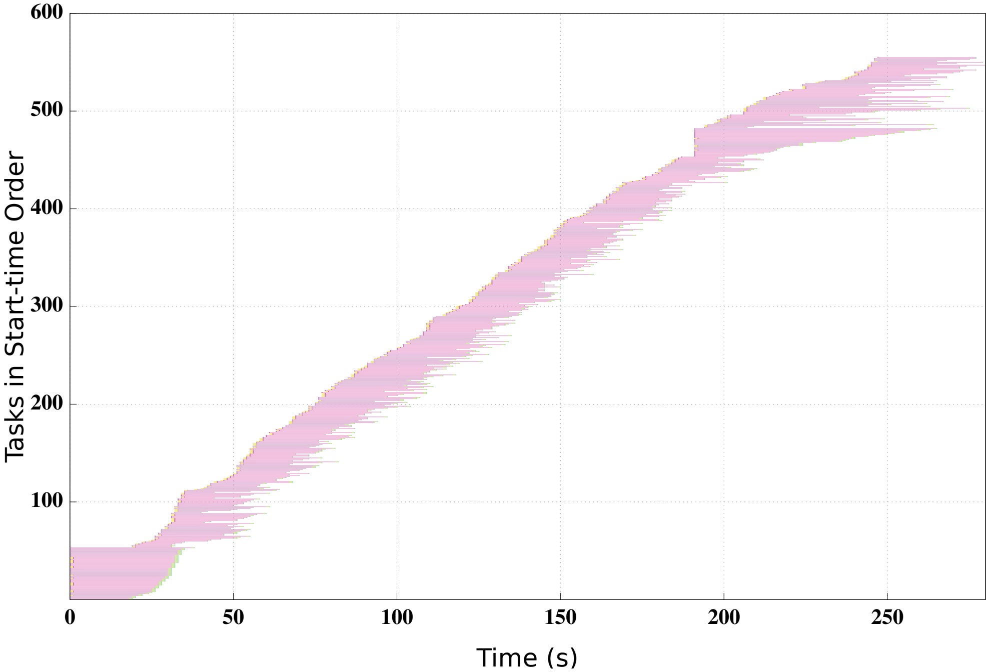
(a) computeHistogram (64 MB)



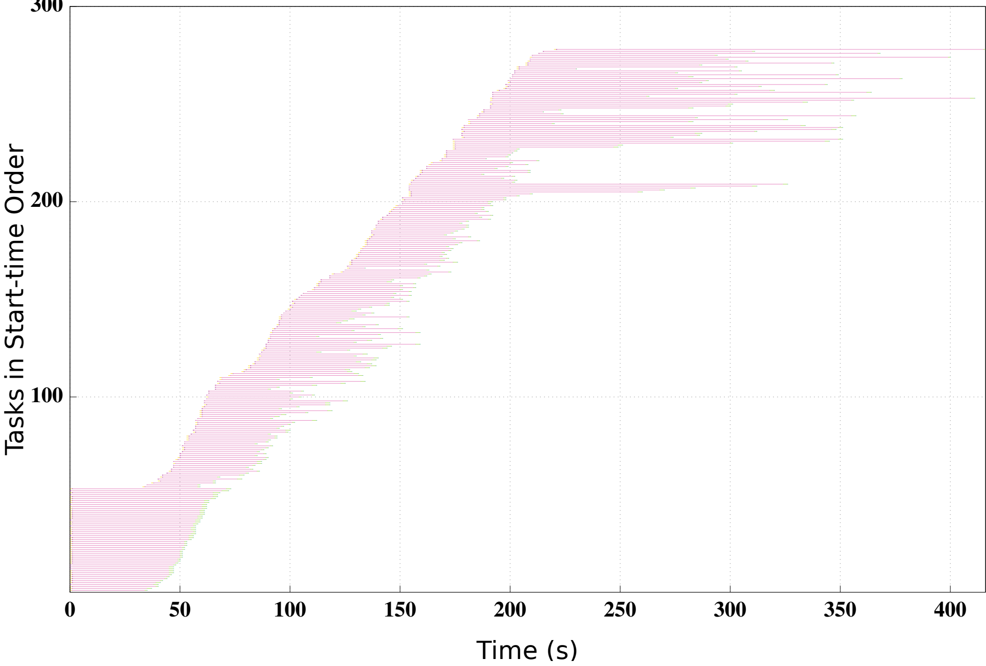
(b) computeHistogram (128 MB)

Figure 4.3: Task Progress Summary

flowerCounter (July, 34.8 GB)



(a) computeHistogramShifts (64 MB)

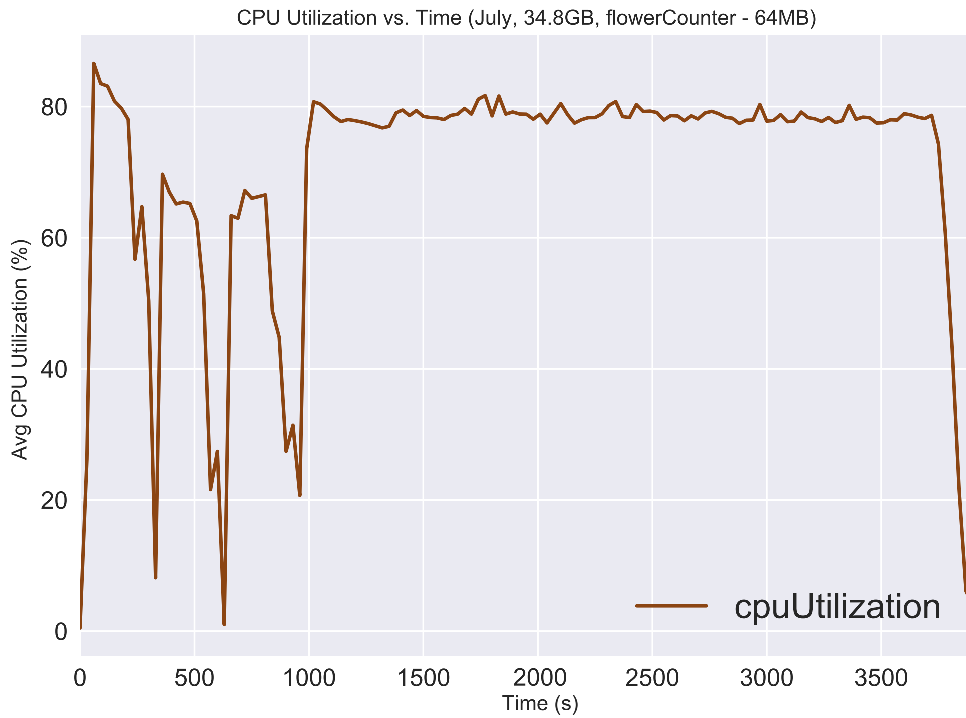


(b) computeHistogramShifts (128 MB)

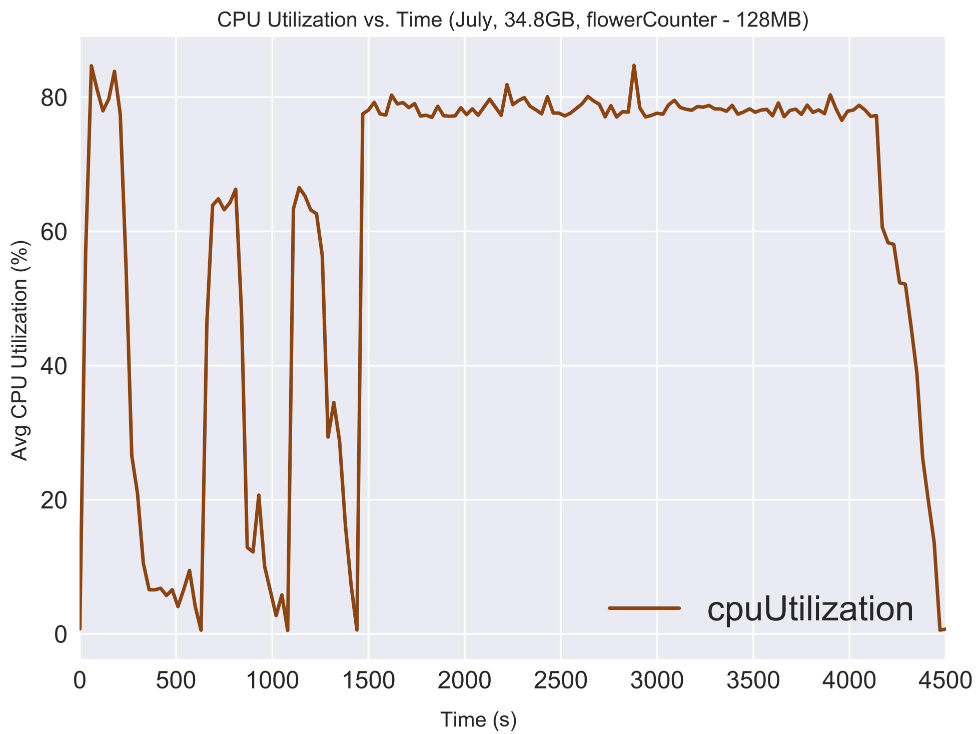
Figure 4.4: Task Progress Summary

Table 4.2: *flowerCounter*: Influence of *spark.files.maxPartitionBytes* (July, 34.8 GB Dataset)

64 MB Partition Size					
Host	Minimum Duration(s)	Average Duration(s)	Maximum Duration(s)	Total Duration(s)	No of Tasks
discus-p2irc-worker1	13	89	400	22000	246
discus-p2irc-worker2	5	89	416	21900	246
discus-p2irc-worker3	13	94	409	22000	234
discus-p2irc-worker4	14	91	424	22000	242
discus-p2irc-worker5	5	85	418	22000	259
discus-p2irc-worker6	14	89	428	21900	245
discus-p2irc-worker7	16	88	440	22100	253
discus-p2irc-worker8	14	89	415	21900	247
discus-p2irc-worker9	2	87	402	22100	254
128 MB Partition Size					
Host	Minimum Duration(s)	Average Duration(s)	Maximum Duration(s)	Total Duration(s)	No of Tasks
discus-p2irc-worker1	22	181	821	23900	132
discus-p2irc-worker2	14	190	823	22300	117
discus-p2irc-worker3	9	176	809	23200	132
discus-p2irc-worker4	28	185	854	23000	124
discus-p2irc-worker5	7	202	858	23500	116
discus-p2irc-worker6	28	188	862	24700	131
discus-p2irc-worker7	29	191	860	23100	121
discus-p2irc-worker8	7	196	844	22700	116
discus-p2irc-worker9	30	182	806	23500	129



(a) Average CPU Utilization (64 MB)



(b) Average CPU Utilization (128 MB)

Figure 4.5: CPU Utilization

To further study CPU utilization, CPU heat maps are shown in Figure 4.6. The stage interchange regions are those with the least CPU usage shown in the figure. The regions are wider in the 128 MB partition size scenario than in the 64 MB partition size scenario. At these regions, CPU utilization drops well below 20% for a prolonged time period in the 128 MB partition size scenario in comparison to the 64MB run. This again shows that the prolonged application processing time of the 128 MB partition size scenario is largely contributed by the slower tasks in the *computeHistogram* and *computeHistogramShifts* stages of the application pipeline. These slower tasks delay the task scheduler in scheduling tasks in the *computeHistogramShifts* stage as tasks in this stage cannot be initiated without the completion of the tasks in the previous *computeHistogram* stage.

RAM usage for both run scenarios of 64 MB and 128 MB partition sizes is shown in Figure 4.7. RAM usage is higher in the first *computeHistogram* stage for the 128 MB partition size scenario in comparison to the 64 MB partition size scenario but usage is higher and less variable in the last three stages for the 64 MB partition size scenario. There is high volatility in the last *computeFlowerCount* stage of the 128 MB partition size scenario in that usage variation is more frequent than in the 64 MB partition size scenario.

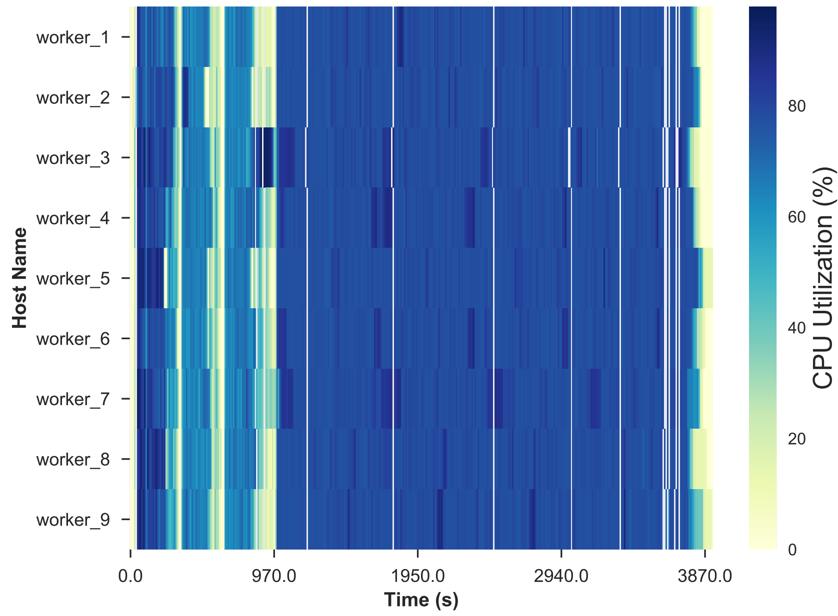
In general, the high CPU and RAM usage by the application in both run scenarios across all the stages of execution in the pipeline is due to large numpy array transformations. These stages involve expensive numpy matrix computation and deep array dictionaries copying for operations such as histogram computation, image correlation for histogram shifts calculation and the clip function for obtaining flower highlight. These operations require expensive use of internal memory and therefore are memory intensive. The high CPU volatility especially in the 128 MB partition size scenario requires further study.

Next, the disk and network throughput due to the operating system's filesystem for the two run scenarios were measured and the result is represented in Figures 4.8 and 4.9. With respect to disk throughput, both run scenarios are low on write throughput except for a few spikes at the interchange between the stages. These spikes are due to the output from each of the stages.

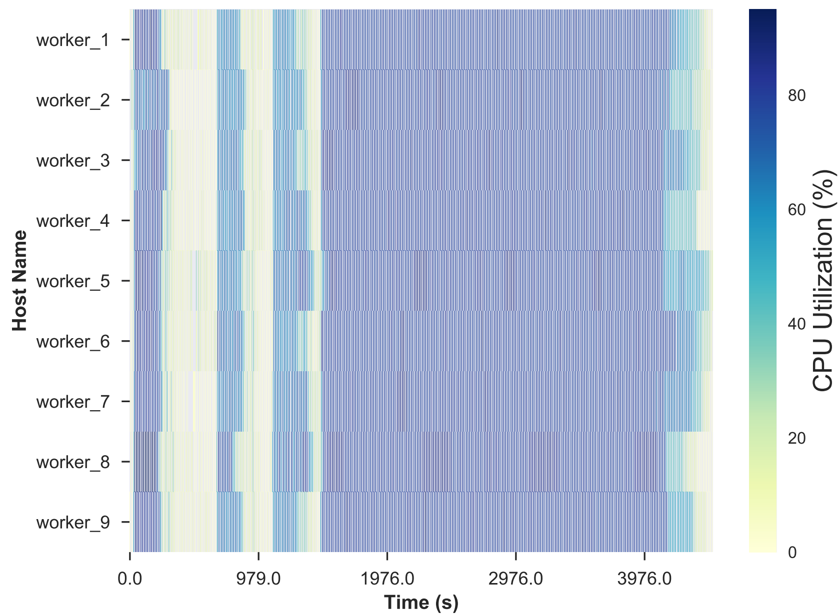
The read and network throughput on the other hand depicts a different usage pattern with higher read throughput in the first three stages than in last stage for the 64 MB partition size scenario. This explains why the execution time is faster in the first three stages of the pipeline for the 64 MB partition size scenario. The first two stages, especially the *computeHistogram* stage, shows elongated near-zero read throughput due to fewer tasks with delayed completion time. Disk throughput is largely dominated by the HDFS I/O activity (particularly the read operations) and are thus discussed in later paragraphs. In both scenarios, network thoroughput is low and negligible.

Also, the network throughput for the 128 MB partition size scenario reflects similar pattern as in the disk bytes read throughput with bytes transferred and received over the network following nearly identical trend in the first three stages. The network usage in these stages, especially in the *computeHistogram* stage, is also synonymous to the disk bytes read throughput with elongated near-zero network usage pattern due to slow tasks.

flowerCounter (July, 34.8 GB)

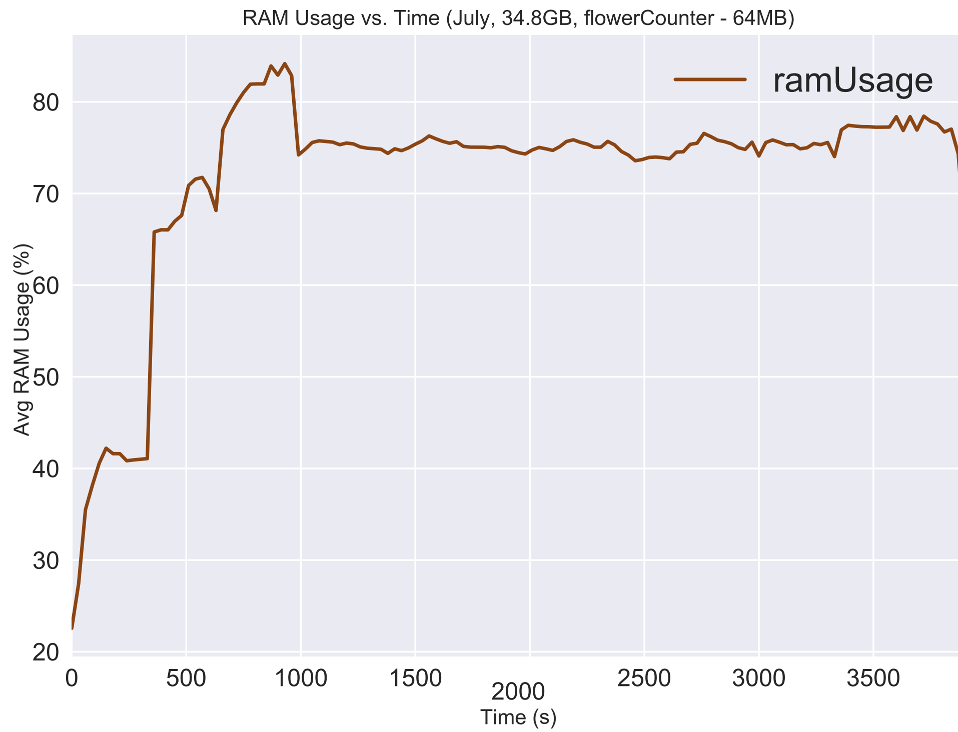


(a) CPU HeatMap (64 MB)

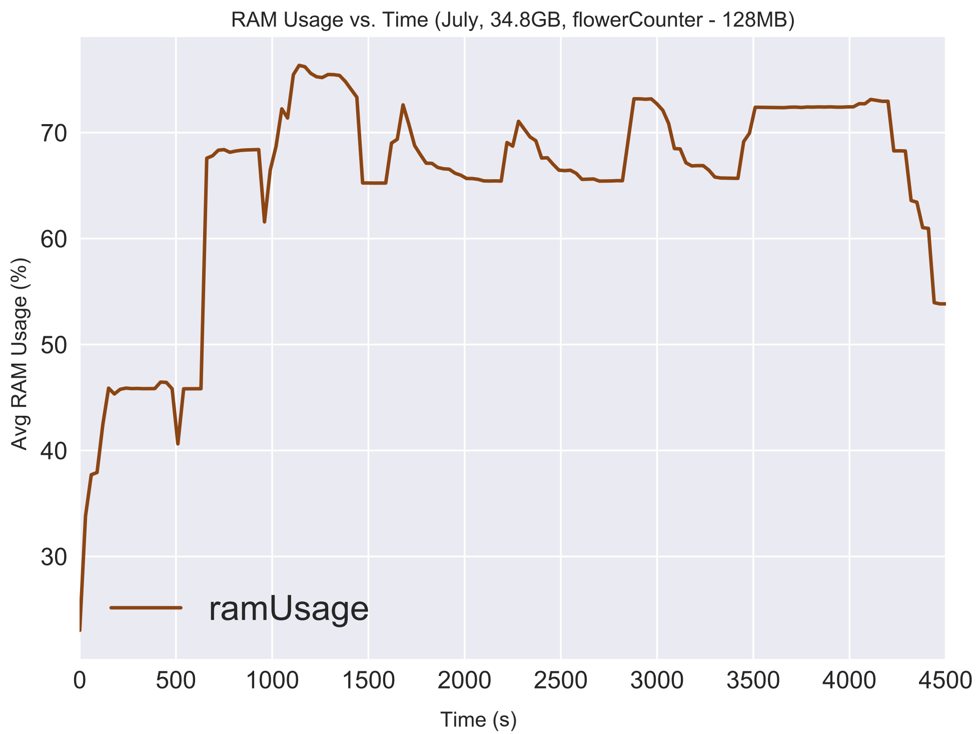


(b) CPU HeatMap (128 MB)

Figure 4.6: CPU HeatMap



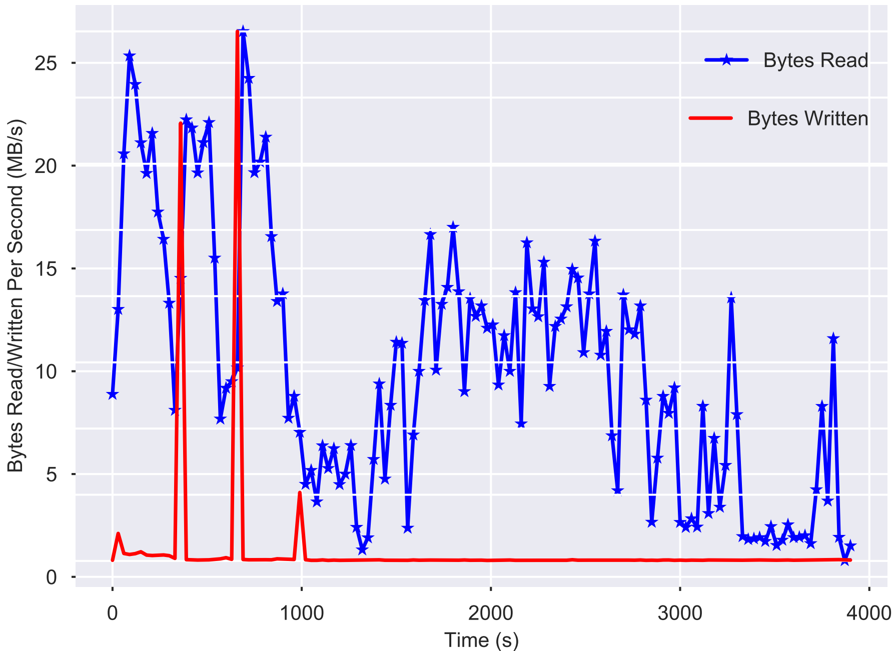
(a) RAM Usage (64 MB)



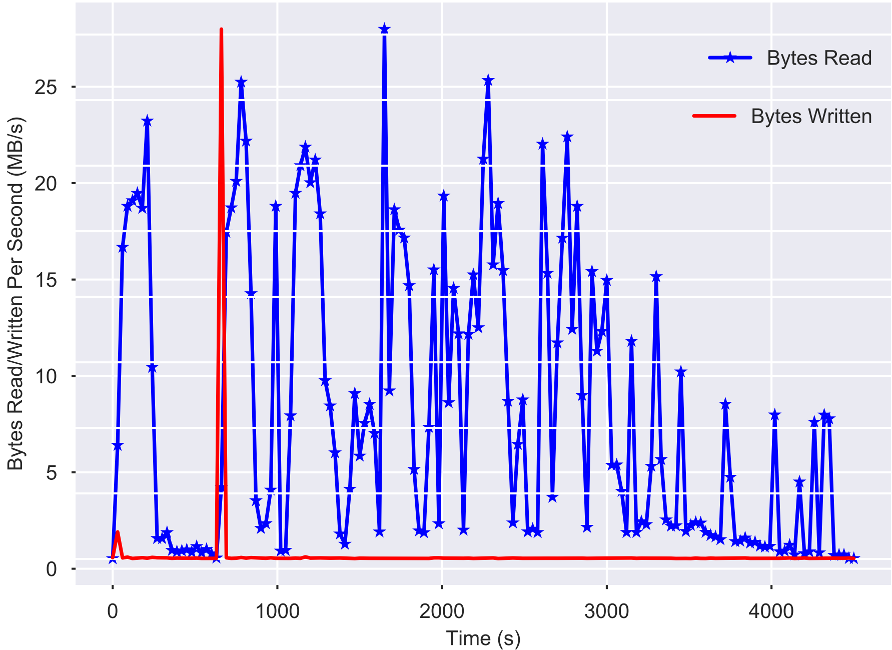
(b) RAM Usage (128 MB)

Figure 4.7: RAM Usage

flowerCounter (July, 34.8 GB)



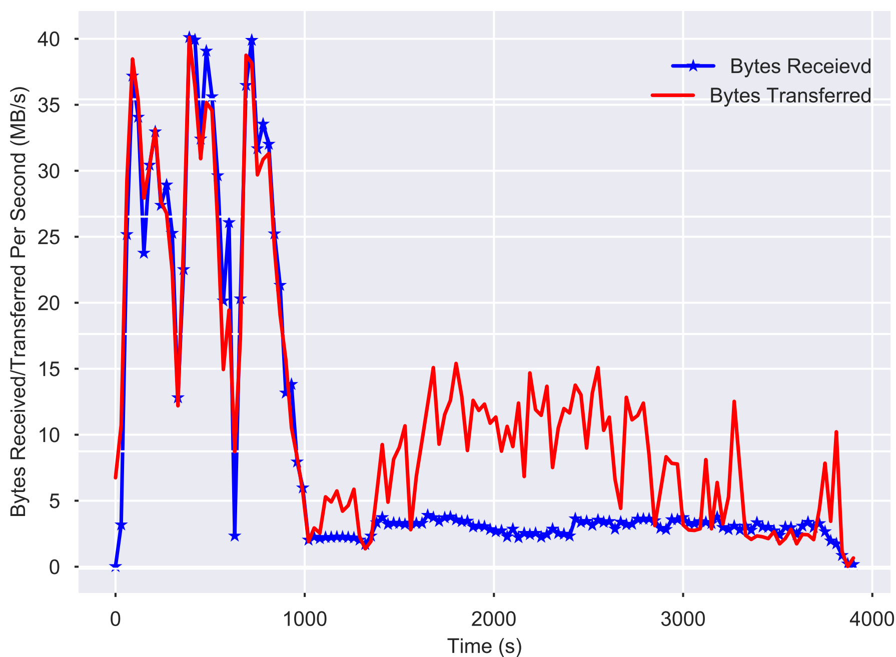
(a) Disk Throughput (64 MB)



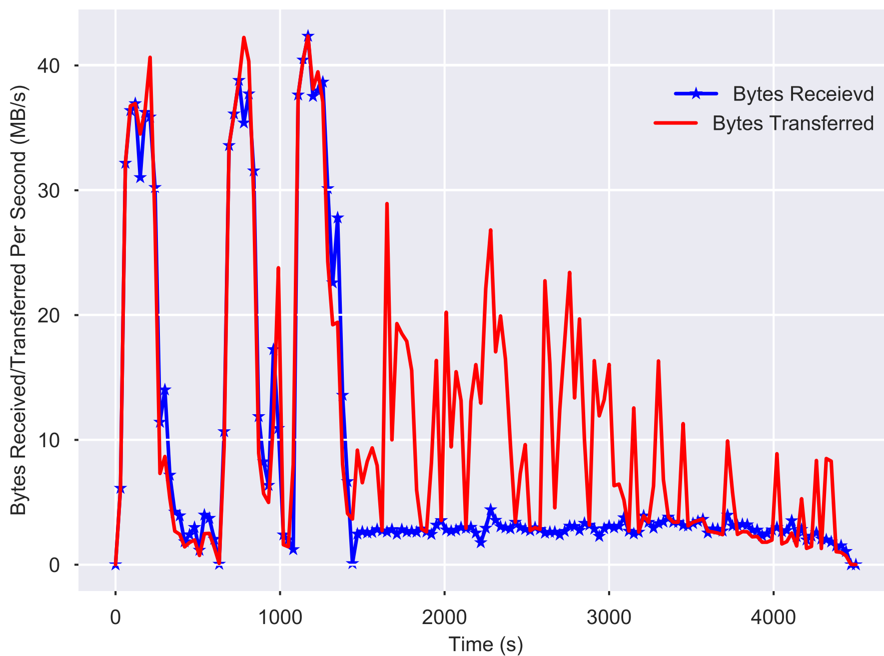
(b) Disk Throughput (128 MB)

Figure 4.8: Disk Throughput

flowerCounter (July, 34.8 GB)



(a) Network Throughput (64 MB)



(b) Network Throughput (128 MB)

Figure 4.9: Network Throughput

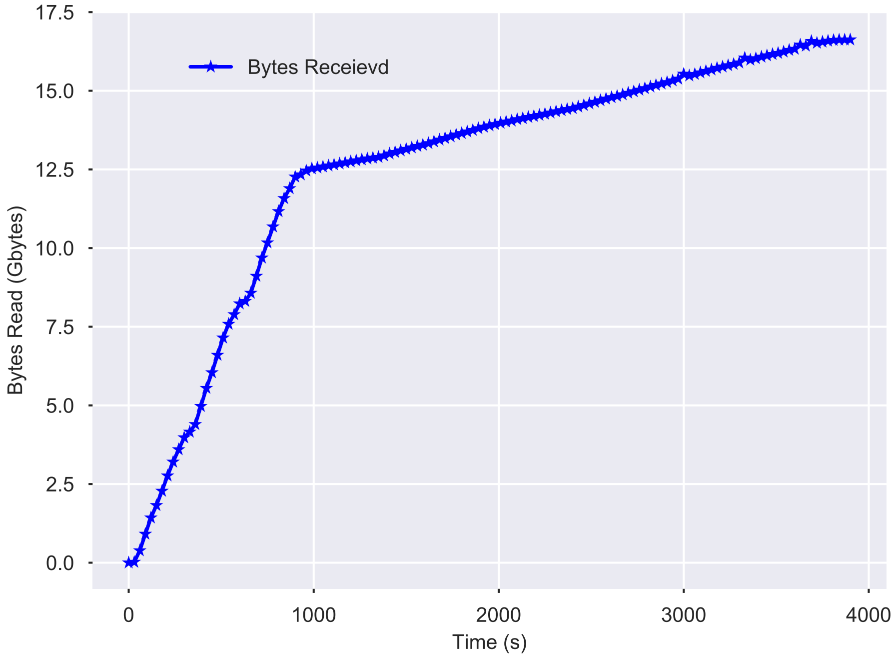
The disk I/O cost of processing the application with different partition sizes includes the I/O cost of reading and writing data to HDFS and as such the HDFS I/O metrics were measured as represented in Figure 4.10. The HDFS write operation, however, is very negligible. The HDFS disk read cost rises steadily in the first three stages of the application in both experimental run scenarios except that in the 64 MB partition size scenario the rate is fast with almost negligible delays between the stages. There is a steady read rate until about 900 seconds in the 64 MB partition size scenario. The read rate then became smaller until the end.

In the 128 MB experimental run however, the increase in the HDFS disk read operation in the first three stages (with a stair-case pattern) is affected by prolonged delays towards the end of the stages especially in the *computeHistogram* and *computeHistogramShifts* stages. The number of bytes written by HDFS is inconsistent in the 128 MB partition size scenario due to smaller number of tasks in comparison to the 64 MB partition size scenario. The read rate is similar as in the 64 MB partition size scenario except that at some points in the early stages of the run, the read rate became very small and bursty.

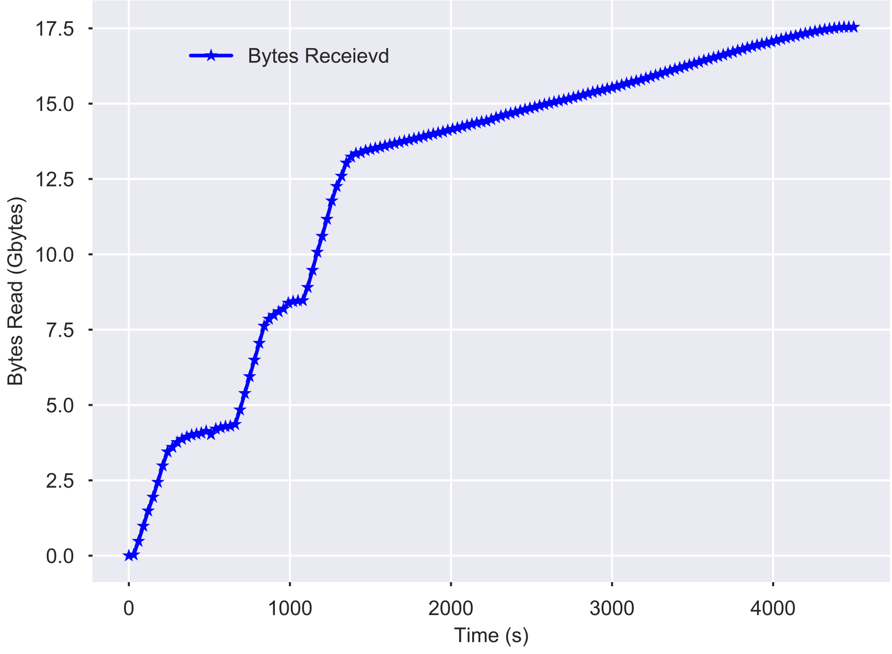
In summary, these results indicate that reduction in partition size reduces the application processing time till 64 MB but subsequent reduction do not yield faster processing time due to the overhead caused by large amount tasks [51]. This finding agrees with the work done by Veiga *et al.* [54] that across a number of benchmarks investigated for Hadoop and Spark, 64 MB partition size is most suitable for Spark workloads. Running the application with the default partition size of 128 MB increases application processing time due slow tasks especially in the *computeHistogram* and *computeHistogramShifts* stages.

Default partition size of 128 MB also causes more frequent variation in the use of cluster resources especially in the last *computeFlowerCount* stage in comparison to using 64 MB partition size. The repeated successive variation in the use of cluster resources might be due to contention from concurrent tasks and because resource usage is outside the control of the Spark's engine but rather depends on the operating system [42]. Generally, I/O is quite small in comparison to CPU usage and the disk reads rate are staggered because cores are waiting for stragglers which confirms earlier analysis.

flowerCounter (July, 34.8 GB)



(a) HDFS IO (64 MB)



(b) HDFS IO (128 MB)

Figure 4.10: HDFS IO

4.2 JVM Executor Scaling

Here, the impact of increasing the number of JVM executors vis-a-vis executor cores on application performance is investigated. In this scenario (in the preliminary experiments), for the *flowerCounter* application, two executor JVMs were used with 64 MB and 128 MB data partitions respectively using the August dataset on the larger cluster. Each JVM executor has 3 cores and 6 GB of memory on all the 9 nodes, making 18 executors in total. The number of executors was then later increased to 27; each node has 3 executors with 2 cores and 4 GB of memory each accordingly. The preliminary experiment with 27 JVM executors was done for the 64 MB data partition size only. The summarized execution time results for the 18 executors and 27 executors are shown in Table 4.3 for each experimental run scenario.

The 27 JVM executors scenario with 64 MB data size performed better in comparison with the 18 JVM executors scenario. The variance in execution times is larger in the *computeHistogramShifts* and *computeFlowerPixelPercentage* stages than in the other stages. The task completion progress views for the 18 and 27 JVM executors scenarios are shown in Figures 4.11a, 4.11b, 4.11c and 4.11d to further make obvious the tasks' execution dynamics for the two notable stages (*computeHistogramShifts* and *computeFlowerPixelPercentage*).

For both stages, the results represented show that tasks took a longer time to complete in the 18 JVM executors scenario than in the corresponding 27 JVM executors scenario, especially in the *computeHistogramShifts* stage. In the *computeHistogramShifts* stage, for example, the tasks run time in the 27 JVM executors scenario is about twice that of the 18 JVM executors scenario.

Furthermore, the task progress views shown in Figure 4.11 for the *computeHistogramShifts* and the *computeFlowerPixelPercentage* stages reflect the tasks' completion rate over the execution period of the application processing. The total number of tasks in both run scenarios for the two stages is 308.

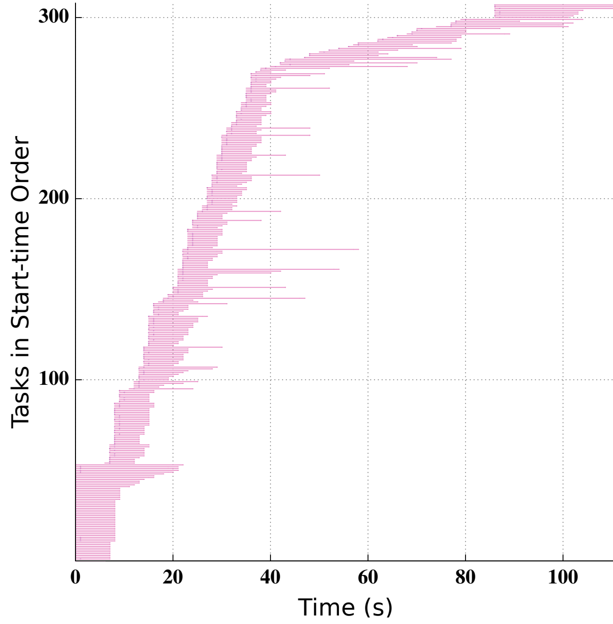
For the *computeHistogramShifts* stage, the tasks' completion rates for the 18 JVM executors scenario is greatly dominated by straggler tasks throughout the entire tasks' execution period in comparison to the 27 JVM executors scenario. Each phase in the task execution progress contains tasks that delayed the starting of tasks by the task scheduler in the next phase of the execution. The 27 JVM executors scenario still has a few tasks that executed slowly but their effects are not substantial in comparison to the 18 JVM executors scenario. This finding is corroborated by the time summary statistics shown in Table 4.3 respectively. The minimum and median times are the same in both experimental scenarios, but the maximum time in the 18 JVM executors' scenario is over twice the 27 JVM executors' scenario maximum time. The resultant effect of these time statistics in the 18 JVM executors scenario is increased stage execution time that is over twice the stage execution time of the 27 JVM executors scenario.

Also, for the *computeFlowerPixelPercentage* stage executed with both 18 and 27 JVM executors, the tasks' completion rate is also shown in Figure 4.11. These views again show that the tasks' completion progress for the 18 JVM executors scenario has more tasks that were prolonged in comparison to the 27 JVM

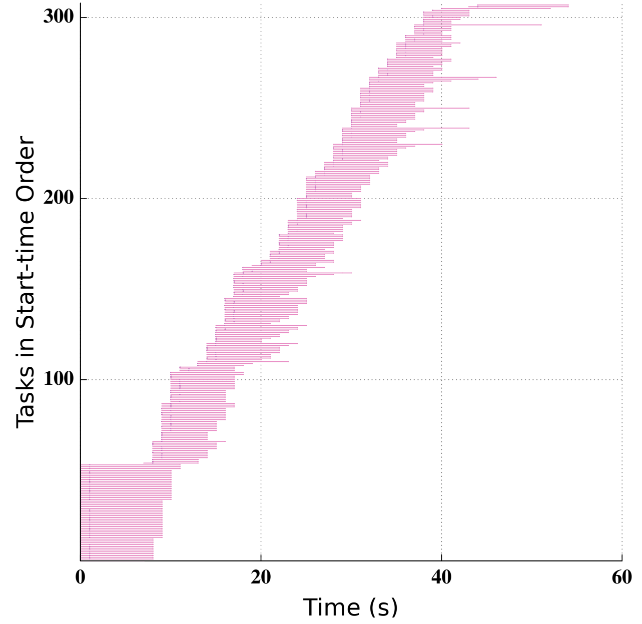
Table 4.3: *flowerCounter*: Influence of *spark.executor.cores* (August, 19.3 GB Dataset)

18 JVM Executors (64 MB)				
Stages (Function Name)	Minimum Time(s)	Median Time(s)	Maximum Time(s)	Total Time(s)
collect (computeHistogram)	7	18	35	132
collect (computeHistogramShift)	3	7	36	114
collect (computeFlowerPixelPercentage)	2	13	72	138
saveAsTextFile (computeFlowerCount)	96	228	408	1440
Total Elapsed Time				1824
18 JVM Executors (128 MB)				
Stages (Function Name)	Minimum Time(s)	Median Time(s)	Maximum Time(s)	Total Time(s)
collect (computeHistogram)	23	38	72	138
collect (computeHistogramShift)	6	27	210	210
collect (computeFlowerPixelPercentage)	13	35	168	186
svaeAsTextFile (computeFlowerCount)	56	456	720	1500
Total Elapsed Time				2034
27 JVM Executors (64 MB)				
Stages (Function Name)	Minimum Time(s)	Median Time(s)	Maximum Time(s)	Total Time(s)
collect (computeHistogram)	8	17	31	108
collect (computeHistogramShift)	3	7	14	54
collect (computeFlowerPixelPercentage)	3	10	21	90
saveAsTextFile (computeFlowerCount)	108	222	420	1440
Total Elapsed Time				1692

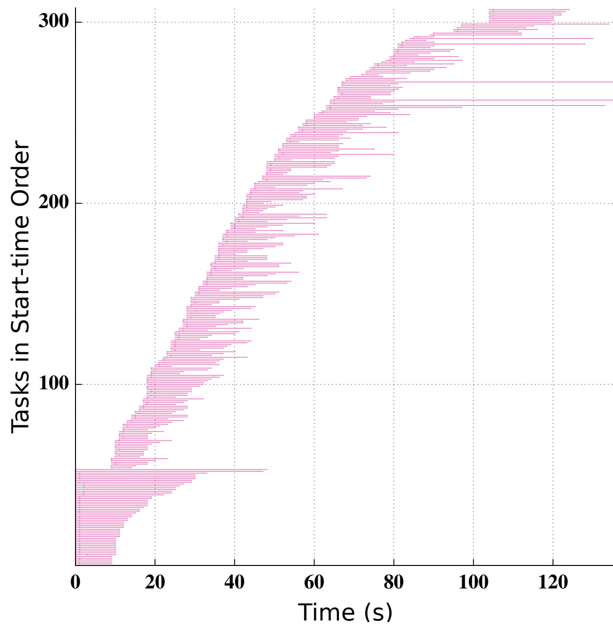
flowerCounter (August, 19.3 GB)



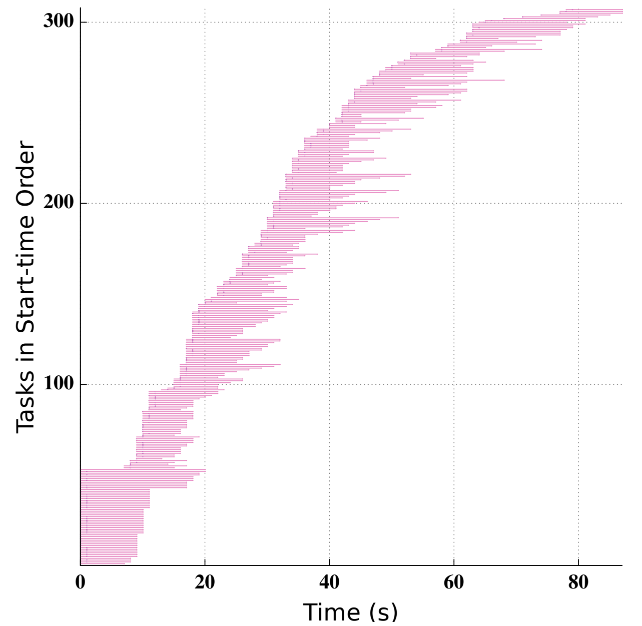
(a) 18 JVM Executors for Shifts (64 MB)



(b) 27 JVM Executors for Shifts (64 MB)



(c) 18 JVM Executors for Pixel (64 MB)



(d) 27 JVM Executors for Pixel (64 MB)

Figure 4.11: Tasks Completion Progress Summary

executors scenario. The difference in the tasks' completion rates is not very substantial when compared with the *computeHistogramShifts* stage. For both scenarios, the minimum and median times are similar but with large maximum time variance which resulted in execution time that is about 1.5 times the 27 JVM executors scenario for the 18 JVM executors scenario.

In the preliminary experiments, for the *imageClustering* application, the effect of the JVM executor scaling was studied for five iteration steps. Two different experimental scenarios were considered on the larger cluster with 128 MB partition size: the first used 9 executors, one executor per node with 6 cores and 12 GB of memory and the other used 18 executors - two executors per node with 3 cores each and 6 GB of memory. The September dataset was used for both run scenarios. There were 46 tasks in each of the stages in the execution pipeline.

For the five iteration steps considered, there are eight key stages involved in the application processing whose execution times are substantial for the purpose of analysis. The summarized execution time results for the 9 and 18 JVM executors scenarios are as shown in Tables 4.4 and 4.5. The results show that the median execution time for the *collectAsMap* stages is reduced by about 30% for the 18 JVM executors scenario than for the 9 JVM executors scenario. This is because the number of executors is halved while the number tasks remains the same and thus, the execution time is slower in the 9 JVM executors scenario. Also, the *collectAsMap* stages are more stable in the 18 JVM executors scenario. The time execution summary also shows that the 9 JVM executors scenario is dominated by slower tasks as reflected in the large variation of the time summary statistics especially in the *collectAsMap* stages and the second *takeSample* stage.

To further corroborate these results, the tasks progress charts for both scenarios are as shown in Figures 4.12 and 4.13 taking into consideration the first three stages. These charts show that data is well distributed across the nodes in the 18 JVM executors scenario which resulted in few slow tasks, better utilization and reduced execution times as against the 9 JVM executors scenario.

For the first *takeSample* stage, the task completion distribution is depicted in Figure 4.12 for both scenarios considered. It is quite obvious that the prolonged tasks' completion time in the 9 JVM executors scenario is due to a few straggling tasks as opposed to the 18 JVM executors scenario. The results represented also show that the run time of the straggler tasks is dominated by the time spent in garbage collection (GC) towards the end of the processing in both instances. The garbage collection time is higher in the 9 JVM executors scenario, but the difference is small. Therefore, the garbage collection time is not represented in the figure as evaluation focused only on the execution times for each of the tasks. The task times correspond to the length of the line for each task respectively.

Also, the tasks' completion distribution for the second *takeSample* stage is shown in Figure 4.12 for both scenarios. For both executors scenarios, the tasks' completion rate is better in the 18 JVM executors scenario (though with a few slow tasks) than in the 9 JVM executors scenario. In the 9 JVM executors scenario, some tasks completed very fast while a few others took longer time to complete as reflected also in the execution time metrics shown in Table 4.4.

In the third stage (*collectAsMap*) shown in Figure 4.13, tasks' completion progress is faster in the 18 JVM executors scenario than in the 9 JVM executors scenario though with a few stragglers in both scenarios. It is also apparent from the result represented, as with other results discussed in the previous sections, that tasks completion is prolonged due to garbage collection activity. Actually, in order for the *collectAsMap* stage to be realized, a *mapPartitions* stage has to be completed. This explains why the result represented in the Figure is segmented into two parts; the first part is the *mapPartitions* stage while the second part shows the *collectAsMap* stage. The *collectAsMap* stage collects the results from the *mapPartitions* stage for subsequent processing in the stages that follow.

To further understand performance bottlenecks in these applications, metrics such as the CPU utilization, memory utilization, disk and network I/O as well as the HDFS I/O were collected for the different run scenarios highlighted above. These metrics, averaged over a time series, are shown in Tables 4.6 and 4.7 for the different applications run scenarios. Both applications are memory intensive with over 80% for the *flowerCounter* and about 90% for the *imageClustering* in all run scenarios considered.

Another observation from the results is that CPU utilization is similar in both run scenarios considered for the *flowerCounter* application. For the *imageClustering* application on the other hand, CPU utilization is about 12% higher in the 18 JVM executors scenario in comparison to the 9 JVM executors scenario.

With respect to HDFS I/O metrics, the number of bytes read and written is lowest for the 27 JVM executors scenario in *flowerCounter* and also for the 18 JVM executors scenario in *imageClustering*. In the case of the *flowerCounter* application, about 2.5 GB was read from HDFS for the 27 JVM executors scenario, 3.66 GB and 7.25 GB were read from HDFS for the 18 and 9 JVM executors scenarios respectively. Similarly, for the *imageClustering* application, about 0.76 GB and 0.40 GB was read from HDFS for the 9 and 18 JVM executors scenarios respectively. It appears that the higher the number of JVM executors (and the lower amount of cores per executor - at least two cores per executor), the lower the size of data read by HDFS. The number of bytes written to HDFS is negligible. In conclusion, the 27 JVM executors for the *flowerCounter* application and the 18 JVM executors for the *imageClustering* application have reduced execution time accompanied by relatively high CPU utilization and better HDFS I/O characteristics.

To investigate the influence of the *spark.executor.cores* configuration parameter on the **small cluster** further, more experiments were conducted to understand the effect of this parameter using all the datasets for both applications. As stated in §3.2.2, the configuration parameter *spark.executor.memory* was kept at 7 GB for the applications while the key parameter *spark.executor.cores* were investigated for 1, 2, 4, 8, 12, 16 & 47 cores (1, 2, 4, 8, 12, 16, 47 for *flowerCounter* & 1, 4, 8, 12, 16, 47 for *imageClustering*) respectively. Measurements such as execution time, average CPU & Memory utilization, Disk & Network throughput as well as HDFS IO (disk IOs include the read and write operations by HDFS similarly) obtained for the *flowerCounter* application are shown in Table 4.8 for the July, August and September datasets respectively and that for the *imageClustering* application are shown in Table 4.9. For the *imageClustering* application, metric results are only shown for the July datasets in the table. However, the execution time (and speedup

Table 4.4: Execution Time Summary of 9 JVM Executors for *imageClustering* (128 MB)

Stages	Minimum Time(s)	Median Time(s)	Maximum Time(s)	Total Time(s)
takeSample	498	660	1200	1200
takeSample	1	96	162	162
collectAsMap	9	102	192	198
collectAsMap	3	108	174	180
collectAsMap	5	114	174	174
collectAsMap	4	102	168	174
collectAsMap	3	90	210	210
saveAsTextFile	660	780	960	1020
Total Elapsed Time				3318

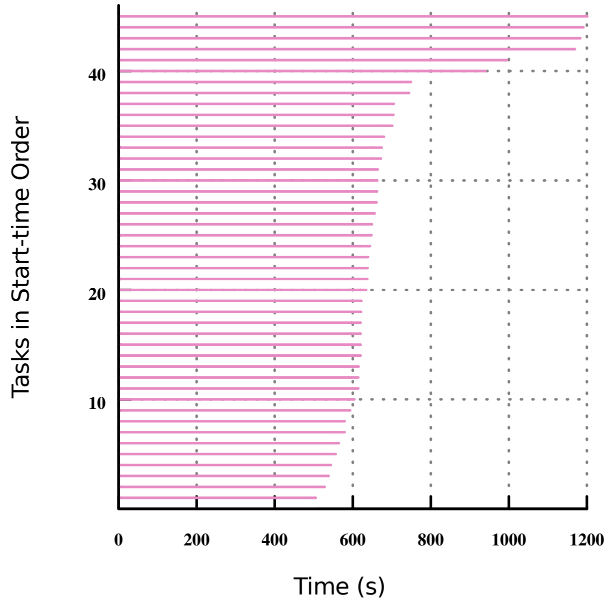
Table 4.5: Execution Time Summary of 18 JVM Executors for *imageClustering* (128 MB)

Stages	Minimum Time(s)	Median Time(s)	Maximum Time(s)	Total Time(s)
takeSample	504	720	780	780
takeSample	47	84	138	138
collectAsMap	48	84	150	150
collectAsMap	52	72	102	102
collectAsMap	50	72	102	102
collectAsMap	50	78	102	102
collectAsMap	50	78	96	102
saveAsTextFile	720	840	1020	1080
Total Elapsed Time				2556

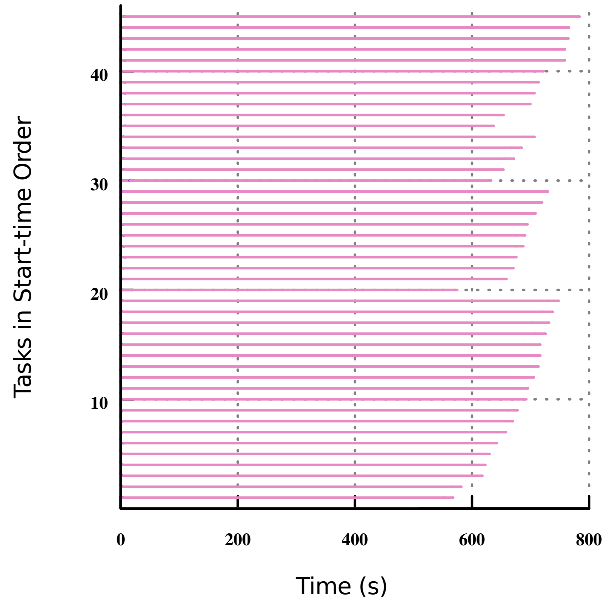
Table 4.6: Average Cluster Resource Usage Metrics for *flowerCounter* (64 MB)

Metrics Measured	**9 JVMs (1/node)**	**18 JVMs (2/Node)**	**27 JVMs (3/Node)**
KbytesWritten/s (Disk)	1132	1106	1141
KbytesRead/s (Disk)	6383	4809	3660
KbytesRx/s	11407	11256	11952
KbytesTx/s	11362	11253	11935
CPU Utilization in %	69	69	73
RAM Utilization in %	88	81	81
HDFS Read Bytes	7247381020	3658918740	2495400050
HDFS Write Bytes	71546	72018	65553

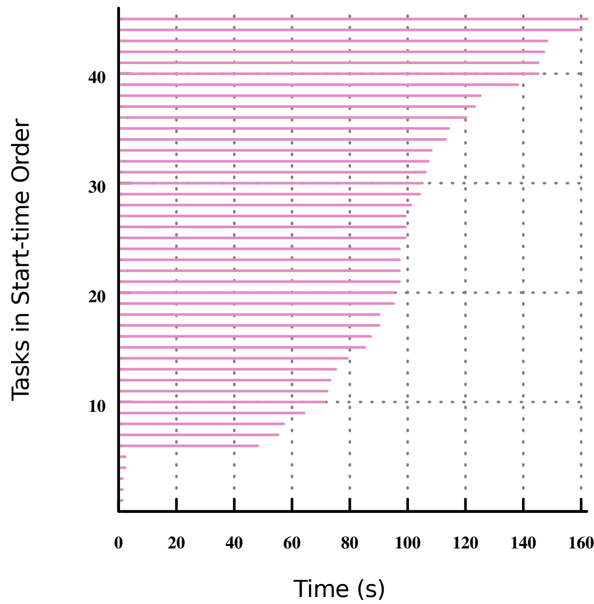
imageClustering (September, 5.7 GB)



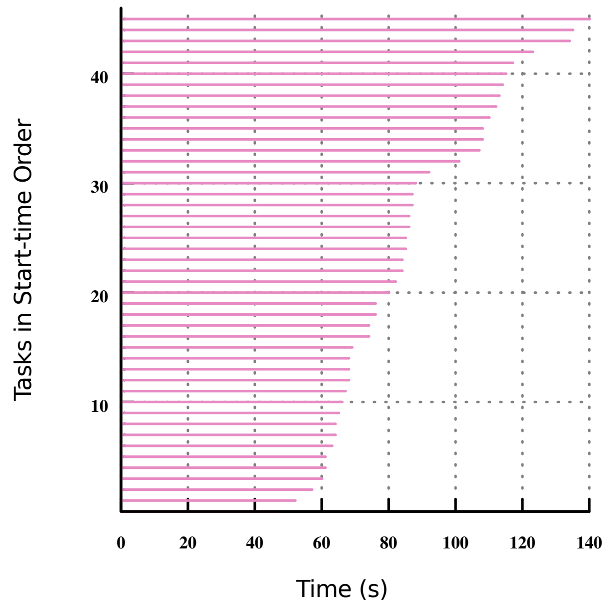
(a) 9 JVM Executors for takeSample-1 (128 MB)



(b) 18 JVM Executors for takeSample-1 (128 MB)

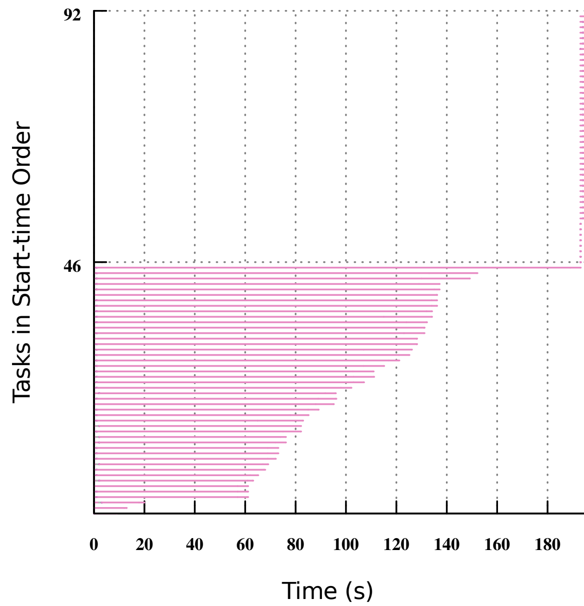


(c) 9 JVM Executors for takeSample-2 (128 MB)

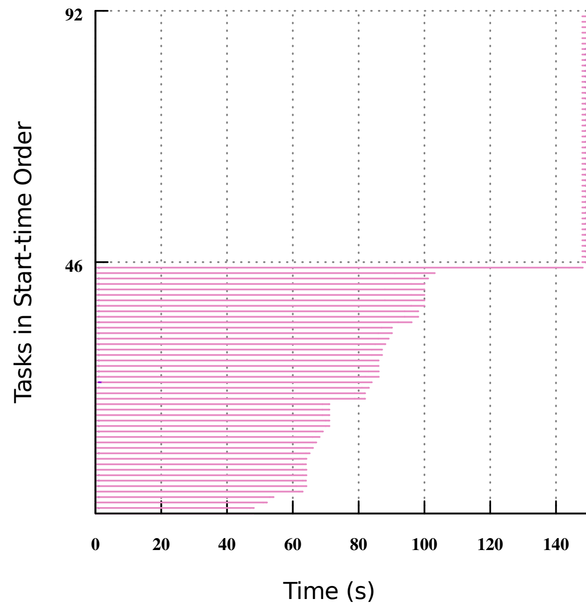


(d) 18 JVM Executors for takeSample-2 (128 MB)

Figure 4.12: Tasks Completion Progress Summary



(a) 9 JVM Executors for collectAsMap-1 (128 MB)



(b) 18 JVM Executors for collectAsMap-1 (128 MB)

Figure 4.13: Task Completion Progress Summary

Table 4.7: Average Cluster Resource Usage Metrics for *imageClustering* (128 MB)

Metrics Measured	**9 JVMs (1/node)**	**18 JVMs (2/Node)**
KbytesWritten/s (Disk)	2990	3936
KbytesRead/s (Disk)	28408	31189
KbytesRx/s	8294	10720
KbytesTx/s	9469	10718
CPU Utilization in %	45	57
RAM Utilization in %	90	89
HDFS Read Bytes	763473945	398618414
HDFS Write Bytes	126925	98547

results) for all the datasets is shown in the Figures 4.14 and 4.15. It is important to state here that the experiment for the *imageClustering* was done with partition size of 8 MB. This is because the application stalled with partition size greater than 8 MB due to recurrent HDFS client errors as explained in §3.2.5.

From the results showed in the table, as the number of executors decreases, and the number of cores per executor increases, memory usage reduces for both applications, except for the instance with 47 cores per executor on the smaller cluster (two executors - one per machine) for the *flowerCounter* application. Similarly, CPU usage in both scenarios reduces as the number of executors decreases except for the runs with two cores per executor using the July dataset.

Data read by HDFS increases as the number of executors decreases especially for the July dataset. This might be due to the limitation of the HDFS in handling concurrent threads (executors with large number of cores) well [13]. This is because Spark launched multiple threads within an executor to read the same data partition simultaneously rather than linearly thus, leading to increased HDFS I/O cost.

For the *flowerCounter* application experimental runs with all the datasets shown in the Table 4.8, the number of bytes transferred and received are similar, but the number of bytes read and written to disk are different. In particular, the number of bytes read from the disk is small, especially in the *flowerCounter* run scenario.

Also, the results indicate that increasing the number of cores with respect to allocated executors reduces application processing time for *flowerCounter* application with all the datasets. It appears that for all the datasets, increasing the number of allocated cores per executor speeds up application processing time up until 4 cores per executor. Subsequent increases in the number of cores per executor did not yield substantial speed advantage. This is because Spark works best for long-running jobs horizontally scaled up on hundreds of nodes and not by increasing the number of threads vertically on just two nodes.

With respect to CPU utilization and memory usage for *flowerCounter* application runs, CPU utilization seems to be negligibly impacted by vertical scaling of cores for the July dataset. Utilization averages are very close in each of the run instances. For the August and September datasets on the other hand, CPU utilization stabilized with 4 cores per executor and greater. Similarly, memory usage averages were almost identical from 4 cores per executor and above. For all the datasets in both applications experimental runs, the average bytes read by HDFS increases with core scaling but with negligible bytes written.

Also, for the *imageClustering* experimental runs, vertical scaling of cores speeds up application processing up untill 16 cores per executor (two executors per node). Runs with one executor per node, each with 47 cores yielded processing time similar to the 16 cores per executor runs. CPU utilization remained almost identical from 8 cores per executor upwards and likewise memory usage averages were comparable after 8 cores per executor. The number of bytes transferred and received stayed equivalent for all the runs considered but the number of bytes read and written to disk were different. The number of bytes read from disk were smaller in comparison to the bytes written to disk especially in the runs with 12, 16 and 47 executors respectively.

Table 4.8: *flowerCounter*: Influence of *spark.executor.cores*

No of Cores Per Executor	No of Executors	Execution Time(s)	Avg CPU %	Avg Memory %	KbTx/S	KbRx/S	KbRead/S	KbWritten /S	hdfsGb Read	hdfsGb Written
July, 34.8 GB Dataset										
1	94	4532	70.68	54.02	45400	45500	0.02	11600	1.10	0.0002
2	46	3605	78.49	37.41	50300	50400	0.01	7550	2.20	0.0001
4	22	3389	75.78	28.10	44800	44900	19.00	4720	4.70	0.0001
8	10	3163	70.10	22.45	46900	46900	0.01	2320	10.40	0.0001
12	6	3055	64.40	19.57	49500	49700	0.01	1970	18.00	0.0001
16	4	3289	57.39	16.64	47700	47800	0.01	1090	26.90	0.0001
47	2	3519	69.19	20.90	45500	45600	269.00	500	52.40	0.0002
August, 19.3 GB Dataset										
1	94	2131	43.76	30.11	37400	37600	0.02	7300	0.20	0.00003
2	46	1581	93.16	32.41	42800	42800	1080.00	210	0.03	0.00003
4	22	1384	57.47	19.38	108000	108000	0.58	38100	0.20	0.00004
8	10	1355	68.22	18.71	56200	56400	0.10	2900	9.90	0.00009
12	6	1328	62.50	14.77	67400	67500	0.00	1760	10.30	0.00009
16	4	1334	56.18	14.17	58700	58800	0.04	1370	23.90	0.00001
47	2	1594	64.55	11.15	55100	55200	47.00	485	28.50	0.00007
September, 5.7 GB Dataset										
1	94	855	95.00	14.96	37300	37300	11.00	200	0.01	0.00001
2	46	832	90.87	22.44	109800	109800	1490.00	240	0.02	0.00006
4	22	513	44.56	16.04	50000	50400	0.02	4720	0.90	0.00002
8	10	475	44.97	10.35	49400	49600	0.02	4380	1.90	0.00002
12	6	468	41.50	9.29	50900	51100	0.02	2320	3.10	0.00002
16	4	493	43.66	6.70	56000	56100	0.00	1280	4.70	0.00003
47	2	519	38.26	5.05	53300	53400	0.01	560	9.20	0.00004

Table 4.9: *imageClustering*: Influence of *spark.executor.cores* (July, 34.8 GB Dataset)

No of Cores Per Executor	No of Executors	Execution Time(s)	Avg CPU %	Avg Memory %	KbTx/S	KbRx/S	KbRead/S	KbWritten /S	hdfsGb Read	hdfsGb Written
1	94	99569	82.89	91.75	41100	42600	17700	13400	0.30	0.0006
4	22	78282	90.32	61.69	84600	84700	11400	26800	1.00	0.0003
8	10	15799	64.35	33.49	41600	41700	16000	12100	3.60	0.0009
12	6	13944	67.22	21.39	56900	57200	1200	18800	5.90	0.0005
16	4	12826	60.72	15.35	58000	58300	800	19300	8.90	0.0005
47	2	13258	61.69	10.24	52100	54900	3700	21200	18.80	0.0007

For the subsequent results, speedup is calculated as

$$Speedup(S) = T_1/T_n,$$

where T_1 is the execution time for one core per executor scenario and T_n is the execution time using n cores per executor accordingly [8].

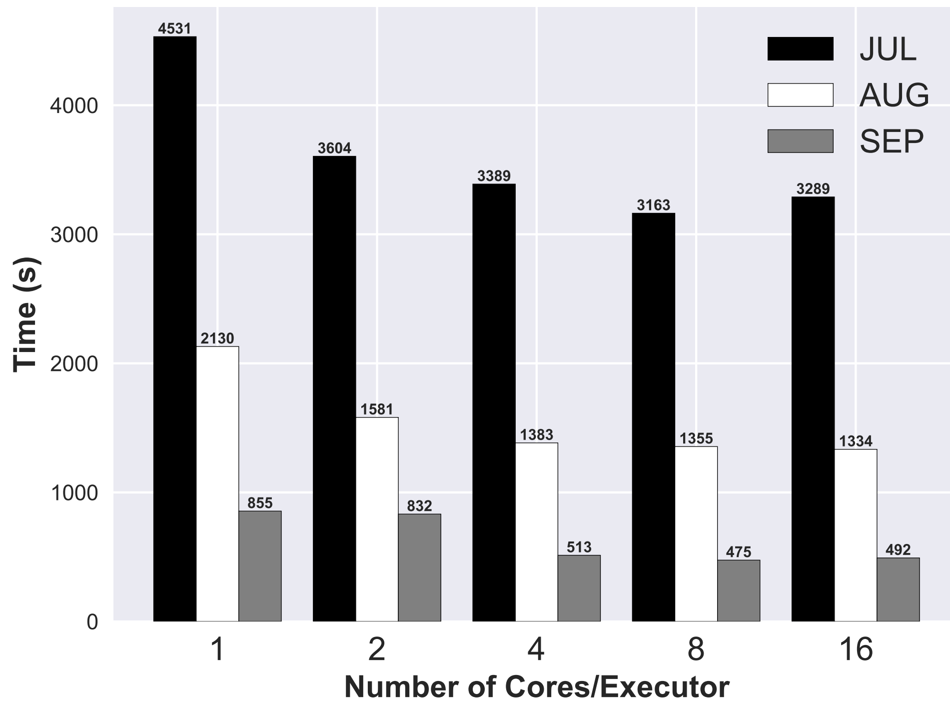
The scaling and speedup results for the execution time are further elaborated in Figures 4.14 and 4.15 respectively. These figures make the results more visible to understand the influence of *spark.executor.cores* parameter on applications processing. As stated above, for *flowerCounter* experimental runs, the processing time is indistinguishable with 4 cores per executor and beyond. This observation becomes clearer with the measured speedup shown in Figure 4.15. The speedup lingers slightly above 1.25 and 1.50 for the July dataset and both the August and September datasets, respectively at 4 cores per executor and greater.

On the other hand, for *imageClustering* experimental runs, the processing time stabilized with 8 or more cores per executor. The influence of *spark.executor.cores* parameter is substantial for the clustering done with the July and August datasets. This could be because of the size of these datasets in comparison to the September dataset (the smallest dataset). The speedup calculated in Figure 4.15 makes the difference more apparent. The speedup is well above 6 and 5 for the July and August datasets, while its just around 2 for the September dataset.

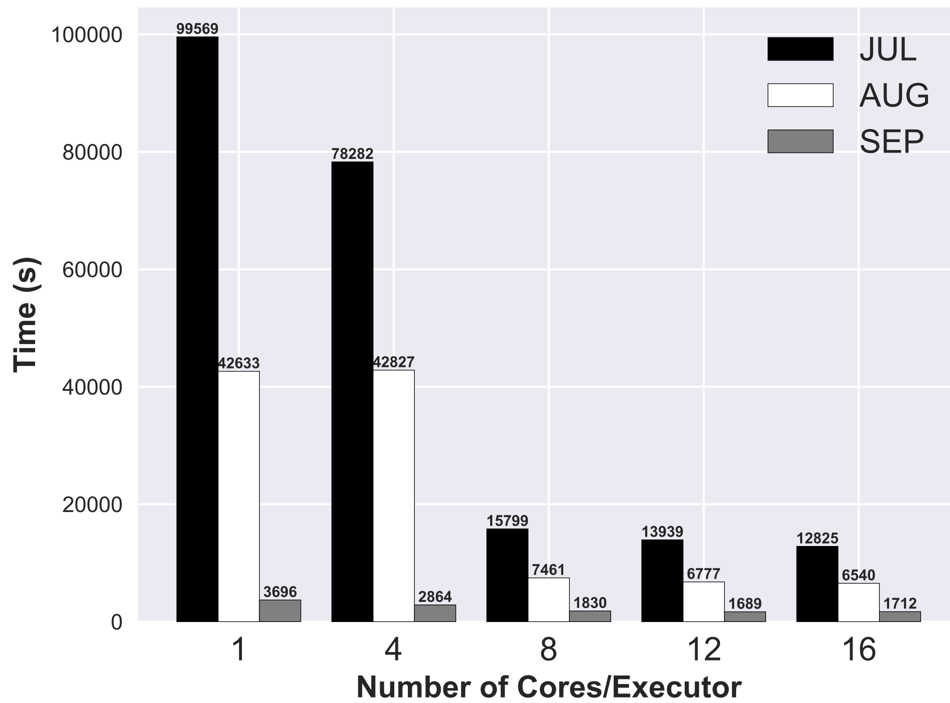
For both applications, a single large (with large number of cores, 47 cores in this scenario) executor resulted in the slowest processing time. This finding agrees with the work done by Chiba and Onodera [12] but contradicts others ([34], [54]). This might be due to the fact that the applications considered in this study are different from those used in the related works. The applications used in those studies are generic benchmarks. The applications used in this study are real-world applications that have varying uses of Spark primitives and RDDs. This makes this study of particular interest, compared to previous work.

A large number of executors (94 executors each with one core) degrades performance for both applications as they result in slow completion time and cause jobs to be CPU and/or memory bound (especially with the July dataset for both applications). This is due to excessive communication overheads caused by the large number of executors [12].

Another interesting observation to note is that as the number of cores per executors increases (and executor number reduces), the jobs become more I/O bound as seen especially by the HDFS read operation. This I/O effect would be different with a cluster with more nodes than that used for these experiments (two nodes) however. For the workloads considered, the *imageClustering* seems to benefit most from the configuration parameter. This is because the parameter favours applications with many tasks (36001 tasks as opposed to 1118 tasks for the *flowerCounter* application) [38].

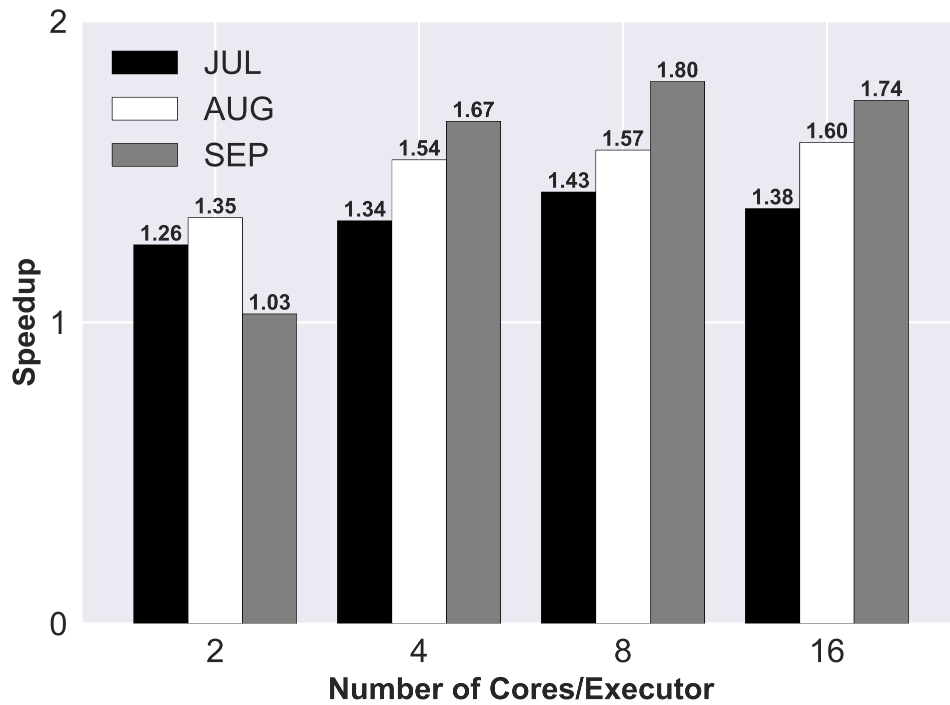


(a) *flowerCounter*

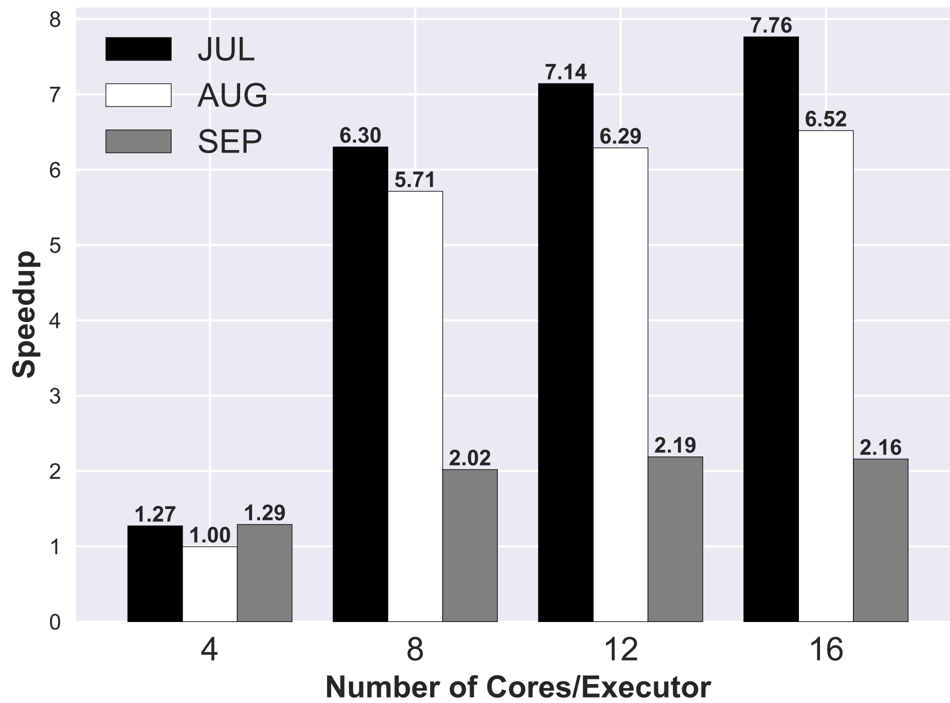


(b) *imageClustering*

Figure 4.14: Execution Time of Core Scaling



(a) *flowerCounter*



(b) imageClustering

Figure 4.15: Speedup of Core Scaling

In summary, the influence of *spark.executor.cores* parameter showed that applications processing speedup steadied at 4 cores per executor for *flowerCounter* application and 16 cores per executor for *imageClustering* application yielded the fastest processing time for all the datasets processed. CPU and memory utilization averages also stabilized at these threshold values. The speedup is more substantial for *imageClustering* application in comparison to *flowerCounter* application. This is due to the differences in the applications especially in the number of tasks and probably because the *imageClustering* application used libraries that are inherently in the Spark engine.

Regarding the influence of the number of executors with respect to allocated cores, findings suggested that for all the datasets processed with the *flowerCounter* application, the threshold settled at about 11 executors per node (2 worker nodes) while for the *imageClustering* application 2 executors per node favoured data processing.

4.3 Impact of Caching

As Spark's engine data abstraction depends heavily on memory, the result here is an attempt to understand the influence of the configuration parameter *spark.executor.memory* on job performance based on the information presented in §3.2.3. Using different configuration memory settings and *spark.executor.cores* fixed at 16, experiments were conducted with the July dataset for both applications but results shown only for the *imageClustering* application on the **small cluster**. Contrary to expectation, increasing the amount of memory allocated for Spark's job execution does not speedup job performance as shown in Figure 4.16. However, some previous studies have shown that allocating more memory for caching RDDs does not always improve job performance as workloads sometimes require dynamically distinguishing job stages that are cache friendly and tune jobs accordingly [34]. The influence of the *spark.executor.memory* configuration parameter on application performance is inconclusive and still requires more detailed analysis.

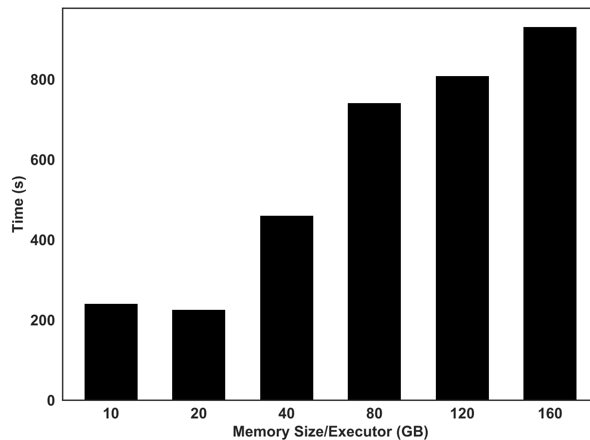


Figure 4.16: Influence of *spark.executor.memory* Parameter

Therefore, the effect of caching the input RDD (input data) using the different storage levels on the applications' execution speed were studied. For the *flowerCounter* application, the input RDD was cached using storage levels including MEMORY_ONLY, DISK_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER and MEMORY_AND_DISK_SER while the *imageClustering* application used DISK_ONLY, MEMORY_AND_DISK and MEMORY_AND_DISK_SER storage levels to cache the input RDD. The MEMORY_ONLY storage level instance failed due to *out-of-memory* exception for the *imageClustering* application. The results from the experiments conducted are shown in Figure 4.17 for only the *imageClustering* application with the September dataset. Results show that MEMORY_AND_DISK storage level yielded the least runtime in comparison with the other storage levels considered. The processing time obtained using *flowerCounter* application on the July dataset is almost identical for all the storage levels considered as shown in Table 4.10.

In order to understand these results, detailed application metrics were collected including the *runTime*, *duration* (sum of individual tasks time), *schedulerDelay*, *executorRunTime*, *executorCpuTime*, *executorDeserializeTime*, *resultSerializationTime*, *gettingResultTime*, *jvmGCTime*, disk I/O and shuffle I/O (shuffle read cost is network I/O) metrics as shown in Table 4.10.

The results in Table 4.10 show that storage levels do not have any effect on disk and network I/O metrics as these remained constant across all levels. In the *flowerCounter* experimental runs with the July dataset on the larger cluster, 7.90 GB of data were read from 375245 records and 18.09 MB output data were written to disk from 93707 records or images in all the storage levels investigated. Likewise, the number of bytes sent and received via the network remained constant at about 967 KB across all the storage levels studied. Also, the data processing time is unaffected by the different storage levels for the same *flowerCounter* application.

For all the storage levels, the *executorRunTime* contributed substantially to the total *duration* or *taskTime*. These results also show that the *executorCpuTime* is less than 1% (approximately 0.5%) of the *executorRunTime* for all the storage levels and similarly the *jvmGCTime* is less than 10% of the *executorRunTime* for all levels considered.

This clearly indicates that the executor is not heavy on CPU utilization but the usage reported by the operating system shows that the application is high on both CPU and memory. The obvious explanation for this dynamics could be unravelled by the *executorRunTime* definition in Equation 3.4. This shows that the *executorRunTime* is high largely due to the time the tasks spent in reading data from the disk - *taskIOReadTime* (the *taskIOWriteTime* is substantially low as the data written to disk is very small in comparison to the data read). Evidently, over 90% of the *executorRunTime* is dominated by disk I/O time cost across all the storage levels considered.

On the other hand, the result from the *imageClustering* experimental runs is shown in Table 4.11. For all the experiments conducted, MEMORY_AND_DISK storage level yielded the least *runTime* in comparison to the other two storage levels considered. Again, the storage levels did not have substantial influence on the disk and network I/O metrics; similar results were obtained with all the storage levels. The number of bytes

***imageClustering* (September, 5.7 GB) - Small Cluster**

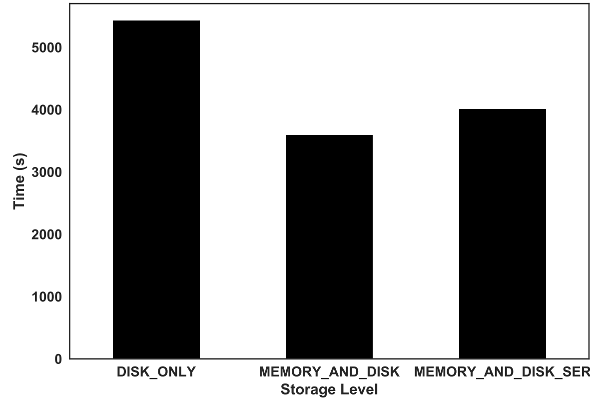


Figure 4.17: Runtime vs Storage Level

read and written from disk was approximately 2 T and 2 MB respectively while also about 12 MB of data were read via the network for the storage levels studied. Similarly, as in the *flowerCounter* runs, the disk I/O (mostly the read cost) contributed over 80% of the *executorRunTime* in comparison to the *executorCpuTime* and the *jvmGCTime* for all the storage levels.

As cached RDDs help to ensure that data is close to the processing node by using different data locality placement algorithms as discussed in §2.21, data locality scheduling for the different storage levels was studied to understand what might be responsible for the differences in the *runTime* reported. The locality level summary observed for both applications is shown in Tables 4.12 and 4.13 respectively. For *flowerCounter* application, all the storage levels considered showed very similar *runTime* except with the MEMORY_AND_DISK_SER storage level whose *runTime* was slightly higher than the other storage levels. The processing time obtained for all the storage levels are very similar because of identical task locality levels.

With the *imageClustering* application on the other hand, MEMORY_AND_DISK storage level exhibited the least *runTime* because of differences in the task locality levels. The locality level results show that tasks with the PROCESS_LOCAL (which is the fastest) task level are the highest in the *imageClustering* application runs for the MEMORY_AND_DISK storage level and similarly that tasks with the ANY (which is the farthest and slowest) task level are the least for the same MEMORY_AND_DISK storage level. This behaviour explains why the execution speed is fastest with the MEMORY_AND_DISK storage level.

Table 4.10: *flowerCounter*: Influence of Caching (July, 34.8 GB Dataset)

Metric Type (s)	MEMORY ONLY	DISK ONLY	MEMORY & DISK	MEMORY ONLY_SER	MEMORY & DISK_SER
runTime	4700	4700	4700	4800	5300
duration	243100	236100	235100	236500	251200
schedulerDelay	240	170	230	150	220
executorRunTime	242600	235700	234600	236200	250700
executorCpuTime	1080	1120	1110	1110	1130
executorDeserialize Time	240	210	290	150	190
resultSerialization Time	4	6	13	6	27
gettingResultTime	9	8	9	9	8
jvmGCTime	17600	19800	19400	19900	22200

In summary, the effect of using the *spark.executor.memory* parameter to control the amount of memory allocated to Spark jobs do not favour application completion speed. Completion time increases as the amount of memory allocated to Spark jobs increases. This is in contrary to expectation as the Spark engine relies substantially on memory for fast job completion and thus increase in memory allocation should sped up job processing. This behaviour might be due to resource contention as its control is handled by the operating system and not by the Spark engine. However, vertical scaling of physical memory as well as the memory size allocated to Spark jobs on the cluster nodes might have speed up application processing substantially. The influence of the parameter on application processing still requires more detailed analysis to understand its influence on application performance.

For the influence of storage levels on application performance, MEMORY_AND_DISK storage level exhibited the fastest runtime for the *imageClustering* application due to tasks with the highest number of the fastest locality level of PROCESS_LOCAL. For *flowerCounter* application, the MEMORY_AND_DISK_SER has the slowest runtime due to tasks with the highest number of ANY task locality level while for the *imageClustering* application, the DISK_ONLY storage level has the slowest runtime due to the same reason as in the *flowerCounter* scenario.

Table 4.11: *imageClustering*: Influence of Caching (September, 5.7 GB Dataset)

Metric Type (s)	DISK ONLY	MEMORY & DISK	MEMORY & DISK_SER
runTime	5400	3600	4000
duration	219400	158400	158900
schedulerDelay	990	390	340
executorRunTime	217900	157400	158100
executorCpuTime	13770	12830	13190
executorDeserializeTime	550	570	460
resultSerializationTime	6	4	4
gettingResultTime	0	0	0
jvmGCTime	17760	17610	12820

Table 4.12: *flowerCounter*: Locality Level Summary (July, 34.8 GB Dataset)

	Task Locality Level Count Summary				
Locality Level	MEM ONLY	DISK ONLY	MEM & DISK	MEM ONLY_SER	MEM & DISK_SER
NODE_LOCAL	225	224	225	225	224
ANY	115	109	104	121	124
PROCESS_LOCAL	778	785	789	772	770

Table 4.13: *imageClustering*: Locality Level Summary (September, 5.7 GB Dataset)

	Task Locality Level Count Summary		
Locality Level	DISK ONLY	MEMORY & DISK	MEMORY & DISK_SER
NODE_LOCAL	547	568	487
ANY	285	145	240
PROCESS_LOCAL	6169	6288	6274

4.4 Compute Node Scaling

Spark is a scale-out analytics engine, that is, it is designed to scale well with increasing number of cluster nodes. This section contains results from the experiments conducted to validate the scale-out design paradigm of Spark for only the *flowerCounter* application. The experiments were executed on the larger cluster with 11 nodes using all the datasets. As seen in Figure 4.18, the jobs do not exhibit absolute linear scalability but show a quasi-linear trend up to 9 nodes.

As the number of nodes utilized for the *flowerCounter* application increases, the processing speed increases at nearly a constant rate for all datasets, especially with the July and August datasets until 9 worker nodes. The speedup obtained with the September dataset is not as consistent as with the other July and August datasets. This could be because the September dataset is smaller and does not contain images with flowers. Spark was actually designed to perform well with huge datasets.

Also, it is important to state that the preliminary version of the *flowerCounter* application was used for this study. The application actually requires that a job should be equivalent to the processing of a day's worth of images from one camera. A day's worth of images from a single camera is just about 400 MB. This is not big enough for Spark if a cluster is devoted to only this application. Therefore, the images were coalesced to make them representative in terms of datasize. Coalescing these images could be the reason why the algorithm is not working well in terms of accuracy. This is because processing might be done on images that do to need to be compared with one another.

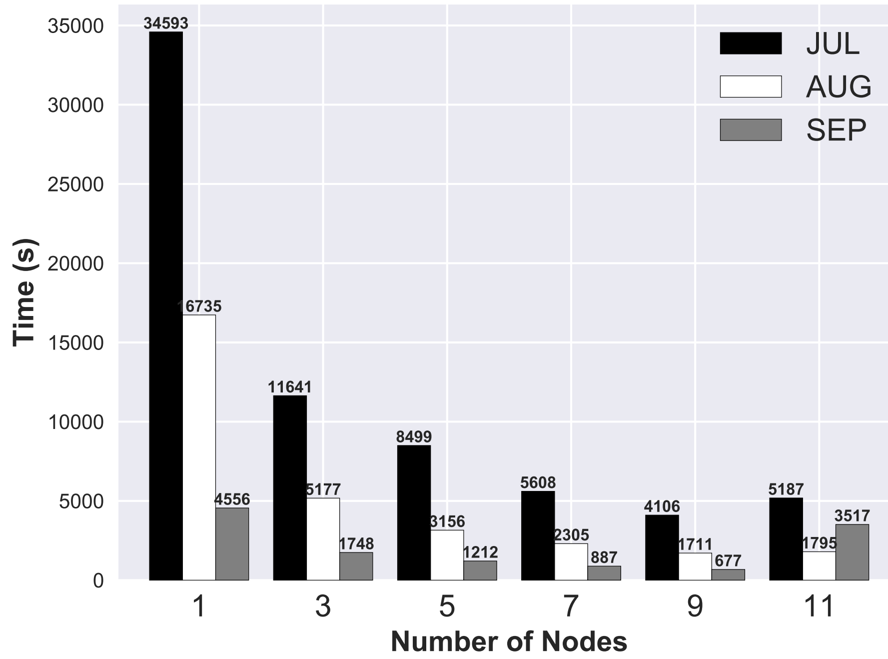
Another deployment scenario would be to investigate large number of parallel jobs of each type sharing a cluster or have higher resolution images as the input dataset. Furthermore, images could be coalesced for a particular day from multiple cameras or orders of combination of images.

The execution time for 11 nodes increased more than that of the 9 nodes scenario. This behaviour is due to the fact that 9 of the compute nodes have twice the CPU frequency of the remaining two nodes (*mario* and *luigi*). This observation, for example with the July dataset, is corroborated by the task execution time summary across all the nodes in Table 4.14 as well as in Figure 4.19. These results show that *mario* and *luigi* executed the least number of tasks but exhibited the slowest job completion time. All other nodes except *mario* and *luigi* showed similar processing time.

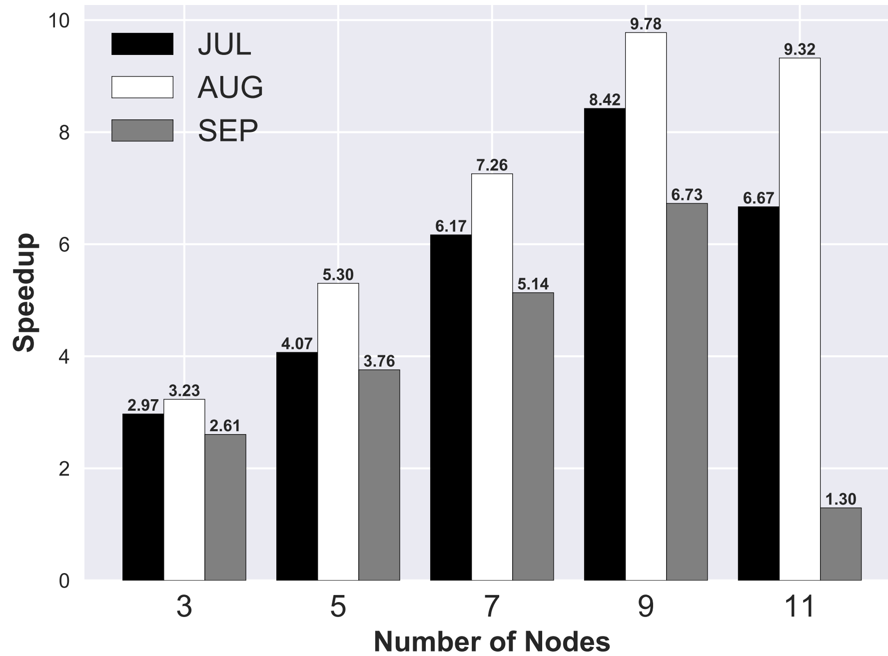
The sharp decline in speedup from 9 nodes to 11 nodes, especially with the July and September datasets is exemplified in Figure 4.19. The figure shows that the decline is due to slow tasks on *mario* and *luigi.* This behaviour of prolonged jobs is typical of heterogeneous environments and inherent tasks schedulers fail to handle this dynamically without affecting the overall job performance [11]. Thus, this makes obvious the need to investigate the speculative scheduler implemented in Spark. Speculative execution forms the basis of the experiments in the next section.

One of the major motivations behind this study was to investigate the performance of the Spark applications with the original sequential applications. To this end, sequential experiments were conducted for only

***flowerCounter* Large Cluster**



(a) *Execution Time*



(b) *Speedup*

Figure 4.18: Effect of Node Scaling

Table 4.14: *flowerCounter*: Tasks Execution Time Summary on all Nodes (July, 34.8 GB Dataset)

Host	Minimum Duration(s)	Average Duration(s)	Maximum Duration(s)	Total Duration(s)	No of Tasks
mario	244	1142	2564	38851	34
luigi	194	796	1284	32600	41
worker1	22	178	723	21900	123
worker2	5	185	850	22800	123
worker3	8	195	798	20900	107
worker4	25	183	885	21200	116
worker5	29	186	929	22000	118
worker6	20	187	972	21500	115
worker7	28	191	1074	20800	109
worker8	3	188	826	20900	111
worker9	19	182	898	22100	121

***flowerCounter* (July, 34.8 GB) - Large Cluster**

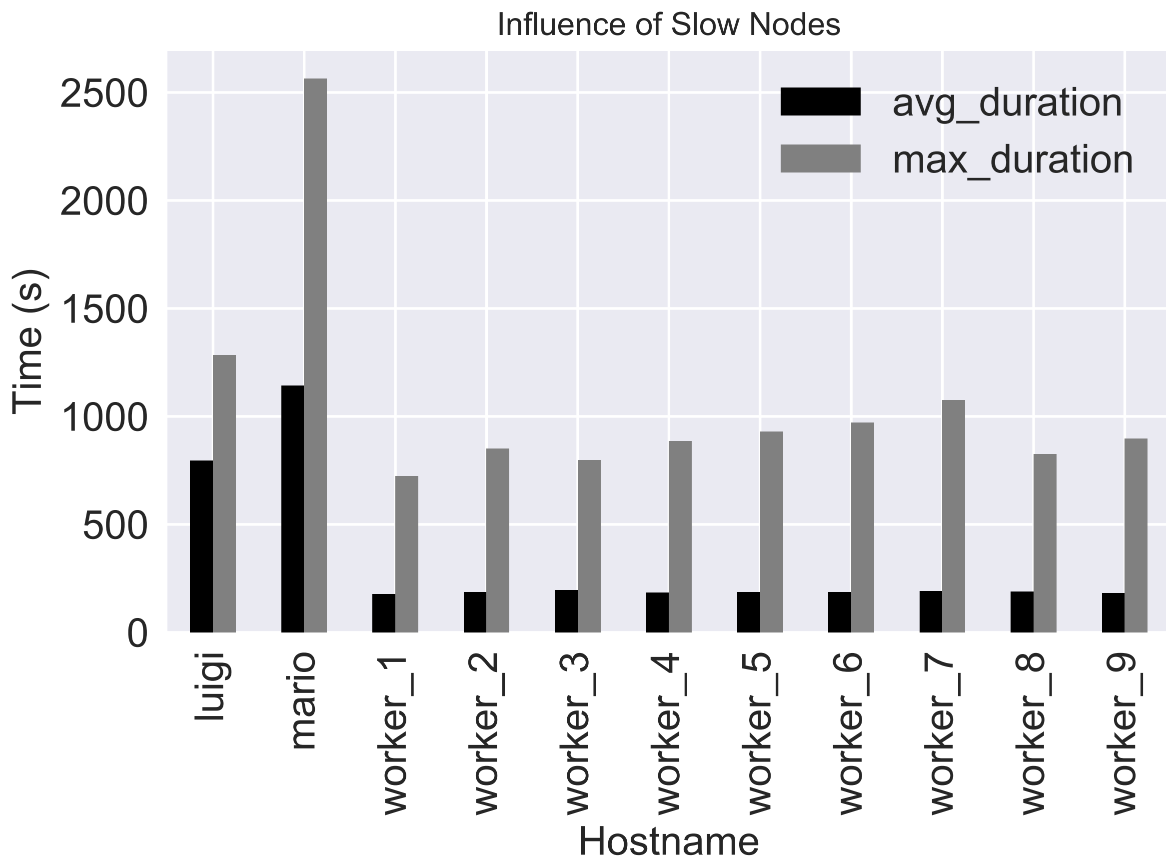


Figure 4.19: Influence of Slow Nodes

the *flowerCounter* application using all the datasets and results obtained in comparison with the parallel Spark execution on both the small (2-node workers) and the large clusters (9-node workers) is shown in Table 4.15. The results show that the Spark application executed in a cluster of computing nodes outperformed the sequential version implemented on the single server machine called *onomi*. The speedup obtained on the 9-node cluster is approximately 3 times the sequential execution for all the datasets with a similar number of compute cores devoted to the work. Additional cores were dedicated to the operating system for managing the Spark environment. Similarly, the speedup on the smaller cluster is approximately 4 times the sequential setup for all the datasets with about 8 compute cores more than the sequential setup used for the experiment (2 executors per node each with 16 cores giving a total of 64 cores on the two nodes).

Table 4.15: Runtime Comparison between the Sequential & Spark *flowerCounter* applications[a]

Month	Onomi (s)	9-Node (s)	9-Node Speedup	2-Node (s)	2-Node Speedup
September	2144	677	3.2	493	4.4
August	6206	1712	3.6	1334	4.7
July	13894	4107	3.4	3290	4.2

[a]The single node *Onomi* has 56 cores, the 9-Node Cluster has 54 cores and the 2-Node Cluster has 64 cores. The speedup is calculated with respect to the runtime on *Onomi*, that is, the runtime on *Onomi* divided by the runtime on the 9-Node and the 2-Node setups respectively

In summary, Spark exhibited a near-linear scalability with all the datasets using *flowerCounter* application up untill 9 nodes. The speedup obtained dwindled with the addition of two nodes (*mario* and *luigi*) whose performance characteristics differ from the other 9 nodes. The slow speedup obtained with the 11 nodes is because *mario* and *luigi* executed the least number of tasks but with large variance in tasks completion time. Findings also showed that the Spark version of *flowerCounter* outperformed the original sequential version with approximate speedups of 4 and 3 on the 2-node and 9-node Spark workers respectively.

4.5 Speculative Task Execution

Slow tasks are expected in a heterogeneous cluster environment due to varying system capacity and traditional schedulers are not designed to mitigate their effects on workload performance effectively. The results obtained from the node scaling experiments clearly validate this premise. Speculative engine is implemented in Spark to detect slow tasks and then mitigates their effect by adding them back to the tasks queue for rescheduling. If the original slow task finished before the rescheduled back-up task, the back-up task would be killed and vice-versa. Speculation, however, does not automatically make all workloads process faster, especially real-time streaming workloads due to processing delay caused by tasks rescheduling (and increased number of tasks) [38].

Speculation experiments (implemented by setting the configuration parameter *spark.speculation=true*) were carried out on both application workloads using all the datasets to study its effects on straggler tasks

on some cluster nodes, notably *mario* & *luigi*. Results from the experiments (3 run instances for each dataset in both scenarios) conducted for the *flowerCounter* application are shown in Figure 4.20. It is important to note that the speculation experiment was conducted only for the 11 nodes scenario and not for the other node setup instances. These results show that speculation slightly favours the *flowerCounter* application with all the datasets. Speculation is substantial for the July and September datasets, but has almost no effect with the August dataset. Speculative did not substantially reduce the execution time of the application with the August dataset probably because most of the tasks finished on time and there was no need for tasks to be speculated. On the other hand, for the *imageClustering* application using the July dataset only, speculation yielded an execution time of about 624 minutes as against 612 minutes obtained without speculation. Here, though the processing time is slightly reduced with speculation enabled, the difference is not very substantial.

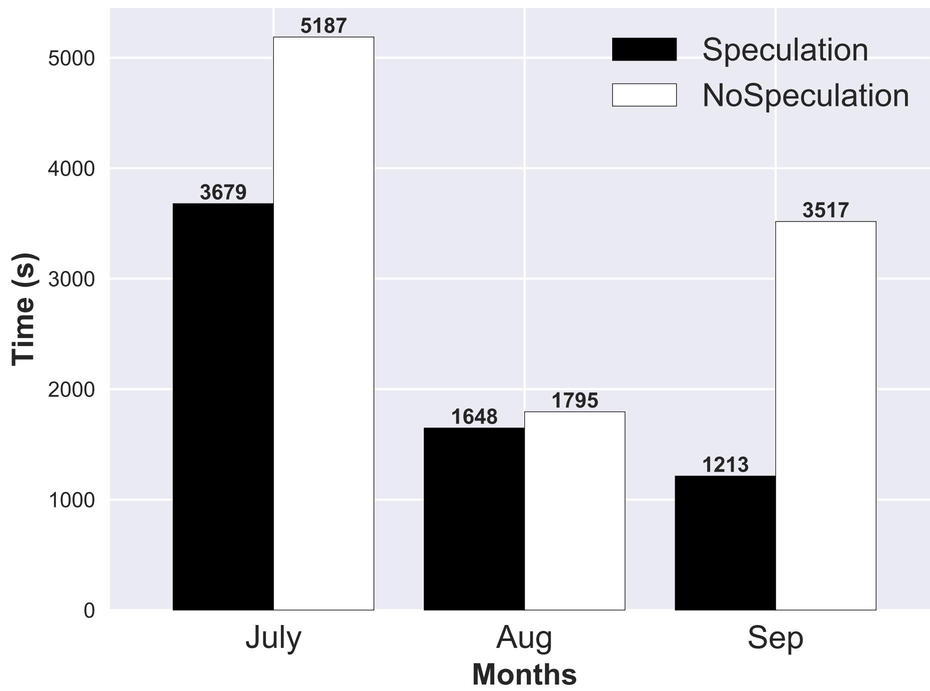


Figure 4.20: Speculative and non-Speculative *flowerCounter* appilcation execution

Another reason could be because of incorrect median time calculation, as potential tasks to execute are identified when their time is more than the median time of the already completed tasks. This could be because the threshold value that must be exceeded before a task can be marked for relaunching was inappropriate for the applications. The threshold value is calculated as the product of the *spark.speculation.multiplier* and the median of the already completed tasks. The *spark.speculation.multiplier* value used for the experiments was the default value of 1.5. Also, this could be misleading because the median calculation does not take into account the computing power of the cluster nodes especially in a heterogeneous environment. Speculative

execution does not work effectively in all scenarios because tasks added to the queue could still be launched on the slow nodes whose tasks have been previously identified as slow tasks [60].

In order to understand the influence of the speculation parameter, the tasks' execution time was analyzed and the summary for the *flowerCounter* application is shown in Table 4.16. Comparing these results to that obtained without speculation in Table 4.19, the same machines (*luigi* & *mario*) still remained the slowest machines as is the case without speculation apart from *discus-p2irc-worker8*. However, speculative execution helps to reduce the average time per task substantially by about 33% and 81% on *luigi* & *mario* respectively. To buttress the findings made by Yang *et al.* [60] that speculative execution in Spark has no way of identifying slow nodes so as not to reassign slow tasks to them, the slow nodes (*mario* & *discus-p2irc-worker8*) still have tasks assigned to them. The column indicating the number of speculated tasks represents the tasks that were placed on the nodes after the original tasks processing time exceeded the median time of the already completed tasks.

On the other hand, for the *imageClustering* application, the execution time for both the non-speculation and the speculation scenarios is shown in Tables 4.17 and 4.18 respectively. The non-speculative scenario shows that *luigi*, *worker6* & *worker8* are the slowest machines with average execution time per task of 148 seconds, 77 seconds and 68 seconds respectively. For the speculative scenario, the slowest nodes are *worker4* & *worker5* with average execution time per task of 1.50 minutes & 1.45 minutes respectively. Both *mario* and *luigi* have over 2x the number of tasks associated with every other node, except *worker2*. *worker5* has the lowest maximum, but almost the highest average. It is unclear the reason for the difference in the behaviour of the individual nodes. It requires more replication and investigation.

Again, the slow nodes still got speculated tasks as was the case with the *flowerCounter* execution scenario. However, the speculative execution took 12 minutes longer to execute than the non-speculative scenario. This unexpected behaviour might be due to increase in the number of tasks due to speculation and probably because the task partition size (8 MB partition size) was too small which could lead to unfavourable task communication overheads. This can be clearly seen in the minimum execution time in the tables as some tasks finished less than 20 milliseconds (minimum task times represented as 0.0 second actually finished much less than 20 milliseconds) as against the recommended least minimum task time of 100 milliseconds [55]

In summary, speculative task execution helps to mitigate the effect of straggling nodes (*luigi* and *mario*) in the cluster by reducing the average execution time per task by about 33% and 81% respectively in the *flowerCounter* experimental runs using all the datasets. The effect of the speculative algorithm was more substantial with the July and September datasets; processing time was unaffected by speculative execution with the August dataset. However, in the *imageClustering* experimental setup, speculative execution slightly increased processing time by 2% in comparison to the non-speculative scenario with only the July dataset. This could be because of the increase in the number of tasks due to speculation and the small data partition size used. In both applications runs, speculative execution in Spark still placed slow tasks on the straggling nodes. The speculation policy failed to identify and isolate slow nodes.

Table 4.16: *flowerCounter*: Tasks Execution Time Summary with Speculation (July, 34.8 GB Dataset)

Host	Minimum Duration(s)	Average Duration(s)	Maximum Duration(s)	Total Duration(s)	No of Tasks	No of Speculated Tasks
mario	25	216.0	1410	21150	98	3
luigi	39	532	1120	17550	33	0
worker1	22	185	840	20310	110	5
worker2	8	155	870	19270	124	1
worker3	13	159	750	20680	130	3
worker4	25	205	830	20480	100	1
worker5	31	189	920	20010	106	1
worker6	25	173	1020	19930	115	2
worker7	4	206	890	20570	100	4
worker8	38	266	820	19410	73	1
worker9	7	157	1100	20270	129	2

Table 4.17: *imageClustering*: Tasks Execution Time Summary without Speculation (July, 34.8 GB Dataset)

Host	Minimum Duration(s)	Average Duration(s)	Maximum Duration(s)	Total Duration(s)	No of Tasks
mario	0.0	37	1740	177920	4758
luigi	0.0	148	3250	177480	1200
worker1	0.3	59	1330	184910	3121
worker2	0.6	49	1443	185260	3782
worker3	0.0	49	1260	190260	3892
worker4	0.0	47	2800	193000	4108
worker5	0.1	62	2420	184750	2977
worker6	0.0	77	2880	168260	2166
worker7	0.2	53	1680	195620	3657
worker8	0.0	67	1480	177790	2660
worker9	0.2	52	1930	192080	3680

Table 4.18: *imageClustering*: Tasks Execution Time Summary with Speculation (July, 34.8 GB Dataset)

Host	Minimum Duration(s)	Average Duration(s)	Maximum Duration(s)	Total Duration(s)	No of Tasks	No of Speculated Tasks
mario	0.0	36	2210	200980	5598	47
luigi	0.0	35	1260	204740	5874	70
worker1	1.6	70	3480	191030	2701	3
worker2	0.4	47	2280	210150	4515	68
worker3	0.0	62	2510	194680	3127	52
worker4	0.0	90	2960	192960	2137	43
worker5	0.2	87	1690	164800	1898	21
worker6	0.0	80	3580	195830	2441	12
worker7	0.0	68	3200	200880	2926	33
worker8	0.0	75	4266	189030	2523	40
worker9	0.0	77	2450	176470	2280	29

4.6 Chapter Summary

In summary, these experiments proved that selecting the right combination of configuration parameters is critical to the performance of Spark applications. For example, using the *spark.files.maxPartitionBytes* paramater to control the data partition size and the level parallelism showed that 64 MB data size speed up *flowerCounter* application processing more than 128 MB partition size. This is because there are few stragglers and reduced CPU wait time with the 64 MB partition size especially in the *computeHistogram* and *computeHistogramShifts* stages of the *flowerCounter* application.

With the *spark.executor.cores* configuration parameter for controlling the amount of thread allocated for tasks within an executor, the processing speed of both example applications improved until about 4 cores per executor (or 12 cores per executor for *imageClustering* application) for all the datasets used in this study. Subsequent increase in the number of cores per executor did not speed up job completion time because Spark does not perform well with small number of nodes and short running jobs. Also, vertical scaling of cores per executors on a single node did not speed up applications processing because many threads initiated by Spark have the tendency of reading the same data partition size simultaneously, thus causing bad HDFS I/O.

Experiments with the *spark.executor.memory* to investigate the memory abstraction of Spark was inconclusive. This is because increase in the amount of memory allocated for the Spark applications did not speed up job completion time. However, experiments showed that using the right storage level policy for caching input dataset might improve performance. The storage level policy ensured that data are placed close to the processing code using appropriate locality preference especially for the *imageClustering* application.

Findings also demonstrated that using a cluster of commodity personal computers for processing Spark applications is faster in comparison to using a single machine with similar performance characteristics as all the nodes in the cluster combined. This scalability result agrees with the scale-out design paradigm implemented in Spark.

In a heterogeneous cluster environment, the effect of stragglers must be mitigated for better job completion time by speculative execution with the *spark.speculation* configuration parameter. Speculative execution reduced job completion time by rescheduling tasks to other available nodes. This speculative policy setting works better for the *flowerCounter* application processing than the *imageClustering* processing. This is because speculative execution was able to reduce the number tasks allocated to the slow nodes and thus, speed up the job completion time.

Finally, the *flowerCounter* application was developed with the notion that a job is the daily processing of images and not several days images coalesced together. Coalescing these images, as was done in this study, could have affected the accuracy of the *flowerCounter* application algorithm. It is therefore important that experimental methodologies in subsequent studies are designed to reflect this algorithm consideration.

Chapter 5

Conclusion and Future Work

5.1 Summary

The goal of this thesis was to evaluate how Apache Spark system performs processing two real image processing applications namely *flowerCounter* and *imageClustering* using images collected in July, August and September of Summer 2016 from selected canola plants growing plot fields in 2 heterogeneous cluster environments. The experiments in this thesis particularly explore the influence of Spark's configuration parameters on job completion time. The particular parameters investigated include *spark.files.maxPartitionBytes*, *spark.executor.cores*, *spark.executor.memory* (including the impact of the different storage levels on execution speed) and *spark.speculation*, respectively.

Firstly, the influence of partition size (studied with the *spark.files.maxPartitionBytes* parameter) on task execution time shows that the partition size of 64 MB performed better in terms of execution speed than other partition sizes considered. This is due to increase in the number of tasks (increased level of parallelism) which resulted in reduced slow tasks and better utilization of cluster resources such as CPU, memory, network and disk throughput. Further experiments also indicate that further reduction in partition size smaller than 64 MB (which led to increase in the number of tasks) did not reduce execution time. This is due to communication delays caused by the task scheduler as a result of the large amount of tasks. These findings confirm the related works reviewed.

Next, the impact of the number of JVM executors with respect to allocated CPU cores was studied using the *spark.executor.cores configuration parameter*. For the *flowerCounter* application using the smaller cluster, the processing time stabilized at 11 JVM executors with 4 cores each for all the datasets. Two JVM executors with 16 cores each outperformed the other configuration settings considered in the case of the *imageClustering* application. For both applications, large number of executors each with one core (94 executors each with one core) yielded the worst performance and one big executor with 47 executors did not give the best execution speed contrary to finding in some related works. Previous studies showed that having many small executors limits the maximum number of simultaneous tasks within an executor to one and thus jeopardizes some benefits Spark provides. Many executors also result to inefficient use of cluster resources due to communication overheads between executors. On the other hand, running jobs with large executors (executors with large amount of resources) do not yield optimal performance due to delays caused

by garbage collection and limitations of the HDFS in handling concurrent threads effectively. Executors with large number of cluster resources is a wasteful job configuration. Determining the right size and number of executors on cluster nodes is essentially a bin-packing issue similar to the Np-complete knapsack problem [29] and thus heuristic approaches are required for determining optimal executor configuration parameters for the different applications Spark supports.

Furthermore, the influence of the cache mechanism implemented in Spark was studied by increasing the amount of Spark memory using the *spark.executor.memory* configuration parameter. Results obtained showed that increasing memory allocated to Spark negatively (execution speed increases with increasing memory allocation) impacts job completion time for the *imageClustering* application using the July dataset. This might be due to contention from the operating system. This requires a detailed study to really understand the effect of the parameter on job completion. However, further studies conducted to understand influence of the different storage levels (for caching RDDs) show that the MEMORY_AND_DISK storage level yielded the fastest execution time among the levels considered, due to better task locality level placement with the highest number of the fastest level (PROCESS_LOCAL) and least number of the slowest level (ANY). Results also show that the high *executorRunTime* was substantially dominated by the tasks' disk I/O time cost and not the CPU time cost of running the tasks (the high CPU usage observed was due to the disk I/O cost and not the CPU cost of performing the tasks by the executor - *executorCpuTime* is less than 10% of the total *executorRunTime* for both applications).

In addition to the cache mechanism experiments, the scale-out design paradigm of Spark was examined for the *flowerCounter* application using all the datasets. Spark exhibited a quasi-linear scale ratio up to 9 nodes with all the datasets but the scale plunged deeply for 11 nodes from the 9 nodes scenario. The reason for this behaviour is that two of the nodes have CPU frequency that is half of the other 9 nodes and thus resulted in straggling tasks that prolonged job completion time. The performance of the sequential and Spark versions of the *flowerCounter* application was considered using the large and small cluster and the one big machine (*onomi*) for all the datasets. The Spark version of the *flowerCounter* application showed speedup of about 3 and 4 on the large and small clusters respectively more than the sequential version of the application executed on the single machine, *onomi*.

Finally, the speculation task scheduling of Spark was turned on with the *spark.speculation* parameter to mitigate the effect of the slow nodes on job performance for both applications using all the datasets for the *flowerCounter* application and only the July dataset for the *imageClustering* application. Speculation execution slightly favoured the *flowerCounter* application with a reduction in execution time by 29%, 8%, 66% respectively for all the datasets. For the July dataset, for example, speculative execution reduced average time per task by 33% and 81% on the slow nodes (*mario* and *luigi*) in comparison to the execution time obtained for the non-speculation scenario. On the other hand, for the *imageClustering* scenario, speculation increased execution time by about 2% more than the non-speculation instance. This might be due to the increase in the number of tasks caused by the speculative execution and large number of small tasks (thereby

leading to delays due to communication overheads) whose completion time was far less than the minimum recommended rule of thumb time of 100 milliseconds.

5.2 Contributions

The contributions and recommendations from this study represent some insights and knowledge discovered in the course of this thesis study. The insights cover a wide range of different aspects involved in the thesis work including cluster administration, resource management, monitoring, telemetry and application execution.

- This study investigates cluster operation with Spark in the context of two image processing applications.
- This study quantifies the effect of key Spark configuration parameters on cluster resource utilization and application processing time.
- For effective cluster administration and workloads execution, automate running of the different workloads and cluster setup tasks with actionable scripts in any chosen language of choice such as Python, Ansible,[1] or Shell scripting. This would help in quickly bootstrapping the cluster and scheduling jobs automatically.
- Depending on the nature of workloads, decide upfront the key resources that are necessary for the particular workload and automate the setting up of such resources. For example, decide what cluster management system to use, whether Mesos [25], YARN [53] or the standalone cluster manager and automate the setting up of such resource along with the other tasks that are necessary for workload execution.
- Continuously monitor the Spark UI for tasks progress, to determine slow nodes and thus tasks that might be running slow. The Spark UI also provides other useful statistical information that could help to identify bottlenecks in the workload execution in order to mitigate or remove the impacts of such bottlenecks and thus speedup application processing.
- Actively observe resource usage metrics while executing workloads to understand utilization, identifying bottlenecks and optimize workloads for optimal resource usage with different operating system tools such as *htop*,[2] *top*, *netstat*, and *iostat*. Also, monitor cluster resources with management systems such as ganglia.[3]
- Ensure that Spark History server is included in the automated bootstrapped tasks and running while executing workloads. The History server ensures that completed workloads can be viewed to under-

[1]https://www.ansible.com/ (Accessed: April 24, 2018)
[2]https://hisham.hm/htop/ Accessed: April 24, 2018
[3]http://ganglia.sourceforge.net/ (Accessed: April 24, 2018)

stand their execution. For the History server to work as expected, the Spark UI property directory. *spark.eventLog.dir*[4] must be specified and the event log property *spark.eventLog.enabled* enabled.

- To reduce the verbosity of logs generated or written to file, configure Spark with the desired logging level such as WARN, ERROR, INFO, FATAL, and DEBUG.
- Use the right partition size for working dataset. This might require a heuristic approach until the best partition size is obtained. The recommended rule of thumb is that tasks should have a minimum completion time of 100 milliseconds as having large number of tasks would overwhelm the task scheduler [55].
- Explore different executors, cores and memory configuration settings until an optimal result is achieved. It is safe to start with two executors per node with tasks equal to 2 or 3 times the number of cores in the cluster and then scale accordingly until a good performance execution speed is achieved.
- Use storage levels based in the input data size and cluster resources. Sometimes depending on the nature of the workloads, different caching policies might be employed especially if there are inherent intermediate RDDs in the pipeline to determine the right policy with respect to workload performance. For example, it might be more performance efficient to use a particular storage level (whether MEMORY or a combination of MEMORY_AND_DISK with replication factors as necessary) for intermediate RDDs rather than caching the input working set.

5.3 Future Work

As this study is a first attempt to understand the influence of Spark configuration settings on image processing related applications, especially in a heterogeneous cluster environment, it could therefore serve as a basis for future researchers to be able to grasp the workings of the Spark engine fast in order to know the key parameters from the myriads of configuration parameters that Spark engineers provide and how best to optimize applications/jobs. Here are a few thoughts on areas where research work could be concentrated on the near future:

- Focus should be given to developing interesting applications that are quite representative in that they cover a wide range of Spark's high level transformations and actions (applications that contains different communication patterns such as collect, aggregate and shuffle patterns). Research based on these kinds of applications could help provide meaningful insights regarding what constitutes bottlenecks in the Spark's system.

[4]https://spark.apache.org/docs/latest/configuration.html#sparkui (Accessed: April 24, 2018)

- Studies should investigate the influence of more configuration parameters in a GPU-enabled Spark cluster as well as using container technologies with automation deployment frameworks such as Kubernetes[5] in a cluster of commodity machines. This is because these frameworks have low overhead and small memory consumption as opposed to the virtual machine implementation used for this study
- Works related to investigation of how Spark performs with shuffle-heavy tasks and how the engine could be optimized to reduce the effect of shuffle I/O on job completion time can also be considered.
- Studies should also consider investigating the reason for the high variability in the use of cluster resources especially CPU and memory in all the stages of the *flowerCounter* appilcation pipeline.
- Future works should be done with large number of cluster nodes to further investigate the horizontal scaling power of Spark with long-running jobs.
- Further, studies regarding the *flowerCounter* application should be designed to process daily images as series of independent Spark jobs to reflect the inherent logic of the application.
- Another interesting focus could be exploring the influence of different JVM configuration settings (with respect to garbage collection) on Spark workloads as the engine runs on JVM. This could also provide some insights into which JVM configuration settings is most suitable for different Spark workloads.
- Finally, apart from the variables investigated in this study, the effect of speculative execution in heterogeneous cluster environment still requires more investigation as there are other tuning parameters apart from the *spark.speculation* parameter studied here. These other parameters can be further investigated to fully understand their workings and how best to use them for optimal Spark jobs performance. There is a knowledge gap in this domain.

[5]https://kubernetes.io/ (Accessed: April 18, 2018)

References

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings. VLDB Endow.*, 8(12):1792–1803, August 2015.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, pages 281–296, San Jose, CA, April 2010.

[3] UC Berkeley AMPLab. The berkeley data analytics stack (bdas) software. `https://amplab.cs.berkeley.edu/software/`, April 2018.

[4] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective straggler mitigation: Attack of the clones. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 185–198, Lombard, IL, August 2013.

[5] Databricks Apache Software Foundation, UC Berkeley AMPLab. Configuration - spark 2.2.0 documentation. `https://spark.apache.org/docs/latest/configuration.html`, December 2016.

[6] Databricks Apache Software Foundation, UC Berkeley AMPLab. Spark programming guide - spark 2.2.0 documentation. `https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html#shuffle-operations`, December 2016.

[7] Databricks Apache Software Foundation, UC Berkeley AMPLab. Tuning - spark 2.2.0 documentation. `https://spark.apache.org/docs/latest/tuning.html#data-locality`, December 2017.

[8] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. Performance characterization of in-memory data analytics on a modern cloud server. In *Proceedings of the 2015 IEEE Fifth International Conference on Big Data and Cloud Computing (BDCloud)*, pages 1–8, Dalian, China, June 2015.

[9] Paul G. Brown. Overview of SciDb: Large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 963–968, Indianapolis, IN, June 2010.

[10] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[11] Quan Chen, Minyi Guo, Qianni Deng, Long Zheng, Song Guo, and Yao Shen. HAT: history-based auto-tuning mapreduce in heterogeneous environments. *The Journal of Supercomputing*, 64(3):1038–1054, June 2013.

[12] Tatsuhiro Chiba and Tamiya Onodera. Workload characterization and optimization of tpc-h queries on apache spark. In *Proceedings of the 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 112–121, Uppsala, Sweden, April 2016.

[13] Cloudera. How-to: Tune your apache spark jobs (part 2). `http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/`, 2015.

[14] Hortonworks Community Concentration. Write or append failures in very small clusters, under heavy load or crash testing. https://community.hortonworks.com/articles/16144/write-or-append-failures-in-very-small-clusters-un.html, 2016.

[15] Transaction Processing Performance Council. TPC Benchmark D (Decision Support) Standard Specification, 1995.

[16] Databricks. Voice from CERN: Apache Spark 2.0 Performance Improvements Investigated with Flame Graphs. `https://databricks.com/blog/2016/10/03/voice-from-cern-apache-spark-2-0-performance-improvements-investigated-with-flame-graphs.html`, 2016.

[17] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] Mark E. DeYoung, Mohammed Salman, Himanshu Bedi, David Raymond, and Joseph G. Tront. Spark on the ARC: Big data analytics frameworks on HPC clusters. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, pages 34:1–34:6, New Orleans, LA, July 2017.

[19] Massimo Franceschet. Pagerank: Standing on the shoulders of giants. *Commun. ACM*, 54(6):92–101, June 2011.

[20] Michael Franklin. Making sense of big data with the berkeley data analytics stack. In *Proceedings of the 8th ACM International Conference on Web Search and Data Mining*, pages 1–2, Shanghai, China, February 2015.

[21] Ilya Ganelin, Ema Orhian, Kai Sasaki, and Brennon York. *Spark: Big Data Cluster Computing in Production*. John Wiley & Sons, 2016.

[22] Gergő Gombos, Attila Kiss, and Zoltán Zvara. Performance analysis of a cluster management system with stress cases. *Acta Polytechnica Hungarica*, 13(2):77–95, 2016.

[23] Lei Gu and Huan Li. Memory or Time: Performance Evaluation for Iterative Operation on Hadoop and Spark. In *Proceedings of the 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 721–727, Zhangjiajie Shi, China, November 2013.

[24] Apache Hadoop. Hadoop, 2009.

[25] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 295–308, Boston, MA, March 2011.

[26] IFIXIT. Computer processor characteristics - ifixit. `https://www.ifixit.com/Wiki/Computer_Processor_Characteristics`, December 2017.

[27] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.

[28] S. Kamburugamuve. *Survey of Apache Big Data Stack*. PhD thesis, Ph. D. Qualifying Exam, Dept. Inf. Comput., Indiana Univ., Bloomington, IN, 2013.

[29] H. Karau and R. Warren. *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly Media, 2017. ISBN 9781491943175. URL `https://books.google.ca/books?id=90glDwAAQBAJ`.

[30] A. Katal, M. Wazid, and R.H. Goudar. Big Data: Issues, challenges, tools and good practices. In *Proceedings of the 2013 6th International Conference on Contemporary Computing (IC3)*, pages 404–409, Noida, India, August 2013.

[31] Joohyun Kyong, Jinwoo Jeon, and Sung-Soo Lim. Improving scalability of apache spark-based scale-up server through docker container-based partitioning. In *Proceedings of the 6th International Conference on Software and Computer Applications*, pages 176–180, Bangkok, Thailand, February 2017.

[32] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[33] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 6:1–6:15, Seattle, WA, November 2014.

[34] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: a spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing*, 20(3):2575–2589, September 2017.

[35] Fredrik Lundh. Python Imaging library (pil), 2012. URL `http://www.pythonware.com/products/pil/`.

[36] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016.

[37] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[38] N. Nguyen, M. M. H. Khan, Y. Albayram, and K. Wang. Understanding the influence of configuration settings: An execution model-driven framework for apache spark platform. In *Proceedings of the 2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 802–807, Honolulu, CA, June 2017.

[39] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at linkedin. *Proceedings. VLDB Endow.*, 10(12):1634–1645, August 2017.

[40] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307, Oakland, CA, May 2015.

[41] Kay Ousterhout. Scripts to analyze spark's performance. `https://github.com/kayousterhout/trace-analysis`, June 2014.

[42] Kay Ousterhout, Christopher Canel, Max Wolffe, Sylvia Ratnasamy, and Scott Shenker. Performance Clarity As a First-class Design Principle. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 1–6, Whistler, Canada, May 2017.

[43] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. Real-time Computer Vision with OpenCV. *Communications of the ACM*, 55(6):61–69, June 2012.

[44] Qubole. How to: Spark tuning. `https://qubole.zendesk.com/hc/en-us/articles/208693126-How-To-Spark-Tuning`, March 2017.

[45] Alexander Rasmussen, Michael Conley, George Porter, Rishi Kapoor, Amin Vahdat, *et al.* Themis: An I/O-efficient MapReduce. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, pages 13:1–13:14, San Jose, CA, October 2012.

[46] S. Salehian and Y. Yan. Comparison of spark resource managers and distributed file systems. In *Proceedings of the 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, pages 567–572, Atlanta, GA, October 2016.

[47] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Washington, DC, May 2010.

[48] James Sofra. Balancing spark - bin packing to solve data skew. `http://silverpond.com.au/2016/10/06/balancing-spark.html`, October 2016.

[49] Apache Spark. Components of a Spark Application. `https://spark.apache.org/docs/latest/cluster-overview.html#components`, 2018.

[50] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 147–156, Snowbird, Utah, June 2014.

[51] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. On the [ir] relevance of network performance for data processing. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, pages 126–131, Denver, CO, June 2016.

[52] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. SciKit-image: image processing in Python. *PeerJ*, 2:e453, 2014.

[53] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 5:1–5:16, Santa Clara, CA, October 2013.

[54] J. Veiga, R. R. Expósito, X. C. Pardo, G. L. Taboada, and J. Tourifio. Performance evaluation of big data frameworks for large-scale data analytics. In *Proceedings of the 2016 IEEE International Conference on Big Data (Big Data)*, pages 424–431, Washington, DC, December 2016.

[55] Roberto Agostino Vitillo. Spark best practices. `https://robertovitillo.com/2015/06/30/spark-best-practices/`, 2015-06.

[56] Mehul Nalin Vora. Hadoop-hbase for large-scale data. In *Proceedings of the 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605, Harbin, China, December 2011.

[57] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13(2):22–30, March 2011.

[58] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with Apache Kafka. *Proceedings. VLDB Endow.*, 8(12):1654–1655, August 2015.

[59] K. Wang and M.M.H. Khan. Performance Prediction for Apache Spark Platform. In *Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 166–173, Washington, DC, August 2015.

[60] Hongbin Yang, Xianyang Liu, Shenbo Chen, Zhou Lei, Hongguang Du, and Caixin Zhu. Improving spark performance with MPTE in heterogeneous environments. In *Proceedings of the 2016 International Conference on Audio, Language and Image Processing (ICALIP)*, pages 28–33, Dubai, UAE, July 2016.

[61] K. Ye and Y. Ji. Performance tuning and modeling for big data applications in docker containers. In *Proceedings of the 2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–6, Shenzhen, China, August 2017.

[62] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10, Boston, MA, June 2010.

[63] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012.

[64] Xuechen Zhang, Ujjwal Khanal, Xinghui Zhao, and Stephen Ficklin. Understanding software platforms for in-memory scientific data analysis: A case study of the spark system. In *Proceedings of the 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1135–1144, Wuhan, China, December 2016.