UiO : Department of Informatics
University of Oslo

Lean MapReduce: A B-tree Inspired MapReduce Framework

Dynamic provisioning of compute resources to adhoc MapReduce clusters based on workload size

Arinze George Akubue

Master's Thesis Spring 2016

Lean MapReduce: A B-tree Inspired MapReduce Framework

Arinze George Akubue

June 6, 2016

Acknowledgement

First and foremost, I would like to express my gratitude to my thesis supervisor, Kyrre Begnum, for his expertise, guidance and inspiration, which helped steer this project down the right path. His encouragement and infectious enthusiasm kept me going throughout the duration of this project.

I would also like to thank Anis Yazidi and Hårek Haugerud for numerous insightful comments which helped improve the quality of this project.

Finally, I would like to extend my gratitude to my family for their immeasurable support, solicitude, and understanding throughout the master's program. I couldn't have done this without you.

Abstract

There is a deluge of unstructured data flowing out from numerous sources, including the devices which make up the Internet-of-Things. This data flow is characterised by sheer volume, variety and velocity, and is expected to double every two years. Organizations perceive hidden value in unstructured data, but are usually constrained by budget and access to the right kind of technology in their effort to extract value. MapReduce has been adopted widely in the big data community for large scale processing of workloads. Current implementations of MapReduce run on persistent compute clusters which feature an underlying distributed file system. The clusters typically process numerous jobs during their lifetime. During periods of low or no activity, the resources are unutilized.

This thesis investigates how resources can be optimally and efficiently utilized through the use of adhocly provisioned MapReduce clusters, which are grown into place for each job based on workload dimensions while meeting results deadlines. In order to achieve this, two different designs are developed based on two distinct adaptations of the B-Tree abstract data structure: a flat tree structure, which grows horizontally; and a chain structure with hanging leaves, which grows vertically. The project results show that resources are optimally and efficiently utilized, with each design implementation demonstrating individual advantages and disadvantages, for different workload dimensions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data exists around us in several forms. The last decade has witnessed a growing torrent of data flowing out from numerous sources including the collection of smart devices that make-up the Internet-of-things, wireless sensor networks, and other sources. The increase in digitization of data catalysed by advancement in digitization and storage technology [1], contributes to this development. According to a report published by the International Data Corporation (IDC), the amount of data created and copied in 2011 was approximately 1.8 zetabytes (ZB) and increased ninety-fold over the subsequent five years. It is anticipated that this figure will double every other two years [2].

Data in the modern age is perceived as being characterized by sheer volume, variety, and velocity [3], demanding techniques beyond the traditional for their warehousing and value treatment; this state of manifestation is termed Big Data. Big data is deemed to be valuable asset by business organizations (small, medium, and large), governmental establishments, and research institutes, as it can potentially deliver fresh insights in any domain of interest; however they face challenges which have to do with financial constraints, and the ability to optimize utility and efficiency of in-house compute and memory capacity to limit waste in processing and analysing their big data.

The drive to extract such value has led to the overlap of traditional High Performance Computing (HPC) concepts with the big data domain. HPC refers to the use of mostly parallel processing on a single or constellation of supercomputers to solve compute intensive problems. The use of HPC systems for big data data analysis is common with experiments in large-scale research projects such as the Large Hadron Collider dark matter experiments which utilizes the LHC Computing grid - a federation of about 500 supercomputers - [4] to sift through about 30 petabytes of data annually [5].

The acquisition and maintenance of supercomputers require a steep budgetary commitment in the order of several millions of dollars [6] making them nonviable options for the computing needs of most business enterprises, especially small and medium scale enterprises, and big data science researchers in developing countries operating on a tight budget. It is also not an appealing option for large organizations that can afford them but would not utilize them to such a degree as to justify the

investment. The use of cloud computing, as a way to harness the capacity of geographically dislocated large data centers, provides an alternative approach to computing for big data analysis. Most data centers host a network of hundreds or thousands of servers built out of moderately priced hardware components. As standalone systems these servers are unremarkable but when each is assigned a portion of a larger problem, a cluster of them is capable of performing highly demanding computations which traditionally were the preserve of supercomputers.

As the resource needs of big data shops grow, most large organizations such as Yahoo, Google and facebook, as well small and medium scale organizations have turned to the MapReduce programming paradigm [7]. A popular implementation of the framework is the open-source Apache Hadoop framework, which is maintained by the Apache Software Foundation. MapReduce features abstractions which a programmer can capitalise on to simplify writing of distributed applications [7]. The framework provides a blueprint for building systems which feature mechanisms for partitioning and distributing data, and ensuring fault-tolerance. Such systems are usually long-lived, meant to receive and process many big data jobs during their life time; for organizations with existing data centers, this calls for partitioning of their data centers and dedicating a portion to their MapReduce installation for big data analysis needs. This arrangement has a side effect of resource under-utilization during periods of low activity.

Similar systems may be implemented on top of a cluster of cloud-based servers, where users specify the number and type of resource sets to be allocated. A resource set is a collection of virtualized resources rented as a single unit - such as virtual machines rented on Amazon web services. Typically, the target of the framework is batch processing of data on a Local Area Network-based cluster (LAN) which relies on an underlying distributed file system to organize the data. If a decision is made to migrate a MapReduce cluster or create a new cluster on another platform, specialist knowledge and a good amount of time is needed for the transition, or installation and configuration. This has some ramifications for the users: there is the likelihood of resource over-provisioning of resources leading to idle resources, and the user could incur a sizable cost due to scope and longevity intentions for the cluster.

Some attempts have been made at improving the resource utilization of MapReduce (see 2.11), particularly Hadoop clusters, but such attempts focus on mechanisms for boosting resource utilization and results delivery time in persistent clusters, and do not address the long term resource implications of such long-lived clusters.

In light of the foregoing, the aim of this thesis is to investigate Lean MapReduce, a novel architecture for MapReduce which enables optimized and efficient resource utilization through adhoc and elastic provisioning of system resources. It embodies the potential benefits that may accrue to a MapReduce process if the participant servers run a uni-kernel operating system to host the collection of applications which makeup the MapReduce system. By so doing, system resources will be more efficiently utilized, creating the possibility of improving virtualization density on a

single physical server. By virtue of being a MapReduce architecture, it is designed to have mechanisms for job completion within deadlines, and system reliability guarantees.

The desired resources may be allocated across multiple clouds. By being able to distribute workloads over multiple platforms, Lean MapReduce gives big data jobs a nimble foundation where resources may be acquired from a separate cloud platform when there is a shortage on the current platform. It may also enable them to partially run to completion or to fail-over swiftly to a disparate computing platform when the current platform is in an inoperative state. Such a framework also allows users to take advantage of different price options offered by cloud service providers.

The rest of this paper describes Lean MapReduce and how it can deliver the benefits, among others, mentioned above.

1.1 Problem Statement

This paper explores a novel MapReduce deployment strategy for big data. The intent behind the strategy is to manifest a MapReduce system which consumes just the right amount of resources needed for a single job window, delivers results in a timely manner, and leaves behind no long-term resource footprint. Typical MapReduce deployments reliably process data, ideally within deadline windows. They require a separate, dedicated network of servers in a data center or a a cluster of cloud-based servers with a long lifetime to achieve this goal, hence they generate resource waste during idle periods. Further more, they are bound to a single local area network due to management constraints and component characteristics. This paper therefore addresses the following challenge:

How to design an adhoc MapReduce architecture which optimally and efficiently utilizes system resources to reliably process big data workloads and deliver results in a timely manner?

In the phrase *Optimally and efficiently utilize*, *optimally* refers to the utilization of idle system resources for processing a workload. *Efficiently* refers to the resource utilization by the member tasks of a big data processing job, in such a way they impose the smallest possible resource footprint, while supporting an acceptable level of performance in processing some workload.

By *Adhoc*, the MapReduce cluster is created for the specific purpose of analysing data within on job window, where a job window is the time it takes to completely perform an analysis task.

Workload refers to the mass of unstructured data which has been presented for processing by the MapReduce system.

The phrase, *timely manner* means that the outcome of a big data processing job should be delivered within a time frame within which it is considered relevant.

Chapter 2

Background

This chapter gives an overview of related work as well as some concepts that are relevant to the presentation of this thesis. A discussion of previous work is also carried out to reveal the state-of-the-art solutions and approaches used in this domain and the underlying concepts which inspired their emergence. The thesis is well placed within the margins of High Performance Data Computing (see 2.4 for more on HPDA) , hence an understanding of the ideas whose marriage became this new computing frontier is a logical first step.

2.1 High Performance Computing

The term HPC refers to the application of parallel processing in executing advanced programs that need significant amount of compute resources and power[8]. The term HPC has evolved in step with the evolution of the modern computer [9]: from being a budget perplexing venture involving a single vendor production built according to specification, to a conglomeration of systems with negligible piecewise budgetary constraint.

The major goals of HPC are a relatively quick execution time and the ability to scale gracefully with the growth of the problem scope and the attendant complexity [8]. Processor cycles can be delivered to a HPC task in one of the following recognized formations:

- Dedicated Supercomputer: A computer with exceedingly more computational capacity compared to general purpose computers. In the past, only systems built out of specialized component could be relied on to deliver the amount of processor cycles needed for resource-intensive problems.
- Grid Computing:

  Grid computing stands a level above clusters and cloud computing in the distribution-based taxonomy of HPC systems: a grid is a pooling of compute resources from several interconnected sites – physical datacenters or cloud-based – which are geographically dislocated.

- Commodity HPC Cluster:

this mode features the use of a cluster of standard servers, which can be purchased off the shelf and are omnipresent in datacenters, to provide for the resource needs of HPC problems. The comparatively low cost of commodity hardware makes this approach a cost-effective option for HPC. The size of the cluster is determined by the perceived size of the processing task, and could go from hundreds of servers to tens-of-thousands of servers.

- HPC Cloud Computing:

  The rise and popularity of cloud computing has seen an offload of HPC problems to the cloud, tapping into the cycles-as-a-service model of computing – a service layer built on top of the broader infrastructure-as-a-service model. It is a manifestation of commodity HPC clusters without the geographical limitation and awareness of its physical datacenter-bound counterpart, and with cheaper cost of ownership. The nature of provisioning compute capacity at a cloud platform implies that a user can remotely access resources a pay-as-you-go arrangement, and can scale the size of their cloud-based HPC cluster as the need arises in an economically friendly way and without added real-estate or hardware footprint [10].

2.1.1 Features of a HPC Cluster

A HPC cluster typically has a master node which acts as a connection broker and schedules user jobs. It is also responsible for balancing load across a cluster of worker nodes. The worker nodes typically host identical programs and work in a confederated manner to execute a scheduled job. Communication between the master node and worker nodes, or between worker nodes, may happen over a low latency interconnect (for a private network) using Gigabit - or higher order Gigabit - Ethernet connections [10], and more recently Infiniband connections.

During the lifetime of a cluster, three major types of traffic will traverse its connecting fabric:

- Administration Traffic: which keeps track of the state of jobs and availability of worker nodes, as well as other management, deployment and feedback notification.
- Computation Traffic: is traffic which flows between nodes - ideally worker nodes - especially in a scenario where partitions of the cluster have been earmarked for blocking or sequential execution of programs.
- File System Traffic: which may be associated with Message Passing Interface (MPI) application communication which occur between parallelized programs executing on member nodes of a HPC cluster or an interaction with a Network File System server [10].

Depending on the needs of a computation task, a HPC cluster can be classified as either a *capability* cluster or a *capacity* cluster. The former points to compute jobs which utilize every node in a cluster, while the latter consumes a portion of the clusters compute resources up to a certain specification defined for the job [10].

High Performance clusters are usually deployed when a compute job cannot fit on a single server, or when the job in view is of time-critical value. Most cluster deployments co-opt MPI message-passing system for intra-node communication during the execution of a job. The frequency of mutual communication between nodes leads to a spike in network traffic - a challenge that calls for the use of a high performance network, as alluded to in the opening paragraph of this section. By virtue of tasking all processors present in a cluster, parallelized programs gain speed-ups several orders of magnitude higher than what is obtainable if a single server is relied upon [10].

Data is kept and accessed in one of two ways: centralized and distributed. Both approaches have their advantages, some factors which influence design choice on this front include the size of the cluster, the network link capacity, scalability, and budget for procurement of software licence and additional hardware [10].

Of particular importance is the underlying file system which facilitates data storage and access. File systems used in HPC contexts have developed from the once popular standard Network File System (NFS) - which suffered parallel access bottlenecks - to a diversity of parallel file systems which feature in most HPC systems of today. A parallel file system is designed to partition and spread out data across multiple nodes in a cluster instead of concentrating storage on a single server. This facilitates faster data reads and writes due to the locality of data needed by an executing program [10]. Some popular file systems in this space include Lustre, developed by Sun Microsystems, and Hadoop Distributed File System (HDFS).

2.2 Big Data

The term Big Data refers to information or data that cannot be processed with the aid of traditional processes and tools [11]. It officially headlines the information explosion phenomenon which was observed and tracked for several years in an attempt "to quantify the growth rate in the volume of data" [12]. Three primary developments which have contributed significantly to the surge in big data include: the increasing degree of instrumentation to more accurately sense and effectively monitor our environments; advances in communication technology, accelerated by perpetual inter-connectivity of devices owned by an ever growing population of people; and the reduction in cost and trend to nano-scale of integrated circuits, which permits devices around us to be imbued with some artificial intelligence. The interconnection of devices, formally known as machine-to-machine (M2M), is pronouncedly responsible for "double-digit year-on-year (YoY) data growth rates" [11] ahead of the other mentioned causes.

When considered superficially, the term Big Data carries the connotation of "plenty of data," which implies that pre-existing data is relatively small hence the challenge we currently face is just overwhelming volume. However, volume is just one axis of the full picture. A better understanding can be derived by looking at the big data phenomenon from three defining perspectives (also referred to as challenges): volume, variety and velocity (the 3 Vs).

- Volume

  The rate at which data is generated and the amount stored keeps growing exponentially from year to year. To put things in perspective, in the year 2000, 800,000 petabytes (PB), or 800 exabytes (EB), was the estimated aggregate of all data kept in all known storage devices worldwide. Fast-forward to 2014, Google was storing 15,000 PB (15 EB) per day, with the US National Security Agency (NSA), Baidu and Facebook not too far behind – 10 EB, 2 EB and 300 PB respectively [13]. These are just a small segment of an extensive list of enterprises that generate data on such a massive scale. IBM forecasts that the total amount of data held in storage globally will reach 35 zettabytes (ZB) by the year 2020 [11].

  As we strive to for improvements in systems which complement our basic functions through ergonomics and other scientific disciplines, more and more data is stored about everything we are able to sense. All basic humanistic events which involve some interaction with technology materialises data which are stored in some form. For example, car manufacturers – including BMW – now employ Big Data in improving their car models by capturing all kinds of data during repeated test of their prototypes; this is in addition to Telematics information that they gather continually when the vehicle is the hands of the eventual owner [14].

  One major challenge that enterprises have to deal with today, is the decline in the percentage of data they are able to process and analyze with the rapid growth in data that is available to them; an issue that cannot be dealt with if the right technologies are not in place [11].

- Variety:

  As our environments becomes more sensorized, and our social lives mesh more tightly with technology, data has become more complex as it now includes not only structured data, but also semi-structured and unstructured data from a myriad sources including system logs, social media platforms, web search indexes, and so forth [11]. In simpler terms, variety is an envelop term for all data types and is a definitive theme for the shift in the ingestion of data from structured to unstructured, enriching the capabilities of organizations to build an edge of competitive advantage whilst challenging pre-existing traditional analytical platforms [11].

  Structured data refers to data that conforms to predefined data models and can be processed in such a way that they can be used in achieving precise results. In practice, such data are logically organized into tables of a database as rows and columns, where well-defined Structured Query Language (SQL) based queries can be applied to them. At the other extreme of the structure spectrum are unstructured data: they are typically text-heavy and follow no pre-defined data model, therefore have no place in a traditional row-column database. Unstructured data are usually found to give structured data meaning: for example, while email files are constructed following a

predefined format which allows them to be stored in a database, the message they contain are free-form text which follow no predefined data model [15].

A significant portion of enterprise data is unstructured [16]. Unstructured data are usually found to give structured data meaning: for example, while email files are constructed following a predefined format which allows them to be stored in a database, the message they contain are free-form text which follow no predefined data model [15]. Other forms are audio, video, JSON text, and so on.

- Velocity

  By convention, data velocity is understood to be the rate at which data is received, stored and retrieved. This is typically the case when static data is at play and when there is the assumption that captured data has value which does not go stale. If the nature of data is viewed against a more dynamic backdrop where the shelf-life is short and hence value must be extracted as soon as the data is sensed – an approach adopted in Nowcasting [17] - then velocity takes on additional meaning which points to the speed of data in motion. This state of data is due to the proliferation of RFID sensors and other information streams, which sustain a steady flow of data "at a pace that has made it impossible for traditional systems to handle." [11]

2.2.1 Why is Big Data Important?

In the enterprise world, Big Data is fast emerging as a key factor in the decisions which shape the direction of businesses. Organizations that have invested in Big Data analysis infrastructure are able to extract insight about aspects of their operations, which were hitherto impossible to obtain, by running queries against the mass of data they have in storage or by tapping into a rapidly flowing stream of data polled by their monitoring and reporting assets.

With Big Data now the umbrella term for structured and unstructured data, it stands to believe that there is a lot of potential that is untapped by organizations which are not big data aware or are partially so. According to the International Data Corporation (IDC) (das2013big) more than 80\% of all data which could yield meaningful business value is unstructured data. Some companies now leverage their ability to process and analyze big data to perform controlled experiments to deliver more convincing and conclusive evidence for their decision-making process [17].

The way organizations treat their data may depend on the metrics they associate with big data. For example, while some consider 10 terabytes(TB) of data to be big data [18] others may hold a different notion of that threshold (less than or greater than 10TB). Irrespective of the current sense of magnitude, what is considered big today won't remain as such in future as existing data sources are optimally tapped and new sources brought into the mix.

2.3 Big Data Analytics (BDA)

With new technologies generating more data than ever before - catalysed by a rapid increase in the three Vs - organizations have had to adapt by putting in place mechanisms for receiving and storing such data. However, data kept in storage is a liability if not tapped for information that give fresh and timely insights. Big Data Analytics is the practice of analyzing the totality of the mass of stored or streamed data to discover new patterns and correlations which could give rise to fresh meaning in the context of the motivating interest. It helps inform on what has taken place, what is taking place and what is likely to happen, and therefore aid in the design of proactive measures to achieve optimal results [19]. In business terms, big data analytics transforms data from liability to high value asset [20].

Big Data Analytics is considered to cater to the outer limits of the scope of traditional Business Intelligence systems (BI): BI systems rely on structured queries - typically SQL -to extract information based on objects stored in relational databases. It must be noted that Big Data Analytics and traditional BI are not mutually exclusive but are complementary concepts. From this perspective it can be seen that with organizations having to curate and present their databases for audit in a manner consistent with data governance requirements [21], and the demands of the management board for a "Single Version of Truth," there is an over-arching need to meaningfully organize unstructured data which are considered essential business asset. For this purpose, Big Data Analytics can be used to deliver an intermediate state of data in classical Extract, Transform and Load approach, and then kept in a conventional database. For big data, Extract stands for the ability to efficiently extract data with minimal impact on the system; Transform refers to the ability to transmute batch or real-time data sets into a format which is consumable by a target system; while Load represents the ability to keep the data persistently in a data store where it can be access with well-defined protocols [21].

2.4 From Big Analytics to High Performance Data Analytics

It is common to observe the terms Big Data Analytics and High Performance Computing used interchangeably in discourse, as they are deemed to refer to intensive processing of data in an overlapping manner [22]; however they differ subtly in the design goals that they focus on. HPC employs the use of a collection of well synchronized processors for parallel processing of resource intensive workloads, therefore it is mostly concerned with hardware and interconnect details; while Big Data Analytics is a practice whereby fast results are delivered for analytical jobs, and pays more attention to the performance of the software layer. Most Big Data Analytics systems are built following a distributed programming model.

To formally highlight the intersection between both concepts, the term High Performance Data Analytics was coined by the IDC [23]. The term is yet to gain mainstream acceptance but is growing in popularity partly because the

fusion of HPC and BDA is self evident in the compound term. Their functional fusion enables analyst to leverage the technical computing capabilities that they independently offer in solving domain challenges using complex algorithms.

HPDA also marks the crossover point of HPC into the commercial domain. HPC is historically associated with research about computing challenges in the areas of Science and Technology [24]. Its evolution is propelled by the proliferation in open source frameworks for big data analytics which can be adapted for deployment public and private cloud computing services, and on commodity clusters available in physical datacenters.

2.5 HPDA Tools

The following subsections introduce a few definitive concepts and tools which facilitate the deployment of HPDA environments:

2.5.1 MapReduce

MapReduce is a powerful programming framework designed for the distributed processing of big data sets [25]. It was conceived at Google in reaction to their big data challenges of parallel computation, distribution of data and failure recovery [25].

The framework provides programmers a means of writing applications which leverage a cluster of resources. MapReduce borrows from the concepts of map and reduce primitives present in many functional programming languages [25]. it facilitates processing by taking charge of a cluster of servers; executing tasks in parallel; monitoring and managing communications and object transfers between the components of the MapReduce system, and enabling fault tolerance.

A MapReduce computation is typically expressed as two functions: a Map function and a Reduce function. The goal is to process input in the form of key/value pairs and output a summarize set of key/value pairs. In an logical sense, Map function process input pairs in parallel, generates a set of intermediate key/value pairs, groups values with identical keys into distinct lists. This is summarized notationally as:

$$\text{Map}(A1, V1) => \text{list}(A2, V2)$$

The Reduce function is a merge-like operation. Ideally, a Reduce function works on a single intermediate key and its associated values, and coalesces the values into a possibly smaller value. At most one value is produced per each for each execution of a Reduce function [25]. This can be notationally be expressed as:

$$\text{Reduce}(A2, \text{list}(V2)) => \text{list}(V3))$$

The following pseudocode illustrates an instance of MapReduce used for tallying the occurrence of words in a body of text:

Map and Reduce Phases:

```
class Mapper
        method Reduce (doc_name name, doc_content content)
                for all word w in content: do
                        Emit(w, count 1)
class Reducer
        method Reduce (word w, list_of_count[a1,a2...an])
                total = 0
                for all count k in list_of_counts[a1,a2...an]: do
                        total = total + i
                        // where i = 1,2,...n
                Emit(word w, count total)
```

The shuffle phase follows an "all-or-nothing" concept; it does not begin until all map tasks are successfully completed. When the shuffle task receive intermediate key/value lists from the map phase, it merges and sorts them based on their intermediate keys. The shuffle task has the added responsibility of delivering key/value list to the designated reduce tasks [7]. It should be noted that the shuffling process is a local action taken by each reduce function for its input data.

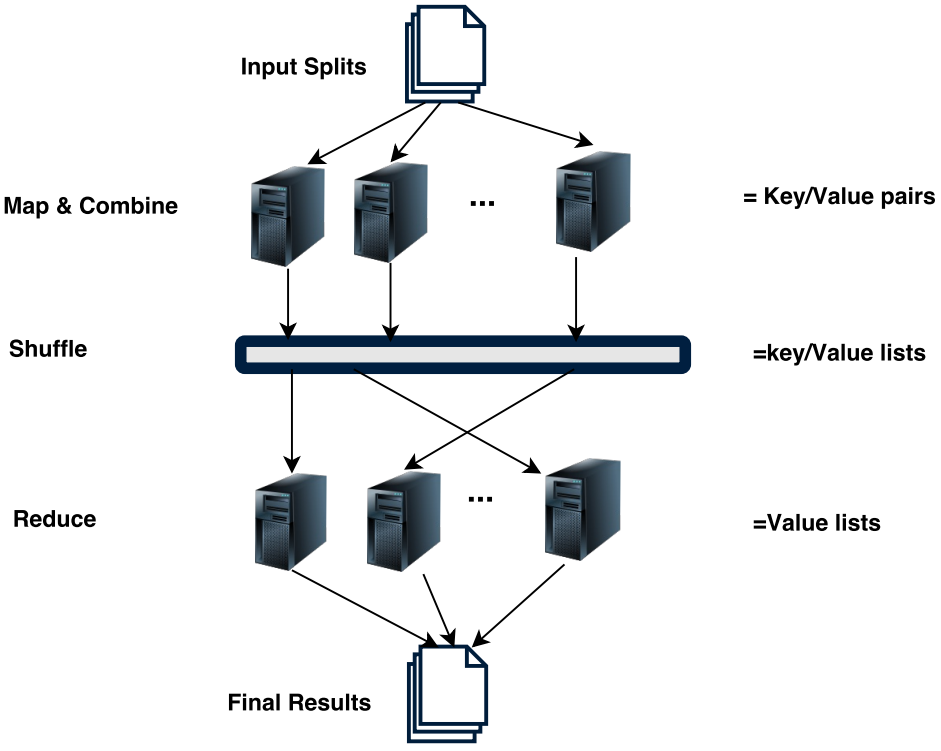

Figure 2.1: Hadoop MapReduce has two distinct phases: map and reduce, and sub-phases called combine and shuffle

To reduce the amount of intermediate data generated by mappers hence the amount of data sent across the network between mappers and reducers, the framework offers an optional feature called the combine function. In essence, the combine function does local aggregation of values with identical keys consequently emitting the keys and the aggregate values, reducing the amount to be shuffled and potentially dispatched across the network to a reducer [26]. It also has the implicit

side-effect of reducing the amount of iterations in the summarizing task of the Reducer. The following pseudo-code illustrates this:

Map function with Combine construct:

```
1  class Mapper
2      method Map (doc_name name, doc_content content)
3          A = new AssociativeArray
4
5          //combiner begin .....
6
7          for all word w in content: do
8              A[w] = A[w] +1
9          //combiner end .....
10
11         for all word w in A: do
12             Emit(w, count A[w])
```

Nodes in a MapReduce cluster are classified into 2 roles: master and worker. Master is responsible for creating and tracking the status of a defined number of tasks which are assigned to one or more workers. Each worker handles a Map or a Reduce task exclusively and reports back to the master upon task completion [26]. To optimize efficiency of the system, MapReduce leverages locality of data, by processing data on workers closest to where it is stored in the distributed system - the location is typically is the same server hosting the worker.

The master also features a fault-tolerant mechanism which works hand-in-hand with its job state monitoring apparatus by prompting a response from each worker. The master reacts to a failed response by marking the worker as “idle” and rescheduling their assigned task for re-execution [26].

2.5.2 Apache Hadoop

Hadoop is a an open-source implementation of the MapReduce (detailed in the preceding section) [22]. It was designed for the distributed storage and processing of big data sets on small-to-large scale clusters built out of commodity hardware. Since it’s first release it has been widely embraced as the defacto standard of innovation in big data analytics. There have been several minor releases and two major releases: versions 1.0 and 2.0.

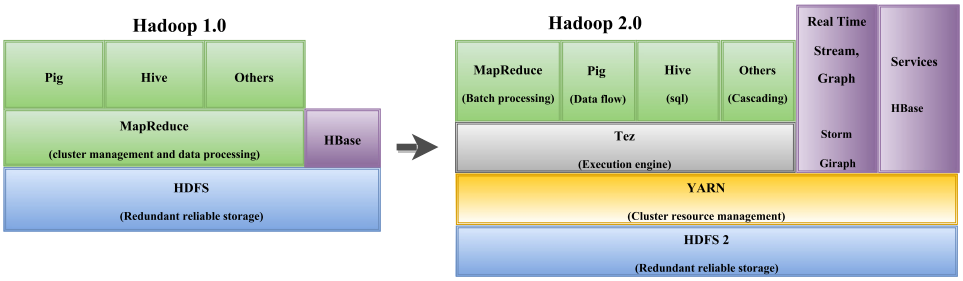

Figure 2.2: Transition From Hadoop 1.0 to 2.0 sees some added features, most notably Apache YARN

Hadoop automatically parallelizes execution across all nodes of cluster and ensures fault-tolerance, taking away the responsible of writing distributed applications away from the programmer and letting them focus instead on the job of building robust applications which leverage the available resources. Hadoop is typically laid out over a cluster of servers belonging to a Local Area Network (LAN) and is used mainly for batch processing of big data.

The hadoop framework employs a master/slave approach in running its clusters. In every cluster, there is a master node which oversees the distribution of work and monitoring of jobs execution, and a set of worker nodes responsible for hosting computation tasks assigned by the master. The nodes are characterised, and distinguished, by the presence of JobTracker and TaskTracker processes; the former confers the master role on the host node, while the latter defines the slave role. The JobTracker performs the job of splitting data as well as preparing a task-queue based on the number of data splits [27]. Tasks are then distributed to the TaskTrackers for processing. In a cluster, there are always as many active map tasks as there are data splits, while the number of reduce tasks is arbitrarily set by the user. Each TaskTracker will typically be assigned a number of tasks up to a fixed limit.

Resources in Hadoop are typically divided into slots, where a each slot is dedicated to either a map or reduce task. Each TaskTracker has a fixed number of slots and this number enforces a constraint on the maximum number of tasks which can be executed concurrently at a node [27].

The core modules in the Hadoop framework – Hadoop Distributed File System (HDFS) and MapReduce 2.0 – were inspired by Google's publication about the Google Distributed File System [28] and their proprietary MapReduce design [25]. Other components of the Hadoop base framework are: Hadoop Yet Another Resource Negotiator (YARN) and Hadoop common. Beyond these, the term Apache Hadoop has come to refer to a pool of additional software packages which can be piled on the Hadoop base stack, and these include: Apache Pig, Apache Flume, Apache Sqoop, Apache Oozie, Apache Zookeeper, Apache Spark, Apache Hbase, Apache Storm and Apache Phoenix. HDFS, Hadoop MapReduce 2.0 and Apache Yarn are presented here in brief - Hadoop MapReduce 2.0 is alluded to as part of the presentation of YARN [29].

2.5.2.1 Hadoop Distributed File System

HDFS is one of the key components of Hadoop. It supports distributed processing of data by managing the spread of data across a cluster of commodity hardware or servers. It enables the storage and access of different formats of data in non-relational databases with a default block size of 64 bytes. The implication of the relatively large block size is reduction in disk seeks [26]. It is robust enough to support up to 200 PB of data and a cluster of 4500 servers [30]. The file system enables even partitioning of data (also known as input splits) among nodes of the cluster and output of MapReduce job results to a central location for easy access to the user [7]. In addition, it provides fault-tolerance by keeping redundant copies of datasets on multiple nodes (at least 3 copies) [26].

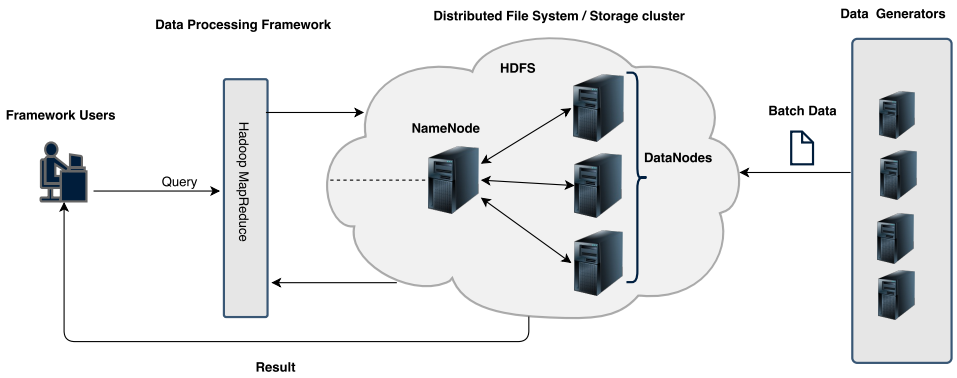

Figure 2.3: HDFS partitions data among nodes and defines NameNodes and DataNodes

A HDFS setup typically defines two kinds of nodes in the Hadoop cluster: NameNodes and DataNodes. The name node provides is cluster-aware and coordinates the distribution of data blocks by managing the file namespace, file system tree and metadata [26]. HDFS design makes provision for a single NameNode per cluster which is solely responsible for file access and poses the risk of being a single point of failure. The DataNodes act as stores for files scheduled to them by the NameNode. In a typical Hadoop implementation the DataNodes know their NameNode but are oblivious of the existence of neighbour DataNodes.

2.5.2.2 Apache YARN

Apache YARN, introduced to the Hadoop framework in the 2.0 release, is a smart tool for facilitating a data computation framework by splitting the functionality of data processing and cluster resource management into two separate daemons. In Hadoop 1.0, the MapReduce programming model and cluster resource management were tightly coupled; the Job tracker which is a part of the MapReduce framework, was responsible for resource management and coordination of task-trackers (per node local job monitors). Under the new architecture YARN assumed the responsibility of cluster management with no need for a job tracker and task trackers, while the MapReduce module was streamlined more towards achieving its core data processing function by strictly following the concepts of classical MapReduce [31]. With this setup, Hadoop systems are able to accommodate a broader variety of processing approaches and a broader array of applications including those which do no follow a MapReduce model [32]. Figure 2.2 illustrates the place of YARN in the Hadoop architecture.

2.6 Virtualization

Virtualization means to create a virtual version of IT infrastructure including servers, storage, network and operating systems. It refers to the use of a layer of abstraction to separate a service request from the delivery of that service at the underlying layer [33]. Through virtualization, an organization can enable a

computing environment in which multiple independent systems are run with little additional financial investment.

Among the key benefits of virtualization are:

- Server consolidation:

  Applications which require a small amount of processing power, and hitherto occupied dedicated servers, can be consolidated in a single server machine running multiple virtual environments. This eliminates server sprawl in datacenters and boosts server utilization rates.

- Optimization of application development and testing:

  With virtual programmers and testers eliminate delays in obtaining the infrastructural pieces required for various levels of standardized testing; pre-configured systems can be used as a template for creating virtual servers.

- Enterprise security:

  Typically unmanaged computer systems can be secured non-intrusively, by introducing a layer of security policy around virtual machines [34].

A virtualization layer, known as a hypervisor, is a piece of software or low-level program which enables the co-habitation of concurrently running operating systems, on a single host computer [34]. The hypervisor accomplishes this objective by assigning a dedicated system resource management module - a virtual machine monitor (VMM) - to each hosted operating system; the VMM delivers a partition of CPU, memory and I/O devices to their coupled virtual machines [33], which have the illusion of total ownership of all underlying hardware. Examples of hypervisors are Xen, Kernel-based Virtual Machine (KVM) (which converts a Linux Kernel to a hypervisor), vmware's range of virtualization tools, and a host of others.

X86 system virtualization techniques make use of either one of two types of hypervisors:

- Type 1 Hypervisors (Bare Metal Hypervisors):

  The hypervisor is installed directly on the hardware before any other piece of software, and exposes hardware resources to overlying operating systems through VMMs. By virtue of its position in the system architecture, it helps to harden the overall systems security against local system faults or a security breach in one of its dependent operating systems [35].

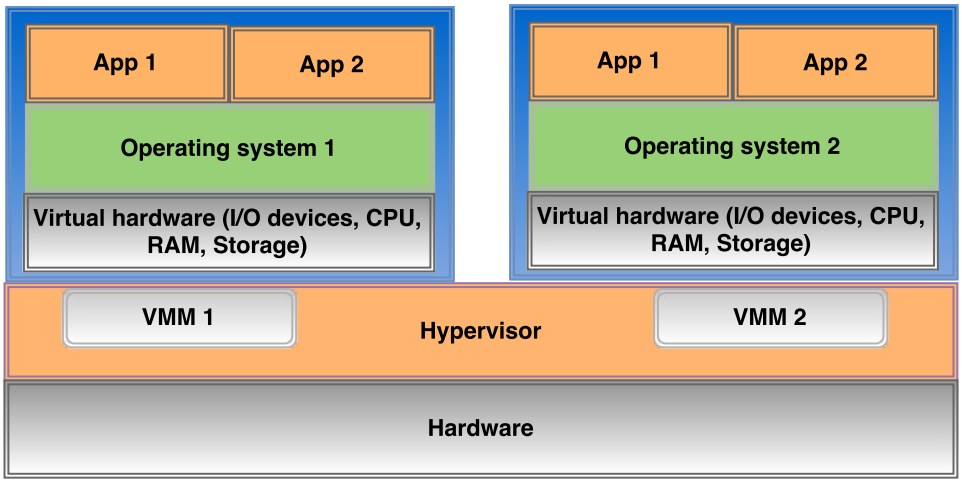

Figure 2.4: Type 1 (bare metal) hypervisor

- Type 2 Hypersivors:

  This type of hypervisor is installed on an operating system to provide a layer of support for guest operating systems. The hypervisor relies on the host operating system to execute user and kernel mode calls from the guest operating system, on the hardware. Type 2 hypervisors allow for the host operating system to be used for arbitrary purposes, but present a less hardened security barrier than type 1, where a breach of the host OS will affect the entire system, irrespective of the security arrangements made for the overlying operating systems [35].

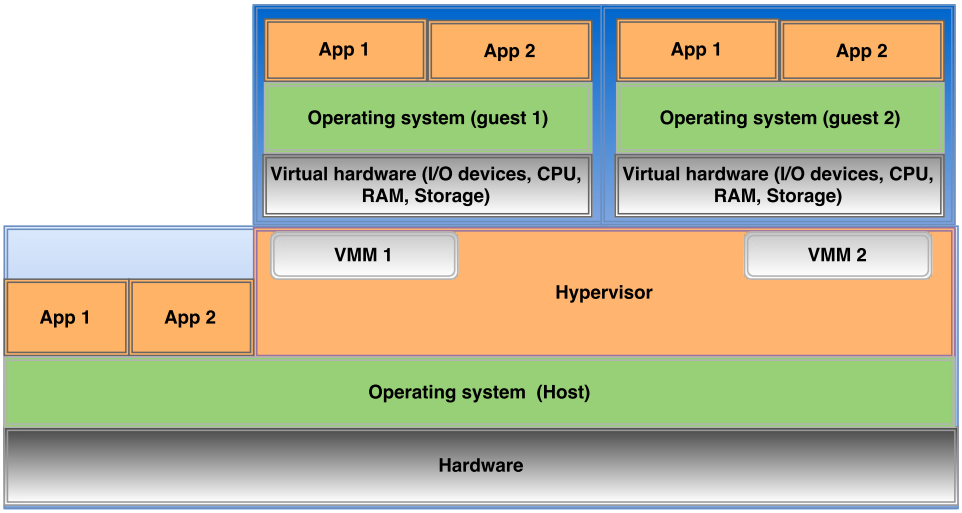

Figure 2.5: Type 2 hypervisor

2.7 Cloud Computing

Cloud Computing is the offloading of IT activities to one or more third parties with easily accessible, available and suitable resources which satisfy the

efficiency needs of an organization, such resources include hardware components, networking subsystems, storage, and software. Several attempts have been made to defined the concept including the following formal definition put forward by the National Institute of Standards and Technology (NIST):

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [36]

Cloud computing gives rise to a model where users are billed for remote access to IT infrastructure, rather than acquiring a physical data center to satisfy their enterprise computing needs. This means dedicating expending less real-estate and budget on acquisition of IT infrastructure, while delivering the same level of IT service. Users are given access to cloud-based computing resources, which can be scaled up or down at will.

The emergence of numerous high-capacity networks, the year-on-year doubling of computing power with inverse trend in pricing, and the evolution of hardware virtualization has led to the rapid growth and popularity of cloud computing.

The following are some of the key attributes which have contributed to the popularity of cloud computing:

- Companies no longer need to own data centers, but can acquire the compute capacity they require from a choice selection of cloud service providers.
- Through a transparent system in which cloud providers state Quality of Service (QoS) terms in proposed Service Level Agreements (SLA), clients are availed sufficient information to make decisions about the providers whose services best match the computing reliability needs.
- Cloud providers provide user-friendly interfaces to their platforms to enable users make optimal use of services. These include graphical user interfaces for regular users and administrators, and programmable interfaces for developers who wish to customize functionality.
- Unlike physical data center where a mix of availability of physical space, bureaucratic red tape, budget and components delivery and configuration time may constrain the extent and rapidity of growth of the IT infrastructure, clients can dynamically scale up or down their resource usage as their needs change over time.
- Clients subscribe for a service and are billed for defined payment periods which cover the access period [37].

Services offered by cloud providers generally confirm with one of the following models:

- Infrastructure-as-a-Service (IaaS)

  This model is used to deliver hardware components to the client. These components include CPU, memory, networking, load-balancing, operating systems, and storage. Clients are given control of these resources and may build the type of service they desire atop them. This model may be referred to as Platform-as-a-Service (PaaS) when on-demand but non-administrative access is given to available hardware resources (same as listed above) so that the users may deploy a custom-built service. Popular vendors in this space include Amazon (Amazon Web Services), Microsoft (Windows Azure), Google (Google App Enginge), and Rackspace (Rackspace cloud).

- Software-as-a-Service (SaaS)

  In this model, clients subscribe to software applications - free or payed - to software applications offered by service providers. The user has no control over the underlying hardware components and operating system. Each user has the perception of total ownership of the application as they are able to utilize its full array of functionality uninterruptedly, but in reality the arrangement is based on a shared-access model, where several users concurrently make use of different instances of the same copy. The model drastically reduces payouts on licensing, and provides ubiquitous access to the desired service [36].

2.8 IncludeOS

IncludeOS is a minimal library Operating System written in C++ and meant to run services, as well written in C++, in a cloud environment (reference please). It's name is derived from the subtle fact that an application and this little OS are packed together at link-time by starting the program with an **#include<os>** directive. IncludeOS has built-in support for x86 hardware virtualization support and requires no dependencies other than those for the virtual hardware [38].

A key feature of the IncludeOS architecture is that a service in hosted exclusively in any IncludeOS instance. A service in this context means a comprehensive system which facilitates a complete transaction initiated by a user request. The service is bundled into the instance along with the minimal set of software libraries which it requires for it operation [39].

The build process for includeOS is straightforward but requires familiarity with systems and C++ programming. The following is an explanation of the steps which are typically followed in setting up a service in IncludeOS:

1. Installing IncludeOS means building all the OS components, such as IRQ manager, PCI manager, PIT-Timers and Device Driver(s), combining them into a static library os.a using GNU ar, and putting it in $INCLUDEOS_HOME along with all the public os-headers (the "IncludeOS API").

2. When the service gets built it will turn into object files, which eventually gets statically linked with the os-library and other libraries. Only the objects actually needed by the service will be linked, turning it all into one minimal elf-binary, your_service, with OS included.

3. The utility vmbuild utility combines the bootloader and your_service binary into a disk image called your_service.img. At this point the bootloader gets the size and location of the service hard-coded into it.

4. Now Qemu can start directly, with that image as hard disk, either directly from the command-line, using our convenience-script, or via libvirt/virsh.

5. To run with virtualbox, the image has to be converted into a supported format, such as vdi. This is easily done in one command with the qemu-img-tool, that comes with Qemu [38].

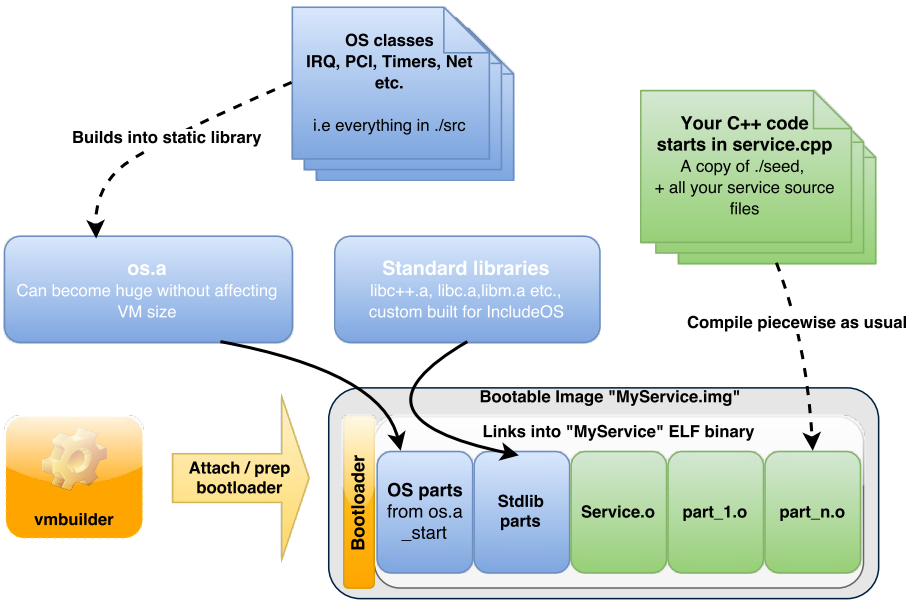

Figure 2.6: IncludeOS build-system overview. Reprinted from *IncludeOS*, by B. Alfred, 2015, Retrieved from https://github.com/hioa-cs/IncludeOS/wiki. Copyright 2016, by GitHub, Inc. Reprinted with permission.

Booting up the IncludeOS instance which will run with a service requires the following steps:

1. BIOS loads bootloader.asm, starting at _start.

2. The bootloader sets up segments, switches to 32-bit protected mode, loads the service (an elf-binary your_service consisting of the OS classes, libraries and your service) from disk.

3. The bootloader hands over control to the OS, by jumping to the _start symbol inside kernel_start.cpp.

4. The OS initializes .bss, calls global constructors (_init), and then calls OS::start in the OS class.

5. The OS class sets up interrupts, initializes devices, etc. etc.

6. Finally the OS class (still OS::start) calls Service::start(), inside your service, handing over control to you [38].

Some of the features of IncludeOS which are relevant to the discussion of Immutable MapReduce include:

- Very Small Memory footprint: The IncludeOS bootable image with a comprehensive software stack sufficient to support a full fledged service can be as small as 693 kilobytes.
- A modular TCP/IP stack designed with the following features
  - TCP: The bare minimum to support HTTP
  - UDP: Stable enough to support a performant UDP service
  - DHCP: Basic support
  - ICMP: Configured to extent of echo requests and replies
- IncludeOS deactivates timer interrupts while idle, significantly reducing CPU overhead on heavily booked hypervisors [38].

2.9 B-Tree

A B-tree is a method of optimizing placement and search for files in a database (http://searchsqlserver.techtarget.com/definition/B-tree). It is an algorithm designed with minimum access frequency in mind; it minimizes the number of times a medium must be accessed to locate a desired record, thereby speeding up the access process. B-trees supports the basic dynamic set operations which can be applied to tree structures including *Insert, Delete, Insert, Maximum, Minimum, Predecessor, and Search*. Common terms associated with B-trees are:

a Internal Nodes: they refer to all nodes with the exception of the leaf nodes and root nodes. They are usually denoted as an ordered set of keys of size $m-1$ and m child pointers. Each internal node must contain a number of children in the range $[m_L...m_U]$, where m_U must be $2m_L$ which implies that the capacity of each node is at least half full. m_L is known as the minimization factor (http://www.bluerwhite.org/btree/)

b Root Node: it marks the origin of the tree. It logically has the same upper limit $m_U - 1$ on the number of keys as the internal nodes with an absence of a lower limit constraint; therefore, when the total number of keys in the tree is less than $m_L - 1$ and tends to 0, there is the possibility that the root node will be the only node in the tree.

c Leaf Nodes: A leaf node has the same constraints on the number of keys it may contain as the internal nodes but has no child pointers hence no children [40].

Classically, the height of a n-node B-tree of height h with a minimum degree "t" equal to or greater 2 can be expressed as the inequality (Cormen, Leiserson, and Rivest):

$$h \leq log_t \frac{n+1}{2}$$

While the worst case height of the n-node B-tree is expressed in asymptotic notation as $O(log_b n)$, where b is the number of keys at a node with a worst case $= 1$ (in the practical case of a data store and search context, this would be the number of records per block on a disk) and n is the total number of nodes in the tree. Extending this logic, with every new level L_n added to the depth of the tree, the capacity of a B-tree can be expressed as the following:

$$L_{i+1} = m_u * L_i$$

A B-tree node may have many children, possibly on the order of thousands (see a, b, and c for node constraints). Each internal node of a B-tree may contain a number of keys, $n[x]$, which extends to mean that each node has $n[x] + 1$ children. The number of child nodes are usually fixed for a particular implementation of the B-tree. The set of keys packed into node x are used as a logical splitting mechanism in separating the range of keys into $n[x] + 1$ sub-ranges, where each sub-range is handled exclusively by a child of x (http://staff.ustc.edu.cn/ csli/graduate/algorithms/book6/chap19.htm)

To keep a B-tree balanced it is required that all leaf modes have the same depth (the distance away from the root node). As new nodes are added to the tree, its depth will gradually increase resulting in all leaf nodes being one more nodes further away from the base of the tree when the max number of keys per node is met. Values are also sorted to enable predictive search: for example, if a node has 3 sub-trees, which implies two keys a_1 and a_2, then all the values in the leftmost tree are less than a_1, all values in the middle sub-tree can be found between a_1 and a_2, and a_1 and a_2 are less than all values in the rightmost tree [40].

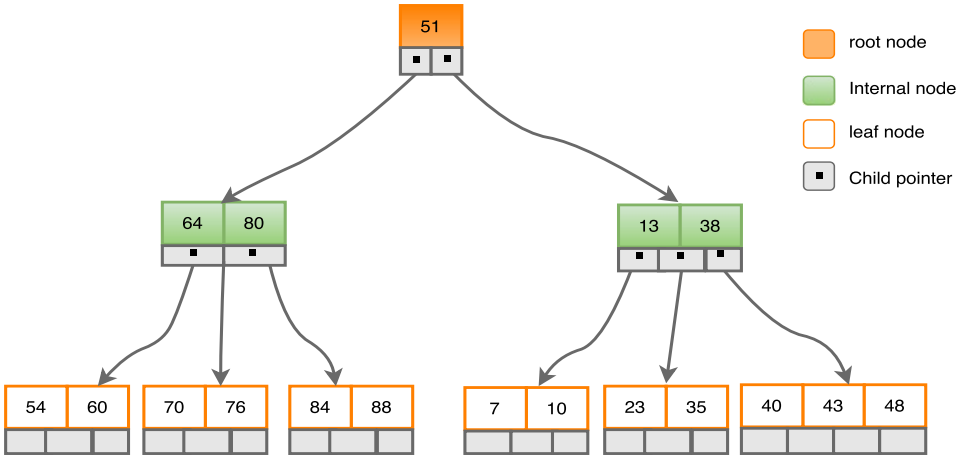

Figure 2.7: The keys in a B-tree are sorted with increasing magnitude from leftmost sub-tree to rightmost subtree

In his authoritative monograph on programming algorithms and their analysis, [40] states that a B-tree of order m is one which satisfies the following properties:

1. Each node of the tree has at most m children
2. Every no-leaf node, with the exception of the root, has at least $\lceil m/2 \rceil$ children
3. The root node has at least two children if it is not a leaf node
4. A non-leaf node with k children contains $k-1$ keys
5. All leaves appear on the same level

B-trees have found application traditionally in theory pertaining to speed-up in data search and access on storage devices; however there are other possible areas of application, even if superficial, as shall be seen later in this report.

2.10 Big O Notation

Big $\mathcal{O}$ notation, or simply $\mathcal{O}$ notation, is a mathematical notation which describes the computational complexity of a function as it tends towards a particularly large value or infinity [41]. It takes the growth rate of functions into consideration and permits the representation of functions which demonstrate the same growth rates using the same O notation. The origin of the symbol $\mathcal{O}$ can be found in the alternate term for the growth rate of functions, *Order of function*.

$\mathcal{O}$ notation is typically used in the analysis of algorithms for time efficiency; for example, the time it takes to compute a problem of size n. This time may be found in the arbitrary expression, $T(n) = 2n^2 + 2n + 1$, it is apparent that as n tends to a large number or infinity, then n^2 will dominate the other elements by several orders of magnitude, making their effect seem negligible [42]. If a new term such as n^5 is introduced to the expression, it would in turn exceed n^2 by several orders of magnitude. The efficiency of algorithm using the original expression may be expressed as:

$$T(n) \in \mathcal{O}(n^2)$$

Big O notation can be used in comparing algorithms of a definite size in order to theoretically estimate their difference in terms of time efficiency. Under such circumstances, O notation is said to denote the *asymptotic complexity* of the algorithms [43]. Some complexity cases which are commonly encountered when analysing algorithms and which may be used in comparing functions are captured in table 2.1 [41].

Table 2.1: *Note:* Big O notation look-up table. Adapted from CompSci 101 - Big-O Notation. Copyright 2015 by www.daveperrett.com

Function	f(n)	Big-O	Operation: 10 items	Operation: 100 items
Constant	1	$\mathcal{O}(1)$	1	1
Logarithmic	$\log n$	$\mathcal{O}(\log n)$	3	7
Linear	n	$\mathcal{O}(n)$	10	100
Log linear	$n \log n$	$\mathcal{O}(n \log n)$	300	700
Quadratic	n^2	$\mathcal{O}(n^2)$	100	10000
Cubic	n^3	$\mathcal{O}(2^n)$	1024	2^{100}
Exponential	2^n	$\mathcal{O}(n!)$	3628800	100!

2.11 Related Work

There have been some attempts at developing MapReduce frameworks, most with diverse goals and targeting different areas for optimization. This section presents a brief survey of some of the previous efforts related to this thesis.

2.11.1 Resilin

Resilin is an effort to improve and extend the functionality of Amazon Elastic MapReduce by performing computation over a range of resources pooled from multiple geographically disparate clouds, be they private, community or public clouds [44]. It supports data analysis through core Hadoop functionality which can be written using Hadoop oriented tools such as Apache Hive and Apache Pig - two tools which may be used for designing SQL-like queries.

Resilin distributes workloads over multiple clouds by allowing users to dynamically integrate or discharge virtual machines from different clouds to or from their active analysis infrastructure. The user also has the freedom to swap out Amazon proprietary EC2 virtual machine images for their images of choice. For reliability, Resilin relies on on a control service which acts as a monitor, periodically taking a pulse of all running service – where a service is one task request submitted by the user to the Resilin Web user interface - and restarting those which have failed. Fault tolerance is ensured by stateless redundancy of each running service. Inter-service communication is fully asynchronous to ensure seamless scalability [44].

However, Resilin incurs some performance overheads dues to inter-cloud connection latencies although results presented reveal these to be limited. Resilin shows immense promise in for multi-cloud MapReduce with future work planned for automatic cloud selection and configuration, saving the users the trouble of manual integration [44].

2.11.2 Benefit Aware Speculative Execution (BASE)

BASE is a framework which aims to robustly perform speculative execution of tasks in a Hadoop cluster. To accomplish this, the candidate tasks for speculation

are prioritized, fast nodes in the Hadoop cluster are selected to run the tasks on, and the number of speculative tasks are limited. Its Mantri monitoring module tracks execution of tasks and acts on those behaving abnormally (deemed to be outliers) by restarting such tasks, performing network-aware placement, and replicating the output of tasks which are classified as valuable tasks [45].

Resource utilization is improved through a technique called *resource stealing* which is applied at the resource level of each node. BASE operates on the theory that real data center utilization is low, and that the resources which sit idle on slave nodes - termed residual resources - can be exploited to minimize job run-time without incurring severe resource usage contention or degrading overall system performance. Residual resources are assumed to be reserved for prospective tasks which will be assigned to idle slots. BASE expands resource usage of running tasks when there are idle slots, and shrinks them to accommodate new tasks to that node. By implementing a transparent process of expanding and shrinking the resource usage of a task (resource stealing) the assumption of guaranteed availability of resources for new tasks is not violated. From the perspective of the central Hadoop scheduler running at the master node, there are idle slots on the slave node to which new tasks can be assigned [45].

According to tests conducted, BASE effectively eliminated a large portion of unnecessary runs of speculative tasks through subtle application of resource stealing which substantially improved performance of compute-intensive and network-intensive applications [45].

2.11.3 Resource-aware Adaptive Scheduler (RAS)

RAS is resource scheduler for MapReduce capable of improving resource utilization, and which is guided by user-defined completion time goals. RAS seeks to overcome challenges of MapReduce in terms of resource granularity and limitations of state-of-the-art solutions deployed to resolve problems associated with outlier tasks (straggling tasks) [27].

Hadoop views capacity as factor of the number of tasks which can run concurrently across the system; each node has a statically set number of task slots throughout its lifetime. Task slots are bound to either a map() or reduce() task, and a constant proportion of these tasks is enforced. A downside of this approach is that a task can execute in any slot, irrespective of the job with which it is associated, limiting the ability to efficiently control system resource utilization. User-specified goals are not properly catered to due to variability in performance resulting from the near-inevitable presence of outlier tasks. Current measures aimed at mitigating the effect of such outliers rely on scheduling techniques such as speculative scheduling, and task termination and restarts. These have the side-effect if higher resource overhead and lower throughput [27].

RAS overcomes the aforementioned challenges through the use of "job slots" as against the traditional task slots, thereby introducing a new and finer level of resource granularity. A job slot is an execution slot bound to a specific job and a specific type of task (map or reduce) within that job. RAS performs runtime

determination of job slot and their relatedness to nodes in the Hadoop cluster [27]. This evolves the behavior of a MapReduce cluster from requiring the system administrative chore of statically configuring global (cluster) and local (node) slot count and type, to a dynamic determination of such configuration settings [27].

Experiments conducted using a representative set of applications included in the Gridmix benchmark (a part of the standard Hadoop distribution) showed performance improvements in the aforementioned area of challenge: specification of job slots through resource profiling over time, and adaptable placement of slots in response to changing conditions in resource demand and availability [27].

Chapter 3

Approach

This chapter outlines the methodology and steps taken in order to address the core question at the heart of the problem statement: *How to design an adhoc MapReduce architecture which optimally and efficiently utilizes system resources to reliably process big data workloads and deliver results in a timely manner?*. Its organization follows the order in which the rest of the thesis work shall process.

3.1 Objective

The main objective of this project is to explore how an adhoc MapReduce framework which is resource and time efficient, and is able to adapt its compute scale in response to job demands, can be built in an environment of choice, be it a private or public cloud, or a physical data center. The framework is geared towards processing of batch data and precludes stream processing.

For this purpose, two alternative frameworks will be developed, each patterned after an adaptation of the B-tree data structure (see section 2.9 for more details) albeit for a virtual network topological purpose, and will utilize some of the technologies described in the background section to demonstrate their lean computing and self-scaling features. To avoid confusion with future references, framework refers collectively to the B-tree adaptations which are discussed in this thesis.

Established MapReduce frameworks, flag-shipped by Hadoop, are based on workflows which require predetermined infrastructure layout in terms of cluster size, node addressing and distribution of data partitions. The system is setup once, and is kept ready for numerous jobs over time. The use of a distributed file system and application management mechanisms help facilitate such workflows. In such setups, the degree of processing concurrency (number of task slots) is predetermined by the system administrator or user, and is the same for all the nodes of the cluster. The input data may be split up to the same degree or less and sited evenly throughout the cluster.

The possibility of developing an alternative framework as a modification of available implementations of MapReduce, especially Hadoop, was considered in order to utilize their rich, robust feature set; however the key functionality that

they emphasize for job management restrict their use to a Local Area Network, and their designs are difficult to tweak to achieve the performance goals at the core of this thesis.

A B-tree-inspired approach provides a novel method architecture for MapReduce systems where the initial size of the cluster could be nil, and the eventual size determined by workload dimensions. With this framework, the infrastructure is in place for only the duration of one job cycle. The idea is that task-discovery will be a feature built into the infrastructure rollout strategy and that resources are allocated in a Just-in-Time manner to accommodate the size of incoming jobs hence the size of the cluster will grow linearly with task size. The proposed MapReduce framework uses the standard MapReduce concept of jobs being composed of tasks which can be assigned to task slots across a cluster; however, it will use this concept in a dynamic sense, where the number of task slots is a function of workload size and available resource capacity. Resource wastefulness is negligible since there is no application idling, coupled with the use of components designed to provide the bare essentials for the running applications. This approach also circumvents overhead associated with active job management, and offers cloudbursting opportunities since the MapReduce fabric is not tightly bound to a distributed filesystem, and by extension a LAN. Cloud bursting is a deployment strategy in which an application is operated in a private cloud or datacenter, and requisitions resources on a public cloud when demand for computing spikes.

A different approach could be the use of a framework which conceptually mimics the traditional MapReduce process flow and layout including a predetermined number of task slots, with the key difference being that this type of cluster only lives for the duration of one job cycle, and does not feature a distributed file system. It also uses the Just-in-Time approach of the proposed B-tree inspired frameworks. Conceptually, each node of the cluster is created to play an exclusive role, where they could be either a mapper or a reducer, with there being as many reducers as mappers. One key advantage of such a system is that due to the clear delineation of roles, job execution can be tracked, and delays or failures owing to faulty nodes or network-based challenges can be quickly detected and possibly fixed. Conversely, the system will be efficient and manageable when a small cluster is in use, but will be unwieldy for instances where the number of task slots has been set quite high as there will be a considerable number of mapper and reducer nodes. This would introduce design complexities due to the nature of intra-cluster network communication. The approach of setting a maximum number of task slots is also flawed, as cluster might either be overloaded or a good portion of their compute resources left idling.

One of the goals of this project is to investigate the viability of the proposed framework though the implementation of working prototypes by following steps which can be also be used to replicate the environment, when followed in the same order, or modified to suit a desired purpose. Some constraints may challenge an attempt at reconstructing the work done here, such as cross-operating system platform equivalency of tools, operating system type, cloud platform, custom scripts and infrastructure deployment tools. Nevertheless, the framework holds

potential performance benefits for minimizing resource utilization and delivering results within deadlines for variably sized workloads. It offers the opportunity to reallocate freed-up resources for other purposes, and possibility of nominal expense on public cloud capacity which will only be needed under cloud bursting circumstances.

The project work has been structured into three major phases, these are:

1. A design phase.
2. An implementation and experimentation phase
3. Measurement, analysis and comparison phase.

3.2 Design

In this phase, the planning of the framework alternatives will proceed with the steps outlined below:

1. Explore 2 types of B-tree derivatives – A flat tree structure of constant depth, and a single-path tree (here on in referred to as a chain structure with hanging leaves) of variable depth
2. Establish correspondence between tree and MapReduce contextual terms.
3. Design framework with the proposed tree-based, resource saving features.
4. Create two algorithms, 1 for each of the tree based MapReduce frameworks
5. Perform estimation and comparison of runtime performance using Big O Notation.

3.2.1 Tree Structures

The first step in the design phase is a presentation of the alternative B-tree-like structures in view. The aim of this exercise is to clearly outline their characteristics, whereupon distinct features which can be adopted for a MapReduce context become visible. It is assumed the two distinct tree derivatives, which inspire the proposed MapReduce designs, exhibit some characteristics of a B-tree including a node having more than 2 leaf nodes. Where they differ is in the nature of their growth as new nodes and leafs are added to the tree.

Of key interest is the impact of depth and spread of the trees on the efficiency of their corresponding technical designs. To put these abstract structures in better perspective for the purpose of this thesis, a one-to-one homomorphism of roles in the tree structures and the technical equivalents will be maintained to aid the work ahead. Formalising the terms drives conformity with the desired namespace, and makes the concepts and implementations intelligible to individuals or organized entities interested in the design and results of this exploratory project.

3.2.2 Design of the Proposed MapReduce Frameworks

The design of the frameworks will reflect the qualities of a tree which support setting up a scalable adhoc MapReduce infrastructure in the cloud or optionally, in a partition of a data center. Fresh capabilities which are not apparent in abstract trees but are otherwise prominent in the technical realm of computing such as the ability of a cluster to fan out over multiple clouds may be factored into the design. The framework will be illustrated mostly with diagrams which concisely capture their expressed intent and possibly evince new use cases - the algorithms described in the next section will closely follow these artifacts.

3.2.2.1 Cluster Node Features

Since resource efficiency is a goal of this project, attention will be cast on the constitution of node servers in terms of their operating system, process diversity and the possibility of limiting CPU and memory overhead, as well as the nature of intra-cluster communication with a view to placing a limit on network bandwidth consumption. The underlying hardware plays a crucial role in determining the capacity to efficiently process workloads and the extent to which the fabric of the proposed MapReduce framework can be stretched in terms of growth in node population.
The hardware will be sourced from an Alto cloud tenant resource pool, where they have been quantized into the following flavors:

Table 3.1: Openstack Resource Flavors

Name	VCPUs	Total Disk	RAM
Micro	1	0	64
Tiny	1	1	512
Small	1	20	2048
Medium	2	40	4096
Large	4	80	8192
Xlarge	8	160	16384
HPC-medium	16	100	32768

The flavors of interest range from medium to hpc-medium with a preference for the higher scale since more virtualized nodes can be supported per CPU, with less chance of memory exhaustion.

The OS should be one which features just the necessary set of functionality to support needed programs including those which encapsulate map and reduce functions. IncludeOS is a fine candidate for this purpose. Another candidate is a Docker container with a base general purpose operating system. In this area, IncludeOS is the facility of choice as it helps simplify the process of resource allocation, is lightweight, and isolates the execution of applications so that the possibility of resource contention is greatly reduced. This doesn't discount the suitability of Docker containers for the same purpose. Where need be, a general purpose operating system may be considered for use.

A balance has to be struck to ensure that a virtual server is neither starved of essential hardware quota nor over-supplied with hardware which could be deployed for improved performance yield in other areas. A combination of unikernel operating system and one of the Openstack hardware flavors in table 3.1results in which are immutable throughout their lifetime. An attempt at modifying such entities will be tantamount to starting a fresh job and by extension a new environment.

Also of importance is a job's Time to Complete (TTC). The framework should roll-out a workflow which can spawn the needed environment in good time, enable a good degree of parallel workload processing, and eventual submission of results within a reasonable amount of time, where a "reasonable amount" refers to an arbitrary value measured in minutes which is estimated based on experience. From the foregoing, there is the corollary that the framework should follow some policy or be trained to detect thresholds which will trigger an offload of some workload to new nodes. Similarly essential is the ability to perform some self-cleaning task upon job completion to reclaim the dedicated resources.

3.2.2.2 Fault Tolerance and Redundancy

A typical MapReduce System provides fault tolerance and redundancy. Here fault tolerance refers to the ability for a failed node or task to be detected on time and restarted with immediacy. Keeping redundant copies of data on multiple nodes makes this possible in combination with active global job and local task monitors. To build fault tolerance into the proposed MapReduce framework, redundancy of workloads may be used. One or more nodes containing duplicate workload are required to run simultaneously with the primary to ensure uninterrupted delivery of results. When working with limited resources, this approach may prove antithetical to the objectives set out for the framework as it requires a relatively strong resource commitment. A similar approach involves keeping an inactive backup node which is activated upon failure of the primary. This approach demands a lower resource investment but requires more complex programming and an active monitoring facility which comes with some resource overhead.

In lieu of creating redundant workloads, a message-oriented middleware may be incorporated to hold message queues, where workloads can be pushed unto a queue to trigger the creation of an environment and distributed processing of workload. Results are labelled with their source ids and this will serve as a control feature in learning which workloads have not been processed within a specified time.

3.2.3 Algorithms for Tree structures

A self-contained step-by-step set of operations for the flat tree and single path tree will be developed as templates from which the prototypes will be created. The algorithms will reflect their subtle differences in the approach to scaling a compute cluster and the decision points which trigger defined actions. They will also indicate the type of data structures that are passed as input and output between titular components of the cluster. Both algorithms will feature functions

for growing or extending the cluster by continually distributing the workload until it is fully apportioned.

Some constraints which will influence the decision making process are:

1. Number of CPU cores and allocation strategy

2. Random Access Memory size allocation

3. Amount of data to be processed and data block size.

It is expected that these factors will remain constant for the alternative architectures, but the point at which decisions are taken during the environment roll out process will have a different TTC side effect for each. With the above constraints well defined, some major decisions can be taken including:

1. Determining the nature of relationship between the number of mappers and number of cores – by extension nodes.

2. Deciding if a cluster should be extended based on known CPU and memory availability levels.

3. Deciding the role of a node based on secondary constraints such as the amount of data, relative to a per-node size limit, yet to be assigned to a mapper.

To get an idea of the limiting factors of each algorithm in terms of runtime, Big O notation will be used to perform an analysis of algorithm. The result will reveal the asymptotic behaviour of each algorithm as the task size gets bigger and more complex. For more on Big O Notation, see section 2.10.

3.3 Implementation and Experimentation stage

In this phase of work, some deliberation and decisions are made about how the algorithms can be implemented in an effort to meet the design goals. Some of the tasks in this phase include:

1. Determining the tools that best suit the prototypes implementation task

2. Building the base infrastructure of the proposed MapReduce framework with deployment scripts.

3. Creating prototypes by following the earlier developed algorithms

4. Performing tests to ensure conformity with design goals

3.3.1 Tools Required for building the prototype

The implementation process is two-fold: setting up a suitable environment for developing the prototypes and for running credible tests, and building the prototypes. Both tasks require resources and a suite of tools. The suite of tools have to be groomed to contain the best-fit tools, not necessarily the best-of-class tools, for the job at hand. The following is a short discussion on the nature of the tools needed and the specific selections:

- Configuration repetition and consistency

  The proposed MapReduce framework will feature one or more nodes which will host mappers and reducers. It is essential that functionality is identical for all mappers and reducers so as to avoid inducing varying processing conditions which are not asymptotic with the expected performance. Through automation this objective can be achieved, and two clear candidates which dominate and could be deployed, mutually exclusively, for this purpose are a Configuration Management System (CMS) and a parameterized script. A configuration management tool is a useful asset in ensuring that there is no drift from a desired state by performing a cluster-wide check and returning all local system environments to that desired state when a deviation in configuration is detected. This desired state is expressed for every node shortly after domain membership. Two great benefits of the generality of CMS are:

  - That the desired state, which is a documented high-level policy or group of policies, can easily be expressed as a sequence of domain specific language commands which are idempotent across multiple operating system platforms or a collection of similar operating system in different configuration states.
  - It can be used to manage machines numbering in the order of thousands.

  A parameterized script, in its right offers the flexibility to enable a level of automation reproducible for different environments, with the necessarily library dependencies in place. In the context of the proposed MapReduce framework, configuration management suffers a drawback given that the framework is short-lived (hence the likelihood of drift is negligible), the cluster may be composed of as little as one node for light jobs, the applications may be baked into immutable nodes (therefore state is constant), and that there may be instances of cloud-bursting – which a CMS may not be able to manage properly. Scripts on the other hand require some manual management to adjust evolving parameters prior to automating a job, coupled with cross-platform issues which may arise due to differences in library composition; however a script does support the inverse of the drawbacks highlighted for CMS, when managed diligently: it can be used to manage a small cluster of nodes, imposes no limitations beyond the programmers creativity for instances of cloud-bursting, and could help start

up the services in very little time with no concern about state drift. For the environment start-up scripts, one written in Python or Bash should do the job efficiently.

- Application development

  The key functional components in the framework are mappers and reducers, along with support features which facilitate a channel of communication, workload distribution, and reliable delivery of results. The code for the mappers and reducers will be done in C++ so that they can be compiled with and wrapped-up in IncludeOS instances. There is the possibility of using Python programming language to implement aspects of the framework which are easier done in this language than C++. Beyond the core application development, Python will be used in writing scripts to facilitate a coherent workflow which begins from the point a new user request is received till the successful delivery of results or the occurrence of terminal event.

- Creation of Nodes:

  It is crucial that the environment be rapidly and easily reproducible to facilitate tests, for future research and other purposes. To this end, a tool capable of spawning a desired number of nodes on demand is required. As the favoured environment is the cloud, the tool should be able to consume the API of the supporting cloud management platform in provisioning the specified hardware resource set and initializing the system environment. One tool which is popular in this space is Manage Large Networks (MLN) [46]. MLN is a system administration tool for simplifying commandline management of virtual machines at scale.

- Splitting Data:

  A crucial aspect in preparing for a computation is splitting up a body of big data into sizable chunks based on a specified block size. There are several built-in OS and 3rd party tools for this purpose. In Linux, two top contenders are *Split* and *7z*. Split has been chosen for this purpose for no reason other than perceived simplicity and familiarity. Splitting a data file of any type with this tool is as simple as executing the following command:

  split –bytes=1M /path/to/file/file.ext /path/to/image/prefixForFilePieces

  The above command will split file.ext into chunks of 1MB and label the chunks with a combination of the PrefixForFilesPieces and a character from an ascending sequence of characters, which may be numbers or letters depending on choice. Split can be split files with other strategies beside bytes boundaries, such as number of lines per split.

3.3.2 Environment Deployment Scripts

The implemented framework will be characterised by the ability to deliver the required node cluster size based on the dimensions of a job and administrative constraints. In a generic sense, there is usually some tolerance for manual

configuration and management of a system built on a small cluster of about 1 to 5 nodes, however as the demands on the system and more nodes are added to reinforce it, the system gradually becomes unwieldy and soon the inefficiency of manual effort is exposed. This project relies on the ability to rapidly launch a cluster of any size and therefore leans favourably toward an automated or semi-automated deployment method.

The job of bringing up an environment can be split into two technical parts: one which involves a deployment framework for defining the parameters of virtual machines: the hardware specification including CPU, RAM, primary storage, and number of network interfaces; an operating system; configuration of the network interfaces for addressing and network reachable; and a hostname for easy recognition within the management domain. The second part could be seen as setting up of a layer of execution which leverages the first part. It marshals the aforementioned deployment framework, splits and apportions data splits, and distributes the MapReduce application designed for parallel execution. This orchestrates a comprehensive system based on the proposed MapReduce framework, and oversees end-to-end workflow.

The deployment framework defined above is provided by MLN and is called by a Python script which forms the control and monitoring layer. MLN will be configured for commandline management of the virtual machine creation process on Openstack. Using MLN templates, a good number of virtual machines can be spawned in quick succession by issuing MLN build and start commands. This is useful when managing rapid deployment of a large number of virtual servers on a private Openstack cloud as well as a public Openstack cloud. A limitation of the use of MLN in prototyping our framework has to do with lack of support for IncludeOS on the generality of Openstack platforms. Barring this limitation it is a potent tool for driving the efficiency of the MapReduce framework. As part of its duties, the Python script will provide the necessary components used for initial system configuration of virtual servers including designated data partitions.

3.3.3 Build and Deployment of Prototypes

Two prototypes will be built based on the designs for the flat-tree and chain architectures. The prototypes will have the same base features, but will implement distinct roll-out strategies - this will be reflected in their designs. The idea is to make the prototypes convergent in terms of the applications which run in their individual nodes and the data they are meant to consume (in other words, the workload).

IncludeOS requires is guest applications to be written in its base language which is C++, therefore all mappers will have to be written in this language. Likewise, the reducers will have to be coded in C++ if they are to be hosted by IncludeOS instances, although there is the potential to express them in a different language such a Python or other languages which provide the required libraries. A Python script will assume the responsibility of configuring roles and establishing communication links between nodes. By and large, the Python script

will be responsible for the complete automation of a Lean MapReduce service, job execution and system tear-down

3.3.4 Objectives Conformity Tests

Performing an experiment requires no deviation in the characteristic behavior of the system when tested over and over again for the same set of parameters. It is therefore necessary to test the constituent parts of the system to ensure that they function consistently. The prototypes be will tested multiple times whilst the workload is varied to reveal any bugs in the functional components and the system as a whole. The tests will also confirm the ease with which relevant data can be collected.

Among the checks to be executed are:

- Tests to ensure the virtual machines are created with the right hardware specification and that there is proper connectivity between worker nodes and the root node.
- Test that data partitioning and distribution strategy works as expected.
- Ensure that data is processed correctly by the distributed map and reduce application, and that all results - intermediate and final - are produced and passed along in the right format.
- Idempotence of workload computation with asymptotic resource consumption and execution duration

3.4 Measurement, Analysis and Comparison

Working prototypes present an opportunity to verify to what degree the mission contained in the problem statement is viable and to learn to some extent the margins of improvement or modification which are immediately feasible. To this end, measurements have to be taken for key performance parameters.

In this phase of project work, experiments are conducted with a view to gathering empirical data which can be transformed into a more meaningful information through the use of graphing tools and analysed with statistical tools to help make sense of the systems performance. To enable this process the following shall be performed:

- Creation a mechanism for capturing data concurrently with the Lean MapReduce runtime.
- Performing some experiments
- Data analysis
- Comparison of data to judge relative performance

3.4.1 Data Classification and Capture Mechanism

The capture mechanism will be managed as an intrinsic part of the environment roll-out and management process. Monitoring and measurement of performance element will happen in parallel with the core map and reduce applications. Hypervisors provide an effective layer of abstraction where activity of the community of guest operating system can be measured centrally in terms of the resource impact they have on the host system. Monitoring will therefore take place at each hypervisor with the goal of polling the same kind of performance parameters, specifically percentage CPU consumption and memory usage. CPU measurements can be read, with a good degree of precision, in "ticks" where one close tick is the maximum number of calculations a single CPU core can perform in a second; however %CPU consumption provides an equally good estimate as well as a more analysis ready decimal precision.

A third performance facet, Time to Complete (TTC), is a factor of the entire system and hence will should be calculated centrally at the root node. Job processing time is made up of two components which are equally significant whether considered together or in isolation: the first component is the time it takes to setup the adhoc computation infrastructure while the other component is counted from when processing begins till when a complete result is emitted. Time measurements can be tracked by using running timers or timestamps; the latter requires that all system clocks - for map and reduce hosts - are synchronized, most preferably, with an Network Time Protocol (NTP) server. Local time measurements will be taken but only to detect run-time anomalies.

The monitoring mechanism, which will be scripted in Python programming language, will marshal the unique features of system tools specialized for the purpose of monitoring resource usage and generating point statistic summaries. The tool to be used will depend on the type of hypervisor in use. For example, a Linux KVM hypervisor has built-in tools like *Top* and *ps*, and third party tools like *pidstat*, which can be used in isolating the statistics bound to the operation of particular processes identified by their process identification number or name. The virtual machine and hypervisor view of available resources and level of usage are somewhat different, where the VMs are only aware of their allocated resource while the hypervisor which serves the VMs resources is aware of actual resource consumption. This justifies the choice of monitoring resource overhead at the hypervisor. Not only that, by centrally generating such reports, less data is processed - yet sufficient data is used - hence there is comparatively less I/O, and lower memory and CPU overhead.

Resource usage data will be polled at intervals on the order of seconds. At each hypervisor a report will be generated in a consistent format to support central aggregation. The resource usage reports from the hypervisors will be transmitted to and collated at a central location, ideally the root node. The individual resource usage report will carry the following structure:

Proposed Resource Usage Report Structure

```
CORPUS SIZE -%CPU CONSUMPTION - MEMORY USAGE
```

While the job processing time will have the following fields

Proposed Job Processing Time Report Structure

CORPUS SIZE - SETUP TIME - APPLICATION EXECUTION TIME - AGGREGATE TIME

The fields in the reports will be delimited by space. This gives it a structure, making it possible to import in a statistical analytics tool of choice where the data can be charted and analysed.

3.4.2 Experiments

Experiments shall be performed in an environment which features one or more Ubuntu Linux virtual machines version 14.04, at least one or as many as needed IncludeOS instances running on top of a KVM hypervisor, and a virtual network for intra-cluster communication and data transfer. The root node of the cluster will have the Openstack X-Large flavor (see table 3.1 for more details). Computations will take place in Alto Openstack Cloud - an institutional cloud platform used for research and educational purposes at the Oslo and Akershus University College (HiOA) - and might extend to Amazon EC2 in occasions of cloudbursting.

Experimentation will involve the processing of a set of workload - which will be incremented on an interval scale - by each of the prototypes. The test data is text corpora - as earlier alluded to - which can be generated by using a simple scripted tool, or retrieved from free electronic corpora databases accessible over the internet. One or a combination of these sources will be employed in this project.

During each iteration of the experiment, the data capture mechanisms will be deployed in parallel with core Lean MapReduce infrastructure. To limit the possibilities of systematic errors which will render data unreliable, all non-essential processes which are always active or run as periodic jobs on the hypervisors and the virtual machines will be terminated. After each iteration, data will be analysed using descriptive statistic checks such as standard deviation, which can be visually appreciated when plotted as a boxplot, to verify the absence of such anomalies. The presence of anomalies mandates that the cause of such errors be identified and corrected, and the experiment repeated with the same job parameters.

There will be multiple iterations for each workload to obtain reliable data so as to determine to a good degree of certainty, the trend of the measured performance facet. To limit the possibility that the measured performance was due to chance, a large enough data sample must be collected to obtain an estimate which is close to the "true" performance of the prototype under test; since the rule of thumb holds that the sample size is enough if it is greater than 30, the sample size for each performance variable will be fixed at 100.

3.4.3 Data Analysis and Performance Comparison

This phase of work will involve inspecting and summarizing the data sets in order to see if and to what degree project objectives have been met. The bulk of the analysis will be done in RStudio.The key variables which will be used are corpus

size, TTC, and resource consumption - separated into %CPU consumption and memory usage. The analysis technique will be a bivariate analysis, where the corpus size would be the preferred independent variable.

The dependent variables have been so proposed because of their relationship to the problem statement: %CPU consumption and memory usage provide clear quantitative measurements of how much resources a running application utilizes for a defined period and workload; TTC provides a quantitative measure of how much time it takes a system to conclusively process a job where in this case the job type is a word counting job on a big data scale. From these variables, veritable parameter estimator can be derived by using inferential statistical.

Analysis connected to system resource utilization will reveal whether idle system resources are being optimally utilized and also the resource impact of adhoc jobs. While TTC will give a sense of the relative time needed for a system based on the proposed architecture to complete a big data analysis job. These will help answer the "optimality and efficiency" and "timeliness" elements of the problem statement.

The analysis will involve a comparison of the performance measurements of the Lean MapReduce implementations. A 95% confidence level will be used in the computation of confidence intervals which will be reasonably tight since a data sample size of 100 is to be used for all variables. By performing a two sample statistical test involving the data sets from the two prototypes it will be determined whether there is a statistically significant difference in their performance measures. A noticeable advantage of one prototype over the other in one or more areas of performance will not result in the repudiation of the latter, but will extend the analysis to reveal less obvious features which open up potential advantages in other areas of application besides text corpus processing

Chapter 4

Design

In this chapter, the outcome of the tasks described under the design section of the approach chapter. The abstract concepts which inspired the technical designs on which the eventual prototypes will be based, are presented as well as the technical designs and facilitating algorithms.

4.1 The Meaning of "Lean" In Lean MapReduce

The word "Lean" in the context of this thesis, bears a similar meaning to that implied in Lean Software Development, albeit in a non-software development sense. Lean MapReduce adapts three core principles of the latter for its purpose, these are:

1. Eliminate waste [47]:

   This implies proper management of resources, where the resources in question encompass several facets, prominent among which are compute resources (CPU, memory, and storage). Waste also implies manual effort put in by a system administrator or a regular user in configuring and troubleshooting a system. At the core of the attempt to eliminate waste is a simplified Just-in-Time strategy: compute resources are only allocated to a cluster as the dimensions of the immediate workload are discovered. Design complexities are transparent such that from the user perspective, they simply submit a request and get a reply after some time, while the system administrator is only saddled with the job of supplying the compute fabric. The resources are also reclaimed at the end of each job execution cycle. This approach is fundamental to a successful attempt at answering the optimal and efficiency aspects of the problem statement, through the proposed MapReduce architecture.

2. Build quality in [47]:

   One of the expected features of a MapReduce system is the ability to sustain computation across task-execution failures. This implies the presence of a fault-tolerant mechanism for increased reliability. In MapReduce systems, as in most distributed systems, "failure is the norm rather than the exception," [45]; therefore, the absence of a fault-tolerant edge will result in some system

administration effort to fix faulty components and restart job execution. This produces two major forms of waste: the effort invested in rectifying the fault, and the time spent doing so. Waste is a good indicator of quality issues [47], and in light of the foregoing, the proposed architecture must incorporate a fault-tolerant feature so as to meet the "reliability" requirement of the problem statement.

3. Deliver fast [47]:

   A high-level design goal of MapReduce is a big data system capable of meeting data analysis deadlines. This implies "timeliness" of results delivery, an element which is prominent in the central question of the problem statement. Fault-tolerance is one among several features which contribute towards meeting this objective. Some other attributes are: the quickness with which resources are provisioned, and the degree of execution parallelism.

Based on the foregoing, it is apparent that the architecture being proposed should support the creation of an adhoc MapReduce system which can be rapidly grown into place and taken down at will. When simplified, this behavior is reminiscent of the behavior of a B-tree (see 2.9), hence the reason why it has been adopted as the base upon which the proposed MapReduce architecture will be built.

4.2 Abstract Structures For Lean MapReduce: B-Tree Derivatives

The design of the Lean MapReduce framework is inspired by the shape and properties of the B-tree data structure. The framework features a pair of alternative structures which are modifications of the B-tree. The structures are called a chain structure with hanging leaves, and a flat tree structure. For the sake of simplicity, these shall be referred to as alternatives A and B respectively.

The alternatives share the basic features of a B-tree including having a root node, zero or more internal nodes, and one or more leaf nodes (in a 1 node tree, a root node $\equiv$ a leaf node); however, there are a few subtle deviations in the properties of the alternatives from the traditional B-tree, particularly constraints imposed on the nodes which are relaxed for the case of the alternatives. Some of the differences are presented in the description of the alternatives below.

4.2.1 Alternative A

This structure may contain at least one node which implies that the root node is the only member of the tree for a one-node tree. A key is stored at each node with the exception of the leaf node. The key represents the length of an array of indices where each entry is an index number for one child: if $key = 3$ then there are 3 children indexed $[1, 2, 3]$. Only one index may have an associated pointer to a directly descended internal node.

Alternative A has a maximum number of allowable children for each node - known as the maximization factor. It imposes a minimization factor of 2 (see fig 4.2 for illustration) for the internal nodes and 1 for the root node: this breaks the B-tree minimization rule which mandates that the maximization factor must be twice as large as the minimization factor: $m_U = 2(m_L)$. Its height, when there are n nodes with a branching degree of t greater than or equal to 2, can be stated as:

$$h \leq \frac{n-1}{t} \qquad | \, n > t, \quad \geq 1 \, |$$

Since the structure is not meant for data storage or optimized search of sorted weighted values, the keys t stored at every node is used to maintain a count of the number of downstream nodes.

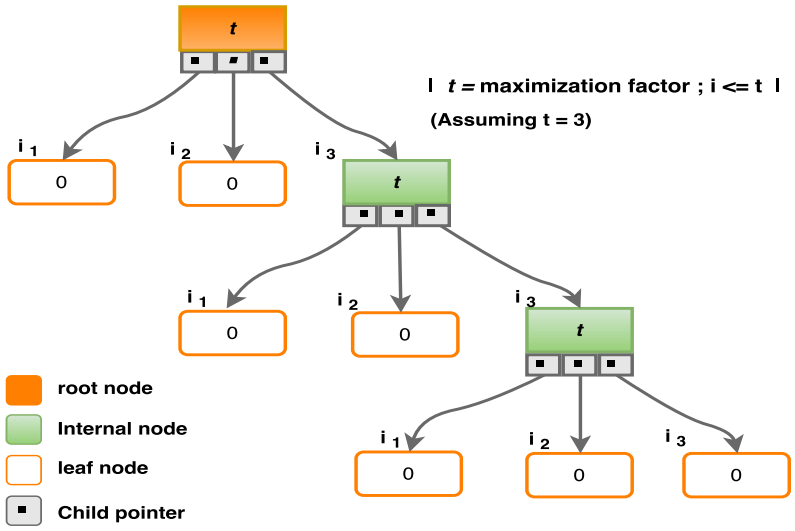

Figure 4.1: a 10-node Alternative A with a maximum branching factor of 3

A conspicuous deviation from the B-tree is evident in the relaxation of the rule for avoiding an unbalanced tree: keeping all leaf nodes at the same depth. Alternative A permits for leaf nodes to be at different depths which is why it is termed a chain structure with hanging leaves.

There are two primary operations on alternative A: insertion and deletion. The structure can be extended horizontally by addition of a new leaf node and vertically by attaching a new internal node. The increase in height of the structure is therefore influenced by the insertion of one node above the maximization factor. For example, an attempt to add 1 node over the maximization factor in fig 4.1 will lead to the insertion of 2 new nodes: 1 internal node and two leaf nodes downstream of it, and the displacement of the highest order leaf node. This is illustrated in fig 4.2.

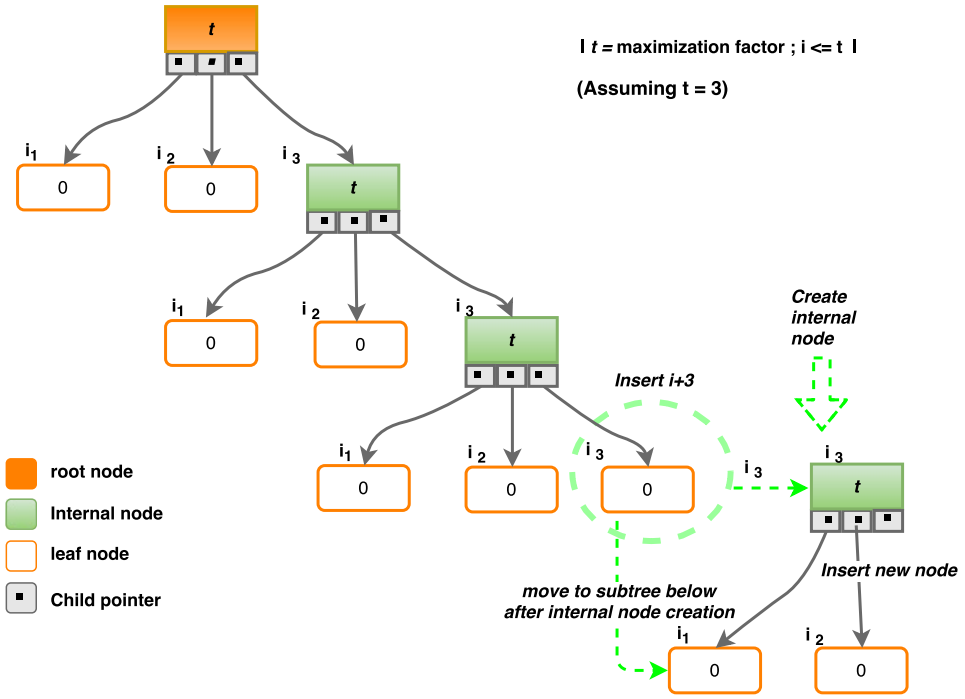

Figure 4.2: The addition of one node more than the maximization factor results in vertical growth

Shortening the tree requires the deletion of one node less than the minimization factor of the structure. Leaf nodes may be deleted in no particular order. It must be noted that deleting an internal node carries the implication of cascaded deletion for all child nodes and sub-trees. There is also the side effect of reducing the tree to a one element tree, if the internal node closest to the root node is deleted. This ramifimication is extended to the case of a root node, whereby the tree is destroyed when the root node is deleted

4.2.2 Alternative B

Alternative B is called a flat tree with a constant depth $h = 2$ (for all cases where the number of nodes $n \geq 2$) due to the nature of its growth about the root node. Just like a B-tree it can have as few as one child node (or may have non when the root node is the only element of the tree), and all leaf nodes are of equal depth. However, it relaxes the rules for the maximum allowable number of children by imposing no upper limit on the number of children for the root node. It specifies a maximization factor t for internal nodes which varies for each incarnation of alternative A. The minimization factor for the internal nodes is 1 while the root node is free of this constraint. Alternative B also enforces a rule which specifies that no direct descendant of the root node may be a leaf node, and all children of internal nodes must be leaf nodes.

In addition to a few of the properties mentioned above which they share, Alternative B also shares the "key" property with alternative A. This refers to the utilization of the length of an array of indices of the children of internal and root

nodes, as keys (refer to the 2^{nd} paragraph of section 4.2.1).

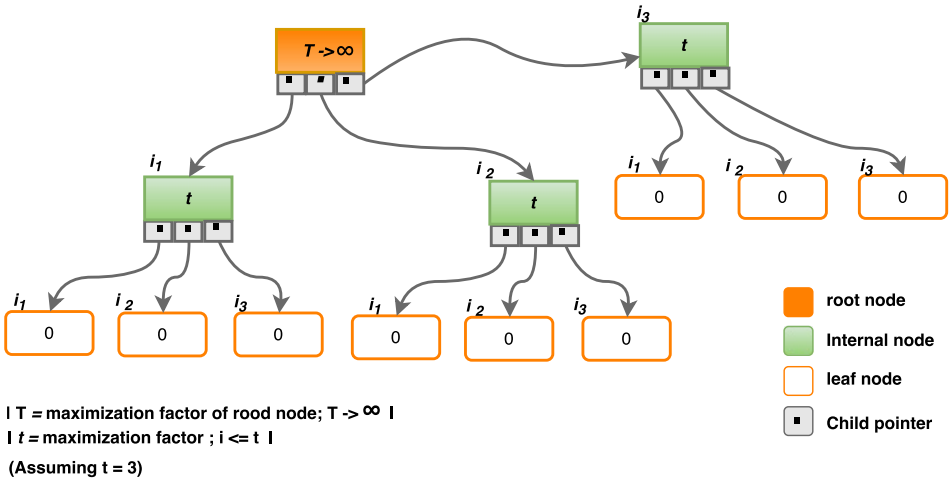

Figure 4.3: Alternative B comprises multiple sub-trees of equal height, all anchored at the root node

Alternative B also supports insertion and deletion operations, although the effect varies slightly from what is obtainable in alternative A. The structure starts with the root as the sole element of the tree. Noting that a root node can have no leaf nodes, when an attempt is made to insert a leaf node, an internal node is first created and the new node attached to it as a child (the internal nodes serves as a control element for maintaining the structure of the tree). Subsequent insertions will add new leaf nodes to that internal node until its maximization factor t is reached. Upon the next insertion, a new internal node - attached to the root node - is created and the cycle repeated. This way all the leaf nodes have the same depth, and as well all internal nodes, hence the term: a flat tree. The eventual structure after multiple insertions - k orders of bigger than the maximization factor - will feature k sub-trees of height $h = 2$, all anchored at the root node.

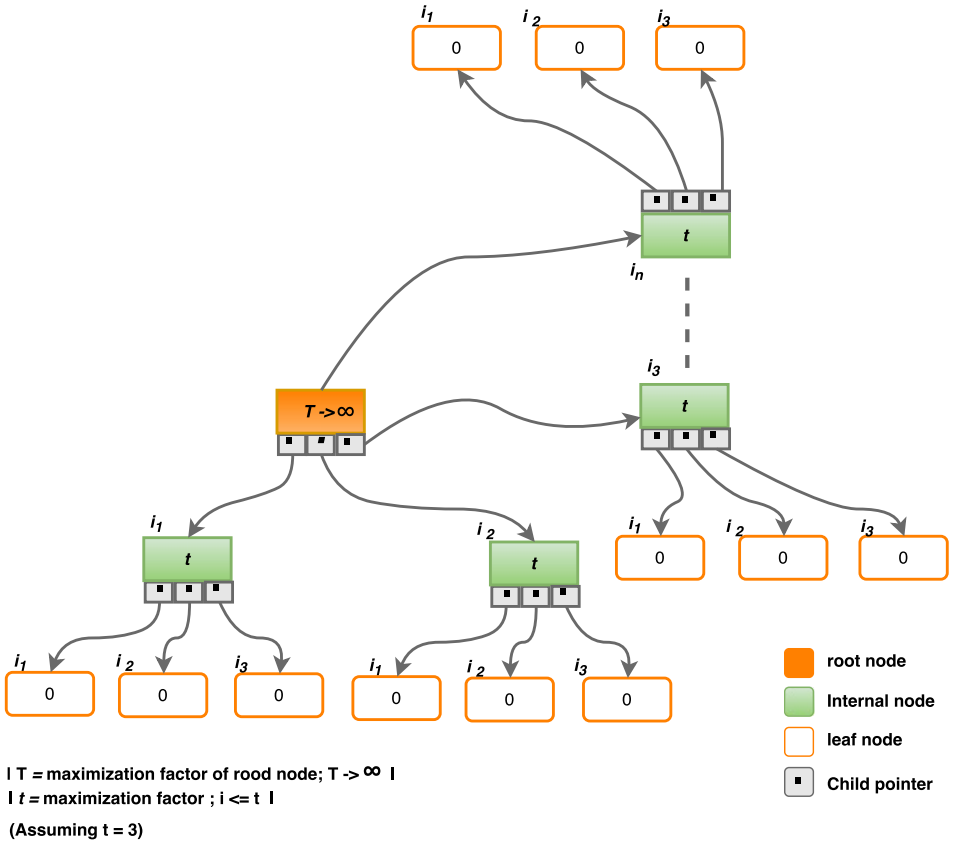

Figure 4.4: Alternative B grows about the root node with the formation of sub-trees all of size $t+1$

The size S of alternative B at any instant, can be expressed as the following inequality:

$$S \leq (n(i_{intNode}) * (t+1)+1)$$

$$| \quad where\ n(i_{intNode}) = \ No\ of internal\ nodes,\ and\ t = Maximization\ factor \quad |$$

The rules and outcome of deletion are unchanged from those of alternative A; however, the effect of deleting an internal node is isolated to its associated sub-tree.

4.2.3 Identity Correspondence Between Alternatives A and B, and Lean MapReduce

The growth of alternatives A and B commence at the root node. Their height or width may increase or broaden, as new nodes are inserted. They have capacity constraints, where an array of indices is used to track the offshoot or children of an internal node or the rood node. An internal node provides a base of support for the leaf nodes, just as the state of the root node determines the fate of the tree. The nodes in the Lean MapReduce architecture are much like their

abstract representational counterparts in that the latter provide the rudimentary set of properties which characterize the elements of Lean MapReduce; however the nodes in a Lean framework are virtual server machines (the leaf nodes are IncludeOS instances). As each node in the architecture performs a distinct role, the role-mapping between the abstract tree structures and Lean MapReduce is as follows:

- Cluster: is analogous to a tree. Just like a tree, it is a collections of cooperating nodes, categorized by function and position, all with the same root node. Each categroy of nodes perform a distinct function.

- Worker Nodes: a server where a single function is applied the local data at a particular point in the job processing work-flow. Typically, the functions is either a map() function - applied by mappers - or a reduce() function - applied by reducers. By definition, every node in a Lean MapReduce tree or cluster is a worker.

- Schedulers: is the equivalent of the root node. It makes decisions on the form of a cluster and packing density (maximization factor) based on the submitted job dimensions, for supervisors and itself (in the case of alternative A). It may be more intuitive to refer to the scheduler as a master reminiscent of a master in classical MapReduce; however, this is a misnomer in the MapReduce context as it does not control the actions of the other members of its cluster. A scheduler is also a specialized reducer which finalizes the job processing with a desired unit result.

- Supervisors: are the equivalents of the internal nodes. They act as intermediate delegation points for a finite set of mapper nodes, or a combination of mappers and a supervisor. The decision to extend a cluster lies with the supervisors. It possesses most of the capabilities of a scheduler but is distinguished from one by virtue of not being the origin of the cluster. In essence, the supervisors, by design and functionality, double as reducers.

- Reducers: The generic term for nodes where a reduce() function is applied to some intermediate data. The nodes may be either schedulers or supervisors.

- Mappers: mappers are essentially leaf nodes in the abstract representation. The mappers by design apply the map() function to local data as the first act of data processing.

- Links: it is akin to the path between two nodes at different levels of a tree structure. It serves as the communication channel for passing data between nodes of the Lean MapReduce Cluster.

Where alternative A and B abstract structures have a static maximization factors for the internal nodes and the root nodes, Lean MapReduce alternative architectures have a dynamic maximization factor which is related to the total amount of unused memory in the scheduler or supervisor system.

4.3 Design of The Lean MapReduce Architecture

The previous subsection was able to execute a transition in minimal vocabulary from that used for the abstract representations to the technical terms in a Lean MapReduce context. This subsection describes what has to be done - in terms of environment setup and preparation of orchestration tools - to successfully build working prototypes. Before elaborating more on these, it is worth reiterating that Lean MapReduce possesses the set of properties which characterise the abstract models, and extends it with a few more for functional efficacy in processing big data jobs. Prominent among these are:

- In every Lean MapReduce, a static system resource size (particularly memory) is configured for each task slot. The number of task which can be hosted by a node is a function of its allocatable memory, workload size, and task slot size.
- Each node is characterised by either a map or a reduce task exclusively, and holds some value which is the computation outcome. See node descriptions in 4.3.1
- After a downstream node (internal or leaf nodes) is spawned, it operates autonomously for the duration of a job cycle.
- The direction of transmission of raw data and processed data are inverse of each other: raw data travels downstream, while processed data travels upstream.
- A post-creation attempt at introducing or modifying functional variables (applications: map() and reduce() functions) or feature variables will culminate in an abortion of the currently executing job cycle and deletion of the servicing cluster.

In order to realise these properties, some decisions were made. These are organized into the subsections below. To ease the discussion about the B-tree derivatives, and to maintain a one-to-one correspondence between the abstract representations and their technical incarnations in the form of the Lean MapReduce alternatives, the terms, alternative A and alternative B - qualified by "Lean" - are carried forward to the succeeding subsections. The abstract representations will be qualified with "abstract" when referred to.

4.3.1 Cluster Node Features

As previously mentioned, three distinct types of nodes feature in a Lean MapReduce: schedulers, supervisors, and mappers. In technical terms, properties such as maximization factor and other constraints, are represented as bounded memory and CPU measurements.In general, when several CPU cores are present in system, distributing execution of an application across all cores is a trivial exercise. In addition, with memory being an often scarce commodity, and lagging in speed behind CPU during data-intensive operations, there is a likelihood that lack of availability of the former will constitute a performance bottleneck; therefore,

the discussion of resource profiles will lean more towards memory capacity, appropriation and utilization.

4.3.1.1 Qualifying Lean MapReduce Nodes

Lean alternatives A and B nodes must meet some functional requirements. The following is a brief discussion of some of the requirements:

a Mapper Nodes:

The mappers apply map task on their local data. Such data processing operations are typically resource intensive and therefore require that sufficient resources be allocated for this purpose. So as not to depart from the thesis objective of optimal and efficient resource utilization, a balance must be struck between the desire to appropriate a lot of resources for use by a mapper (to ensure it is performant) and limiting resource waste (maintaining a tiny resource footprint) while taking cognizance of available resource capacity.

Mappers are technically a combination of IncludeOS and a map task, hence the presence of a map() function confers the mapper role to an IncludeOS instance. The lightweight nature of IncludeOS means that the size of the OS kernel and device drivers is of little concern, rather the run-time nature of the guest application and the size of the target data are critical factors. In its current state of development, IncludeOS relies on virtual memory for all storage needs and presents no interface for storage volumes. For this reason, when used in a data processing context, the amount of data that can be served to one IncludeOS instance is limited relative to a virtual server (running a general operating systems) on which volumes can be mounted. This however provides the convenience of leveraging the speed of access of RAM, and gives an opportunity to drive a good degree of processing modularity, as a large data set can be split among a good number of mappers. The degree of modularity is only limited by the available memory capacity on the hypervisor (see **??** below).

Theoretically, the memory profile of an IncludeOS instance should be characterised by its available memory capacity (free memory) which should be more than three times the size of the data which will be processed by the encapsulated service or application. Keeping in mind that all data will reside in memory, this makes adequate room for variables declaration and associated memory allocation when the data is parsed. This theory is supported by results of trivial tests which show that memory is exhausted at some point leading to premature termination of execution, if the memory is sized less than thrice the input data.

The performance of an IncludeOS-resident service degrades as its workload, in terms of data size, increases. Given that IncludeOS is not designed for robust operations, and to take advantage of a good degree of parallel execution, the size of input data will be kept below 100 megabytes. This implies that for the largest possible input data, the memory size of an IncludeOS instance

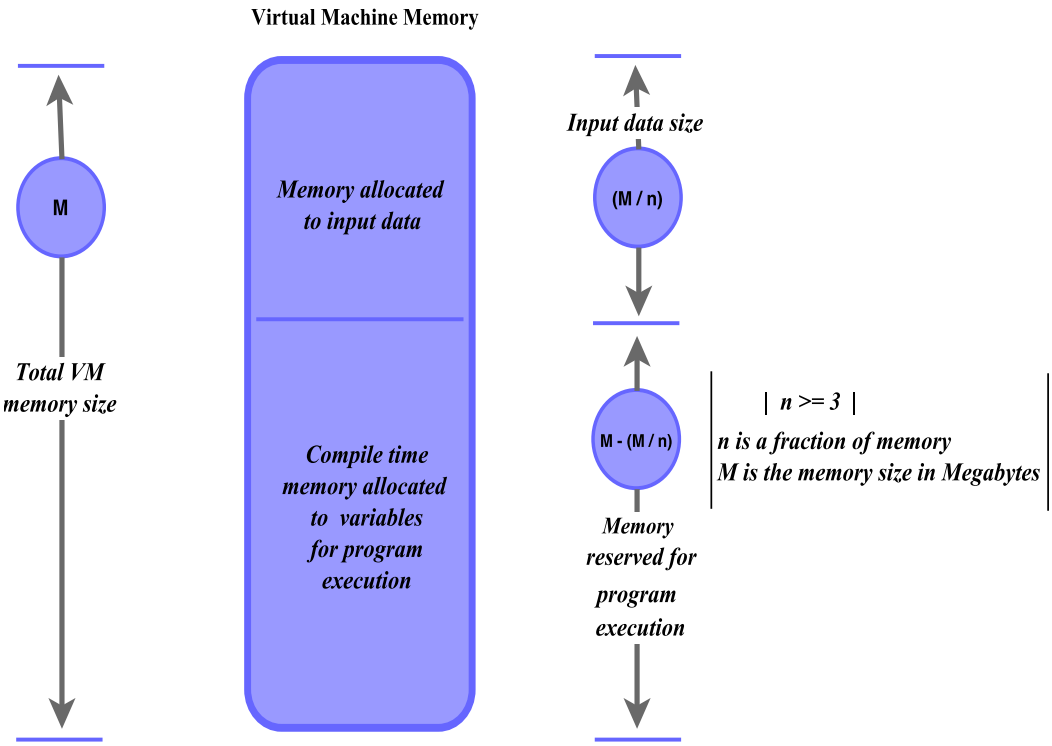

Figure 4.5: The memory size of an IncludeOS instance should be at least three times the size of the input data

will be at least 300MB. Using the same instance as an example, for resource efficiency to be satisfied the memory size should be configured as a value within a small neighborhood of 300MB.

By default an IncludeOS instance is assigned one virtual CPU core. Considering that IncludeOS is single-threaded, adjusting this to a higher value in order to derive more computing power will not lead to a performance boost,hence the default value for the CPU component will be retained.

IncludeOS is designed to only support native applications written in C++, so the map() function shall be written this progamming language. The function counts the occurrence of words in input data and builds an index in the form of a key-value pair. It formats the key-value pairs as a json like string, which it emits as its result. It must be noted that the input data is injected into a mapper when an application is compiled into an IncludeOS disk image.

b Supervisor Nodes:

The supervisor role is conferred to a node by the presence of a reduce() function which is responsible for aggregating the output of a set of local mapper nodes. It is usually created out of necessity to extend a cluster and offload some workload from a scheduler or an upstream supervisor (in the case of Lean alternative A). It provides the virtualization layer which supports some mapper nodes, and also acts as a hold of input data destined

for those mapper nodes; it is responsible for splitting up a sub-portion of a text corpus and dispensing the resulting data chunks. This is depicted in fig 4.6

The process of splitting data takes up a fair amount of memory, as the totality of data to be split and some of the splits will reside in memory at any one time. To aggregate resources the supervisor - acting in the capacity of a reducer - receives semi-processed data into memory buffers from the mappers, processes the intermediate data, and emits the result of that activity. Given that activities of the reduce task and overlying mapper nodes are compute and data intensive, it is vital that a supervisor node be made robust enough to support such activities.

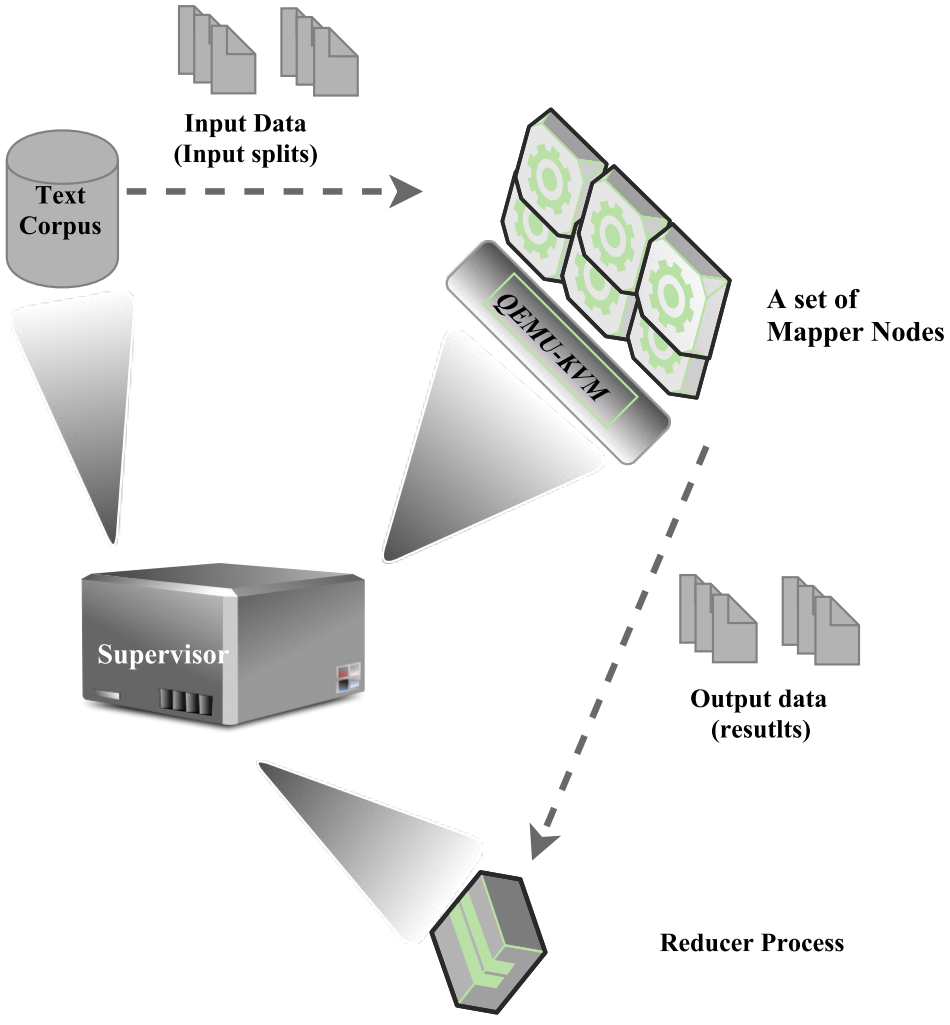

Figure 4.6: A supervisor provides for the resource needs of the overlying mapper nodes and the resident reduce() function

Due to the complex of system events needed to satisfy its function, a supervisor node will run a diversity of processes. To handle such complexity, a general purpose OS - in this case, Ubuntu 14.04 - is be better suited than a

Unikernel OS. By virtue of the OS being free and open-source, the system stands to benefit from a rich pool of open-source tools essential for big data processing, and also sidestep copyright issues which may arise from the use of proprietary operating systems and tools. To properly support the host operating system and its delegated tasks, an Openstack xlarge flavor is a good fit.

Incorporating a general purpose operating system enables an environment in which applications written in a variety of languages are well supported. At the heart of a supervisor is a tcp server (implemented in Python) at which a supervisor receives results streamed to it from its overlying mappers. The supervisor expects the reports from such nodes to be in json-like format and maintains this structure when sending on the outcome of its reduce() operation to the next upstream worker.

c Scheduler Node:

A scheduler is where all data-processing related activities begin and end for each job processing window. It holds the full mass of data meant for a job window and may process all of it (this depends on the Lean alternative under consideration) if resource constraints are not exceeded. A scheduler may host a set of mappers while extending the cluster if need be (Lean alternative A) or outrightly relieve itself of such hosting responsibilities by spawning and activating supervisors (Lean alternative B). Decisions on parameters which will be used across the cluster, such as maximum data block size (a mappers input data size, and ideally a third of the task slot size) are made here. Put more succinctly, the scheduler is the sole component that defines the behavior of the Lean alternatives and distinguishes one from the other, without necessarily controlling the other nodes.

One of the ideas behind the Lean MapReduce architecture is optimal utility of available resource capacity in a data center, or through the dynamic and adhoc provisioning of compute resources on a cloud platform. A scheduler is key to realising this goal. A scheduler may reside on a physical server beside other application which, due to the nature of their activity, may leave the system's resources idle for a considerable period of time. The scheduler can potentially fill out the unused capacity with some big data workload, and then extend its cluster based on defined constraints and rules which govern the operation of the Lean alternative in use. For context, the Lean MapReduce framework will be implemented in an environment bereft of pre-existing applications.

Much like the supervisor, a scheduler may host (or may not host - Lean alternative B) a set of mappers and, mandatorily, a reduce task. Its resource needs are similar to a supervisor's; however, it deals with data on a larger scale both in the form of unprocessed text corpus and the intermediate or semi-processed data submitted to it by downstream nodes for finalization. Notwithstanding the foregoing, its activities are sequenced such that its scope of operations doesn't outstrip that of a supervisor by much. For this

reason, it will also be given an Openstack xlarge flavor.
The code base of the scheduler, including the reduce() function and a tcp server socket, is written in Python. The tcp server socket receives results from all the scheduler's child nodes or downstream workers. The inbound data from downstream workers is typically a json-formatted string, while the final result emitted by the scheduler is a list of words and their corresponding count.

4.3.2 Lean MapReduce Workflow and Schema: Alternative A

This version of the Lean MapReduce architecture is patterned after its abstract equivalent. It derives its structure from the manner in which a scheduler facilitates the processing of big data. It is assumed that the target data is readily available at the scheduler node. When a user request arrives for processing of a text corpus, the scheduler application is executed against the data. It evaluates the job based primarily on the size of the target data. The measured size is checked against available system resources to verify sufficiency; the resources requirement, particularly memory, is calculated using a simple formular which sizes memory in other to ensure optimal resource utilization by MapReduce-oriented processes, while also making provision for other system events - caching, execution of program processes auxiliary to the scheduler, and others - which are essential for an acceptable level of system performance. The formular is as follows:

$$allocated\_mem = unused\_mem - \left(unused\_mem * \frac{1}{fraction\_of\_mem\_to\_be\_reserved} \right)$$

If the job dimensions fall within the processing capacity of the scheduler, it splits the data uniformly according to a specified data block size (the final chunk might be smaller than the rest) - these chunks are referred to as input splits. For each input split, a mapper node is spawned; the mappers are distributed round-robin style across all available CPU cores and executed in parallel. A reduce() function is called in tandem to wait for results emitted by all mappers. The scheduler uses a count of the number of spawned mappers as a lax control parameter in determining if all mappers successfully completed theirs task, when collating the streamed intermediate results. If the expected number of results is not obtained, the reduce() function will carry on waiting and will block transition to the next processing phase.

If capacity threshold is crossed, a supervisor is spawned and the excess data at the scheduler, as well as the map() and reduce() code, is transferred to it over a secure file transfer channel. As illustrated in fig 4.7, the scheduler then proceeds with the process of executing the workload which it carved out for itself just as described in the previous paragraph. When listening for incoming results, the reduce task expects to receive one more result than the number of overlying map tasks in order to proceed with merging intermediate results. The extra result corresponds to the intermediate result expected from the new supervisor.

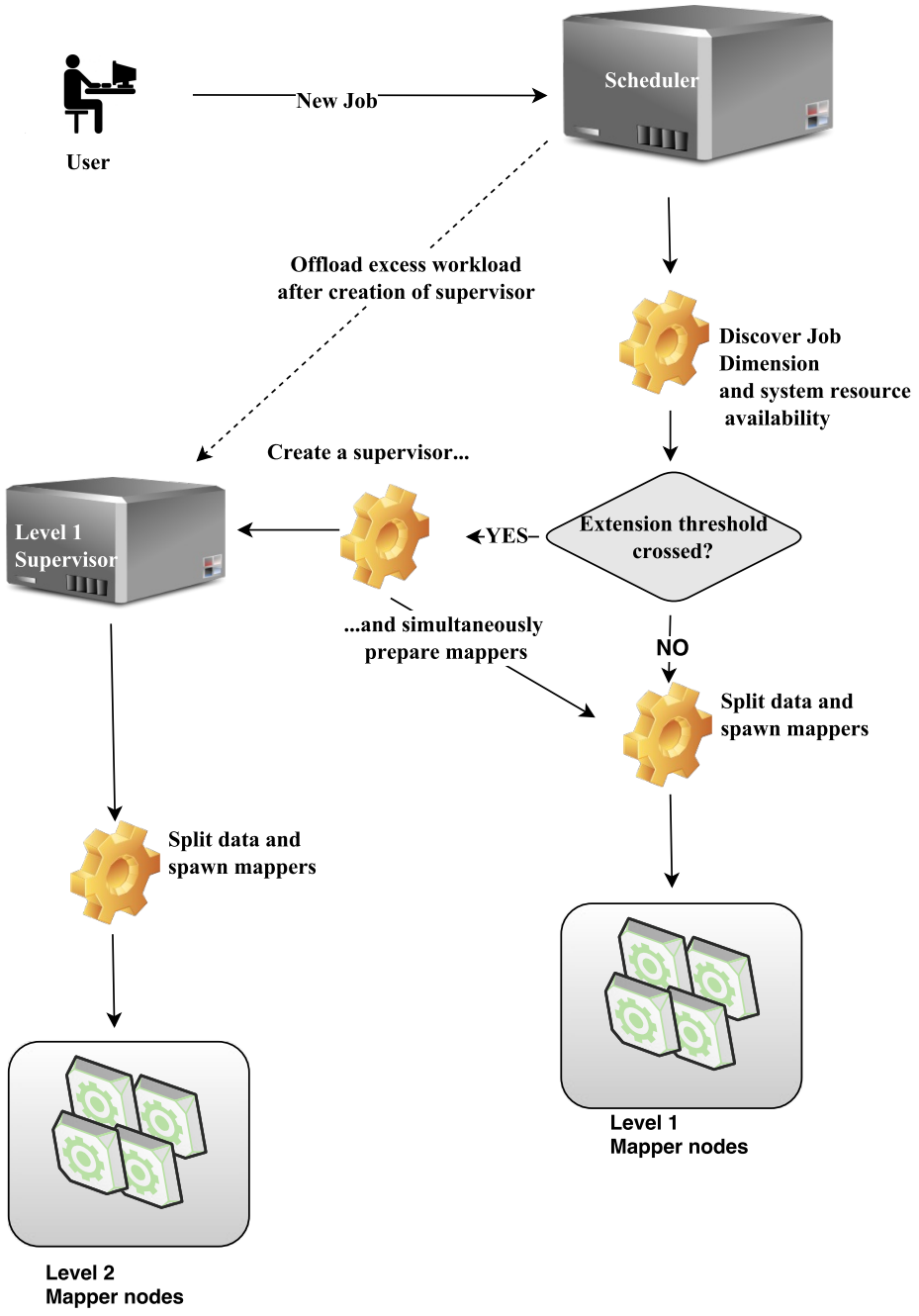

Figure 4.7: The scheduler may execute the job itself or extend the cluster based on resource constraints

A supervisor implements similar logic to that used at the scheduler in deciding whether or not to extend the cluster in other to accommodate the subset of the workload submitted to it (as can be seen in fig 4.8. The process of extending the cluster does not stall execution of the directly attached mappers as they act autonomously; they will run to completion and emit their intermediate results, while the reduce task they report to waits for results from them and the downstream supervisor.

The results of the task processing performed by mappers are aggregated by the reduce task on a reducer node, and where necessary passed on to the next upstream worker until it gets to the scheduler where processing is finalized and a final result submitted to the user.

4.3.3 Lean MapReduce Workflow and Schema: Alternative B

Lean alternative B is inspired by abstract alternative B. It is designed with the assumption that the data to be processed is large enough such that it is prudent to unburden a scheduler and shift processing of a job to a supervisor or distribute the load among a group of supervisors. The idea is that the scheduler can operate out of a server hosting other applications but which has some just enough capacity to host a reduce task and the cluster preparation utility. A server could also be dedicated exclusively for this purpose, but utilization considerations have to be made along cost-benefit lines: the cost facet has to do with the resource specification of such a server and the percentage of those resources that might go unused during active (when a Lean MapReduce cluster is up and running) and inactive periods; the benefit aspect points to how well job processing is facilitated by having such a server in place.

Lean alternative B starts out much like alternative A: a memory calculation is performed and the result is compared to the target data's size to see if a capacity thresholds is crossed. This Lean alternative is however different in how the processing framework is rolled out. A scheduler does not support mappers, and employs the notion of supervisor-oriented thresholds; it relies on its knowledge of the memory size of the proposed supervisors. The scheduler uses the premise that the supervisor nodes are entirely dedicated to the execution of reduce tasks and hosting of mappers, as well as a few auxiliary processes. Employing this logic, the scheduler can establish the number of supervisors that have to be spawned to successfully complete a job. Data is split according to this number and a supervisor is created for each split. A portion of data, a suite of tools - including code for map and reduce tasks - and processing parameters - including data block size - are then fed to each supervisor. This means that only the logic for deciding the size and shape of a cluster is concentrated at the scheduler (see fig.4.9 for illustration).

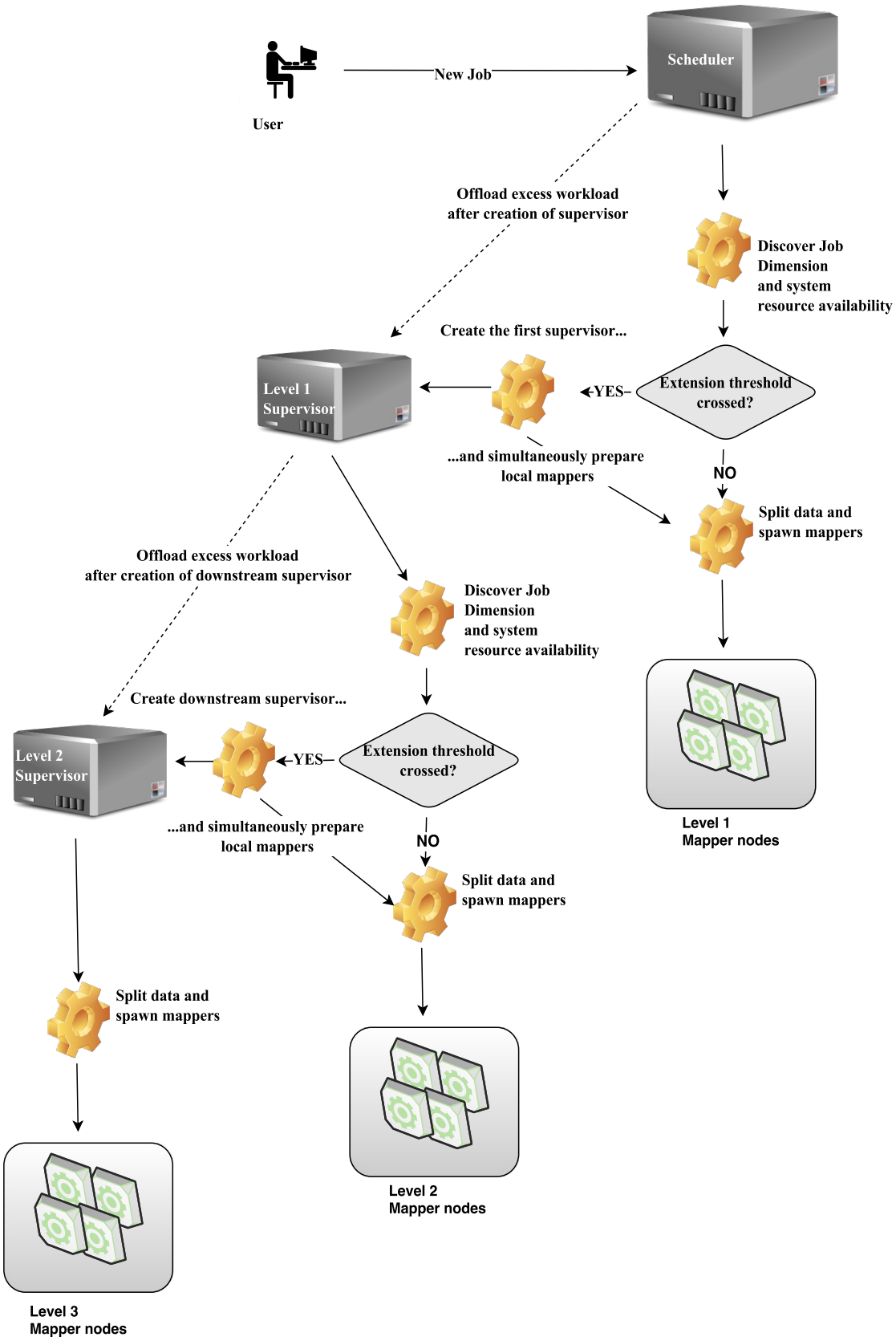

Figure 4.8: A supervisor employs the logic used by the scheduler in determining how to treat the workload received

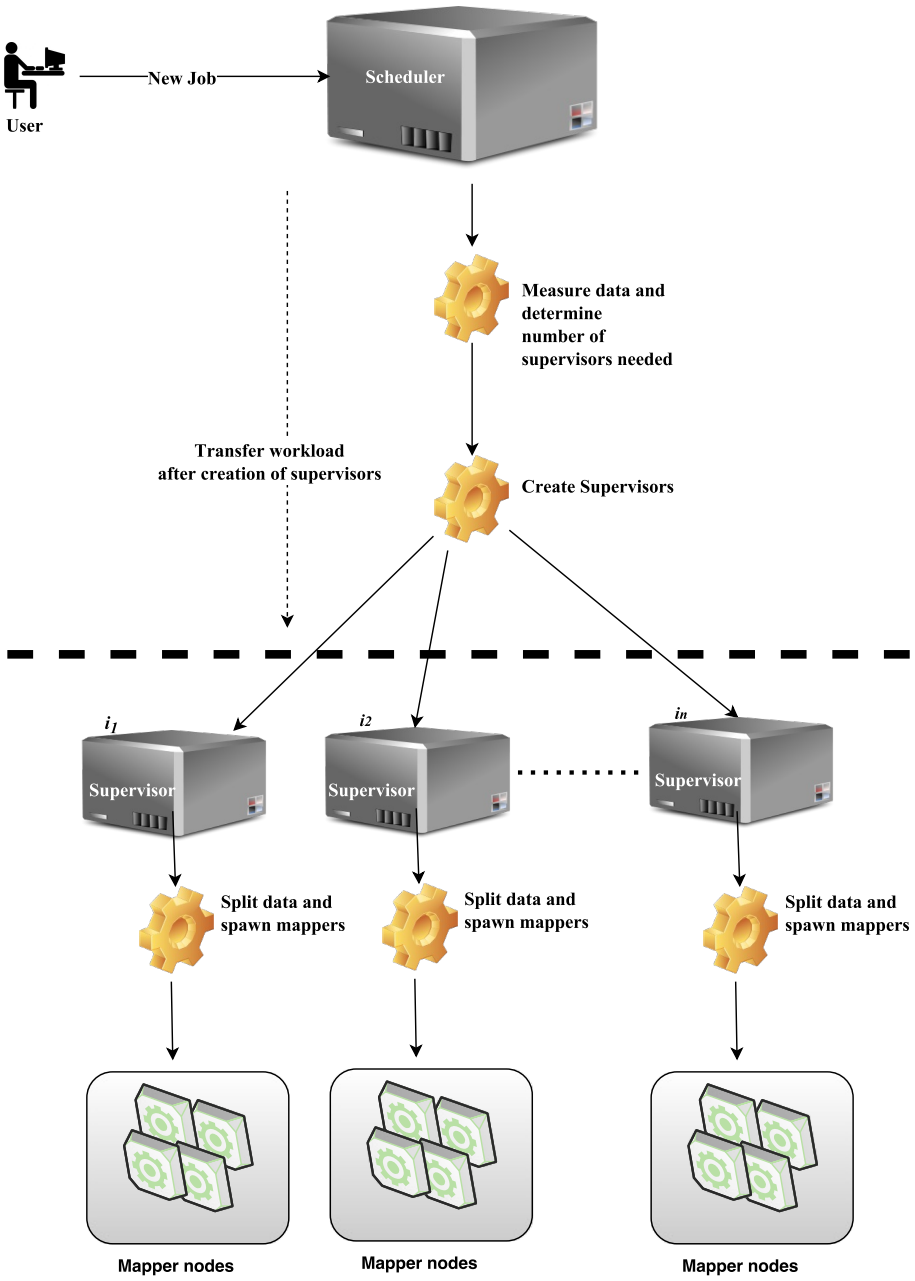

Figure 4.9: Schedulers communicate directly with supervisors and supervisors communicate directly with mappers

The pre-processing activities of a scheduler is made trivial due to the work done by the scheduler. A supervisor only has to determine how many mappers it can support. It derives this number by dividing the size of data it is responsible for by the data block size parameter submitted by the scheduler.

$$number\_of\_mappers = \frac{data\_size}{data\_block\_size}$$

The result of this cluster strategy is a structure where the schedulers communicate directly with supervisors that in turn communicate directly with mappers, over network links. For forwarding data and application files to supervisors the scheduler employs file transfer protocol over secure communication channels, while processed data, flowing in the inverse direction, are sent over a plain TCP connection.

4.3.4 System Reliability through Fault-Tolerance Features

The Lean MapReduce architectures described so far are vulnerable to issues which may arise due to failures of at least a single node, either a mapper or supervisor; this could stymie the entire system workflow. Owing to their characteristics, there is a greater likelihood that a mapper will constitute a bottleneck than a supervisor, since measures have already been taken to ensure that the latter is never overloaded. This calls for a fault-tolerant edge to be built into the alternate architectures. The subtle difference of the nature of the Lean alternatives implies that different approaches have to be employed for this purpose:

4.3.4.1 Lean Alternative A with a Fault-tolerance layer

The Lean alternative A architecture features a hierarchy of mapper and supervisors, with each entity type recurring at various levels. The potential points of failure are at the mapper level and the supervisor level. The fault-tolerant edge will involve a replication of workload at the supervisor level. To ensure the least possible resource overhead, the degree of redundancy is fixed at 2. Data splits will be prepared in duplicates and a copy given to each of a redundant pair of supervisors. If one supervisor fails, the results from the other is utilized. When both of them report results, the first result to arrive may be used and the second discarded, or they may be compared as an accuracy check.

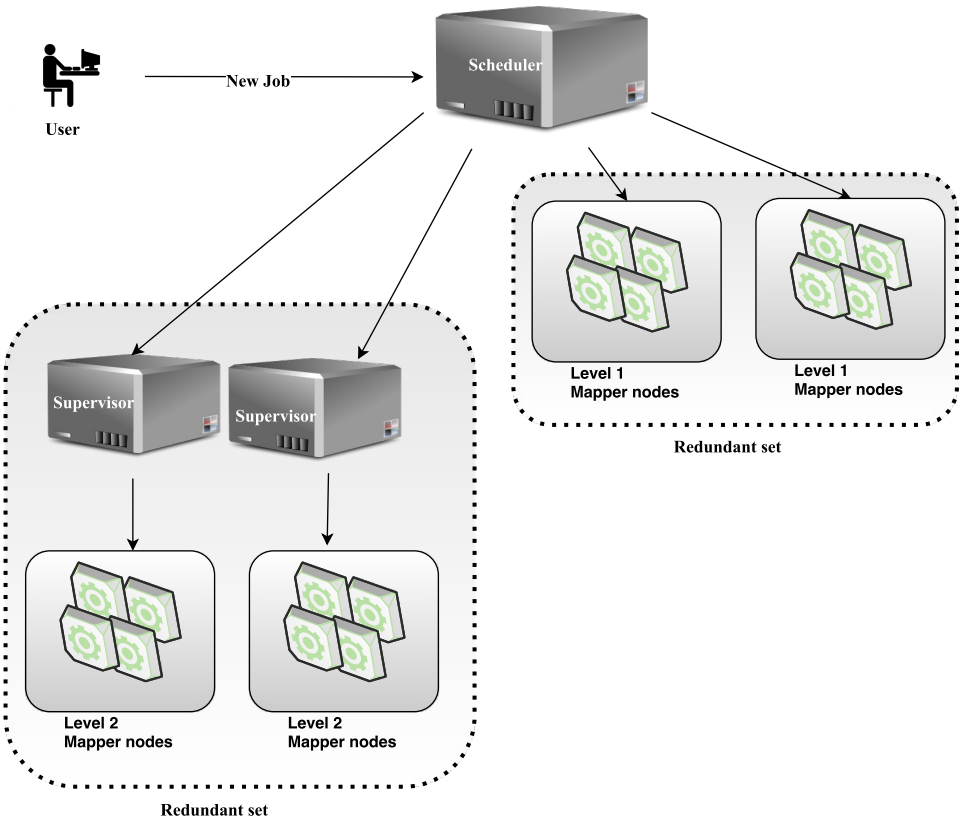

Figure 4.10: Redundant pairs are used to reduce the probability of job failure

For its local mappers, the scheduler will create an active replica of each. This approach is adopted since a cluster cannot have more than one scheduler. With this built-in layer, Lean alternative A imposes a resource overhead almost twice that of its non-fault-tolerant version because the scheduler and each supervisor will host half the initial number of distinct mappers, while the height of the nodes will double. This strategy is not immune to failure as there is still exists the probability that replicated pairs could fail in a job processing window.

4.3.4.2 Lean Alternative B with a Fault-tolerant layer

The layout of Lean Alternative B presents an opportunity to integrate a message oriented middleware (MOM); a construct which eliminates the need for existence of active replicas. The message queuing service may be installed on the same node as the scheduler application or on a different virtual server. Whatever the option, a message broker - the core component of a MOM - will be the logical interface for communication between supervisors and the scheduler.

The scheduler assumes the role of a Producer in this approach, while the supervisors are the consumers.The scheduler carries out the preprocessing exercise described in 4.3.3: it creates the necessary number of supervisors and transfers the designated workload to each of them. In addition, the supervisors are connected to the queuing service to await messages from the scheduler. To trigger execution, the scheduler connects to a queue hosted by the message broker and inserts a message,

one for each supervisor in the cluster, containing the task processing parameters and task execution signals. The messages contain sender and receiver information. The message broker stores each message until they are taken off the queue by the designated consumer. When the message broker resides on a separate virtual server, producers and consumers use TCP connections to communicate with it

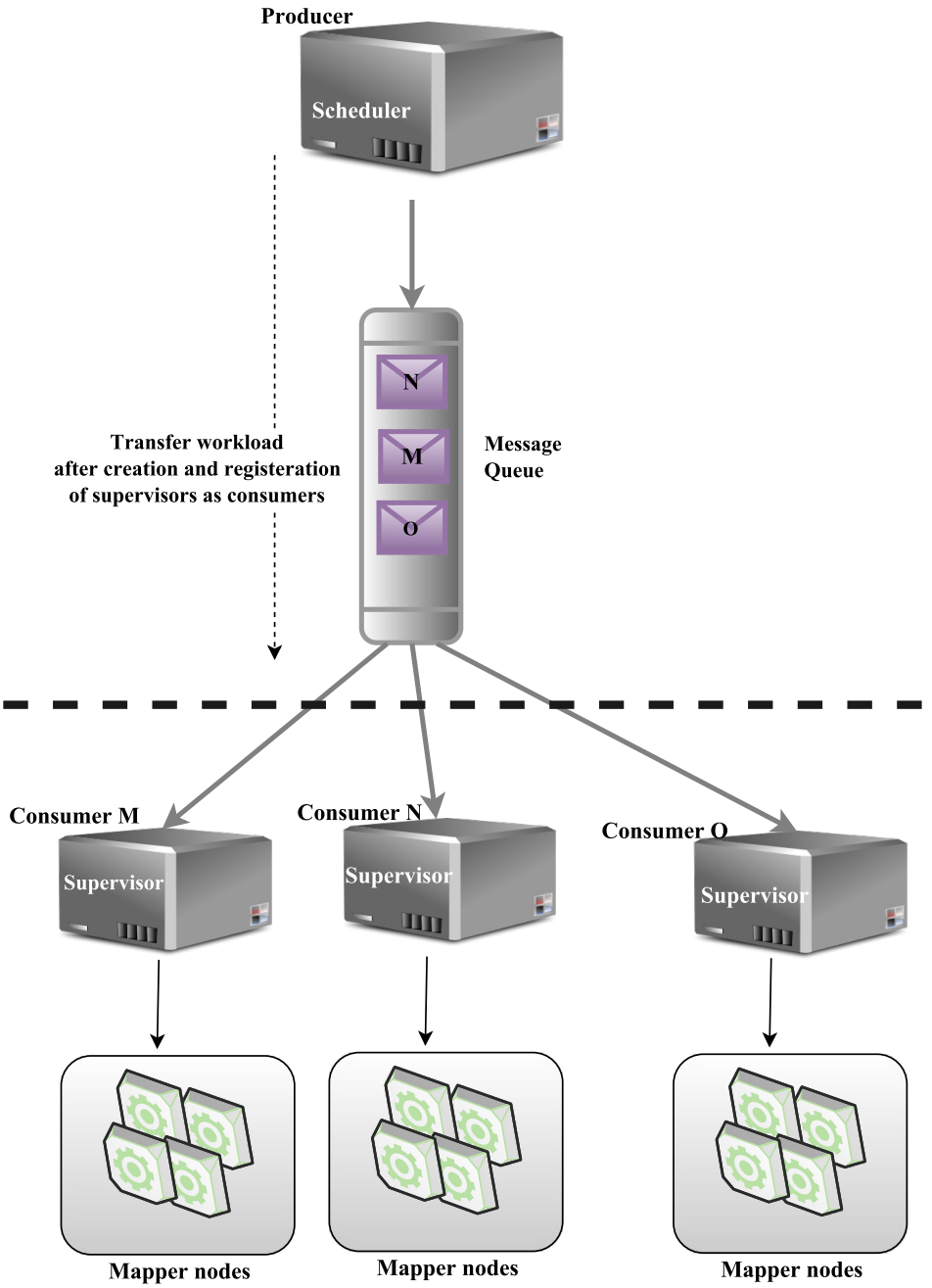

Figure 4.11: Message queuing presents an interface for reliably managing job execution

Conversely, when results are ready to be delivered, the supervisors assume the role

of a producer and the scheduler becomes the consumer. As illustrated in fig 4.12, actual results are not sent through the queue, instead "ready" messages are pushed into the queue by each supervisor - to indicate that task processing was successful - before they establish a tcp connection to deliver their results to the reduceask at the scheduler. The *out-of-band* TCP connection is needed since the the size of the results may be greater than the size limit placed on messages bound for a message broker's queue.

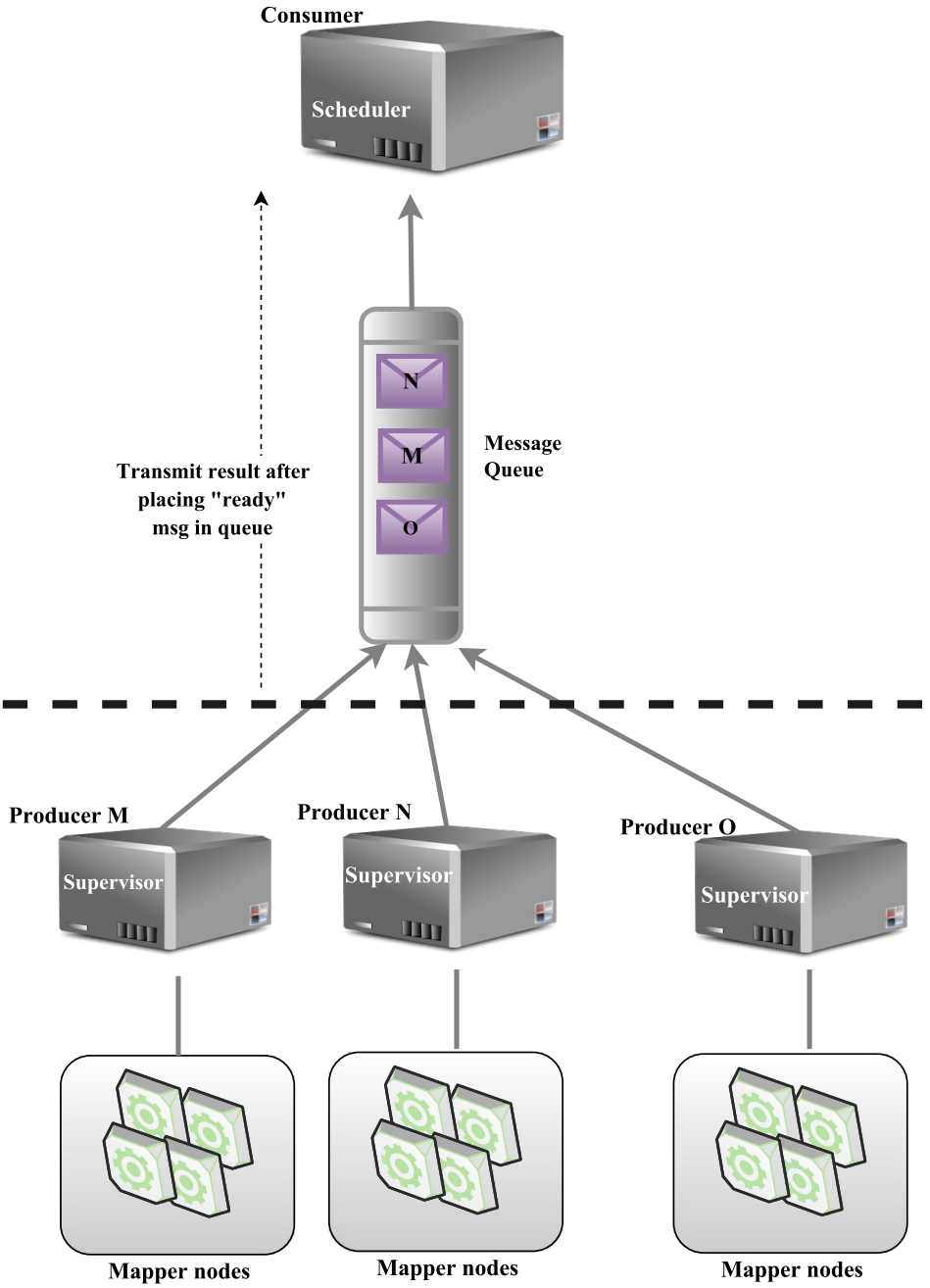

Figure 4.12: To track the state of individual task execution, each supervisor places a "ready" message in queue to indicate successful execution

The scheduler maintains a count of the number of supervisors in its cluster and their IP addresses (it builds a array of IPs). Using this information, it knows that the number of messages it should consume must tally with the supervisor count. For each message consumed, it removes the associated IP from an array of IPs until there is non left. If a message is not placed on the queue by a supervisor or set of supervisors within a specified time window (which starts counting from the arrival time of the first message) which is adjusted to accommodate tolerable delays, the scheduler creates another supervisor and transmits a message to it to commence execution. It is of no consequence if the problematic supervisor submits it result set before the new node since the scheduler expects a specified number of results, and will stop taking messages from the queue once this number is met.

4.3.4.3 Lean MapReduce Algorithms

In this subsection the, pseudo-code which will serve as the blueprint for creating component parts of the prototypes for Lean alternative A and B, are developed. Some of the functions, especially the map() and reduce() functions, recur in the alternatives.

4.3.4.4 The Logical Units of Lean Alternative A

The primary units of logic which will interplay to manifest a working Lean alternative A prototype are named as follows: init_cluster(), reduce(), map(), spawn_mapper(), and forge_supervisor(). The following is a brief discussion of what they do:

1. Init_cluster()

   This unit is triggered when a scheduler program in launched; ideally, it is the logical equivalent of a constructor function. It also kick-starts execution at a new supervisor when the cluster is extended. The unit assesses data size and memory availability, and follows hard-coded rules for determining if a supervisor is needed to complete a job. It marshals the other functions where the need be in performing a requisite function at a predefined point in the execution flow. See algorithm 1

2. reduce()

   The reduce() unit is responsible for aggregating result from mappers at a supervisor, and from both supervisors and mappers at the scheduler. It expects all incoming data in a json-like format (a list of key-value pairs). The result of "reduction" will be written to a file on disk if the current node is a scheduler, or forwarded over a tcp connection to the next upstream worker if the current node is a supervisor. See algorithm 2.

3. map()

   Map() comprises the logical set of steps used by mappers to build an index of words in the portion of the text corpus that they have been assigned. After processing data, a mapper emits the data towards the waiting reduce() function (created out of the create() logical unit) on its supervisor. The high-level steps are outlined in algorithm 3.

Define Init_cluster(*data, data_block_size***)**
{

data_size = get_size(data)
available_memory = get_size(memory)

$$\text{allocatable} = \left(\text{available\_memory} - \frac{(\text{available\_memory}) * 3}{n}\right)$$

// n is the integer denominator of the fraction of memory
// ... to be reserved for other use
// 3 signifies that each mapper requires memory at least 3
// ...times the size of data it must process

greater = False
no_of_mappers = 0

// "greater" serves as a control variable for cluster extension
// Next: Check to see if cluster extension is needed

if *allocatable* > *data_size* **then**
{

$$\text{no\_of\_mappers} = \frac{\text{data\_size}}{\text{data\_block\_size}}$$

}
else
{

greater = True excess_data = data_size - allocatable

$$\text{no\_of\_mappers} = \frac{\text{allocatable}}{\text{data\_block\_size}}$$

}
input_splits[] = split_data(data, no_of_mappers)

for each *split* in *input_splits[]* **do**
{
// map() function is compiled into a mapper node
// ...along with a portion of data
spawn_mapper(split, map())
}
if *greater* == *True* **then**
{
// create a supervisor and start up reduce() function
create_supervisor(excess_data, data_block_size, init_cluster(),
reducer(), spawn_mapper(), map(), create_supervisor())
run_reduce = reduce(no_of_mappers + 1)
// "+1" factors in result from the supervisor
}
else
{
// start up a reduce()
run_reduce = reduce(no_of_mappers)
}
}

Algorithm 1: This logical unit initializes the cluster at the scheduler and supervisor levels

```
Define reduce(no_of_mappers)
{
    key-value = { }
    while n < no_of_mappers do
    {
        repeat
        {
            data = listen_for_connections.recv()
            for each word, value in data do
            {
                if word in key-value then
                {
                    key-value[word] += value
                }
                else
                {
                    key-value[word] = value
                }
            }
            n + 1
        }
        until n = no_of_mappers;
    }
    if current_node==scheduler then
    {
        for each word, value in key-value do
        {
            write_to_file(key, ":", value, "")
        }
    }
    else if current_node==supervisor then
    {
        send_to_upstream_node(key-value)
    }
}
```

Algorithm 2: The reduce() unit aggregates results from downstream workers

```
Define map(input_split)
{
    key-value = { }
    data = input_split
    for each word in data do
    {
        if word in key-value then
        {
            key-value[word] += 1
        }
        else
        {
            key-value[word] = 1
        }
    }
    output = format_json(key_value) emit(output)
}
```

Algorithm 3: The map() function builds an index of words in its input split

4. spawn_mapper()

   This unit is used to spawn a mapper for a particular data split. It is called repeatedly by either a supervisor or a scheduler until the higher limit of the no_of_mappers count is met. In involves compiling an application into an IncludeOS disk image and starting the instance to process an portion of data. The steps for his are outlined in algorithm 4

```
Define spawn_mapper(input_split, map())
{
    IncludeOS = compile_IncludeOS()
    service = map(input_split)
    mapper = compile_service(IncludeOS, service)
    return run(mapper)
}
```

Algorithm 4: Spawn_mapper() unit spawns a mapper for every input split passed to it

5. create_supervisor()

   Create_supervisor() is responsible for adding new nodes as supervisors to the cluster, when called. It ideally iterates the entire workflow just as scheduler would when init_cluster() runs. The function relies on an MLN template to describe the specification for creating an Openstack virtual machine. When the supervisor is deemed to be reachable and accessible, all the material (all the other functions plus data) it requires for process its portion of the total workload is transferred to it

```
Define create_supervisor(excess_data, data_block_size, init_cluster(), reduce(),
  spawn_mapper(), map(), create_supervisor())
{
    new_vm_spec = build_template(mln_template)
    vm = start(mln_project)
    IP = scrape_ip(vm)
    send_workload_to_IP(excess_data, data_block_size, init_cluster(), reducer(),
      spawn_mapper(), map(), create_supervisor())
}
```

Algorithm 5: Create_supervisor builds a new Openstack virtual machine for extending the cluster

4.3.4.5 The Logical Units of Lean Alternative B

Lean alternative B has the map(), reduce(), and spawn_mapper() functions in common with Lean alternative A, hence these will not be treated beyond the discussion under Lean alternative A; however, the steps for the init_cluster() and create_supervisor() differ slightly from those of its counterpart. These two functions are outlined below:

1. Init_cluster()

   This variant of the constructor unit predetermines the size of the cluster needed. It performs no functions which directly affect mappers. Its two core

functions are to create all intermediate workers (supervisors) and to finalize all results. To facilitate this process, information about the Openstack flavor is used. "Sizing" calculations are performed to ensure that most of the systems resources (memory) are utilized and at the same time ensuring that the system is no deprived of sufficient amount of memory to maintain an acceptable level of operability. Algorithm 6 shows the logical steps:

Define Init_cluster(*data, data_block_size*)
{
data_size = get_size(data)
std_memory_size = get_size(openstack_flavor)

$$\text{allocatable} = \left(\text{std\_memory\_size} - \frac{(\text{std\_memory\_size}) * 3}{n}\right)$$

// n is the integer denominator of the fraction of memory
// ... to be reserved for other use
// 3 signifies that each mapper requires memory at least 3
// ...times the size of data it must process

// Determine the number of supervisors needed

$$\text{no\_of\_supervisors} = \frac{\text{data\_size}}{\text{allocatable}}$$

data_segments[] = split_data(data, no_of_supervisors)

for each *n* in *data_segments[]* **do**
{
// create a supervisor and send it needed material
create_supervisor(data_segments[n], data_block_size,
reducer(), spawn_mapper(), map())
}
}

Algorithm 6: Lean Alternative B: This function initializes the Lean alternative B cluster at the scheduler node

2. create_supervisor()

   The process of creating a new supervisor is straight-forward. A supervisor is created and given all the material it needs to evenly distribute its workload across its local cluster of mappers, a number of which it will have to spawn based on data block size constraint and the size of the portion of data it must process. It adds an extra bit of processing over its Lean alternative A equivalent by calling the splinter() function (see next item). The flow is captured in algorithm 7

3. splinter()

   Splinter is a data splitting utility passed on to a supervisor by a scheduler to guarantee consistency in workload partitioning rules. It takes some data and splits it into multiple chunks, one for each mapper to be created. It also has the logic for spawning the required number of mapper nodes. Algorithm 8 gives the high-level steps for this utility.

```
Define create_supervisor(data, data_block_size, reduce(), spawn_mapper(), map(),
  splinter())
{
    new_vm_spec = build_template(mln_template)
    vm = start(mln_project)
    IP = scrape_ip(vm)
    send_workload_to_IP(data, data_block_size, reducer(), spawn_mapper(), map(),
      splinter())
}
```

Algorithm 7: Lean Alternative B: Create_supervisor builds a new Openstack virtual machine for extending the cluster

```
Define splinter(data, data_block_size)
{
                                    no_of_mappers = data_size / data_block_size
            input_splits[] = split_data(data, no_of_mappers)

            for each n in input_splits[] do
            {
                mapper=spawn_mapper(input_splits[n], map())
            }
}
```

Algorithm 8: The splinter function splits data, distributes the splits among a group of spawned mappers

4.3.5 Estimating The Computational Complexity of Lean Alternatives A and B

The depth and arrangement of Lean alternatives A and B suggests that there are some factors which may influence their system resource utilization and Time to Complete (TTC). Theoretically, these components of efficiency can be qualified for each of the Lean alternatives by determining the asymptotic complexities of their algorithms; this will help in setting them apart for further analysis. The goal here is to get a sense of the reasonable worst case estimate using $\mathcal{O}$ notation. Working with the assumption that Big $\mathcal{O}$ will not help in providing a good estimate of CPU and memory measurements, the emphasis will be on the time efficiency component.

Five algorithms were described for Lean alternative A and six (including those already discussed under alternative A) for alternative B. Looking at alternative A, let the algorithms be referred to as f_1 (init_cluster()), f_2 (reduce()), f_3 (map()), f_4 (spawn_mapper), and f_5 (create_supervisors). Given that f_2 to f_5 are the core worker functions which are consumed by f_1, they are more critical factors than the latter.

Some assumptions are necessary for this discussion: let n, a constant, be the maximum number of task slots (mappers) which can be hosted by a supervisor, and d the size of the data mass to be processed by all mappers. To induce a worst case attribute, let d tend to infinity. By implication the total number of supervisors, m, also tends to infinity and is deduced as the inequality:

$$m \leq \frac{d}{n} \quad where \lim_{m \to \infty}, \lim_{d \to \infty}$$

f_2 features nested iteration, 2 levels deep (a for-loop within a while-loop); it also features another for-loop. Two arrays of marginal order, k are iterated at each level of the nested construct. This can be approximated as:

$$f_2 = k^2 + k$$

with an approximate worst case:

$$f_2 \in \mathcal{O}(k^2)$$

It must be noted that the marginality of k's degree makes time to process input data approximately the same for all possible iterations; therefore its estimate becomes $f_2 \in \mathcal{O}(1)$, which makes it of less significance than the other factors.

The chained nature of alternative A means that creating new supervisors happens in a perfectly sequential order; the same holds for the different levels of mappers. Since the number of mappers for a supervisor is constant, and given that creation is near-parallel, the time required to spawn many mappers is approximate to the time required to spawn one, hence $f_4 \in \mathcal{O}(1)$. The time it takes to get all supervisors and mappers in place increases linearly as the number of supervisors increase. This can be expressed as:

$$f_5 = m + 1$$

Which approximates to:

$$f_5 \in \mathcal{O}(m)$$

Time to process data is a constant value - with the assumption of parallel execution of the map() functions and static data block size - where $f_3 \in \mathcal{O}(1)$. In reality, when the order of m is a small value ($m \to 1$), then the approximate worst-case of f_3 is greater than that of f_5, however the inverse is the case as $\lim_{m \to \infty}$, which implies the following relationship:

$$f_3 \in \mathcal{O}(f_5)$$

Going by the foregoing, it is apparent that the f_5 induces the dominant time factor for alternative A. Time efficiency for alternative A can therefore be thus stated:

$$\mathcal{O}(f_5)$$

All approximations discussed so far hold true for alternative B except for f_5, the splinter algorithm - which may be referred to as f_6 - and the relationship between f_3 and f_5. f_6 is assumed to be a fast operation; splitting-time scales linearly with size but is a negligible factor. To differentiate f_5 and f_3 for alternatives A and B, an extra marker "b" is added to the subscript for alternative B.

For f_{5b}, as $m \to \infty$, the time for creating all supervisors is approximate to the time required to create one. This implies $f_{5b} \in \mathcal{O}(1)$. It is safe here to assume that

$\mathcal{O}(f_{3a})$ dominates $\mathcal{O}(f_{5a})$, since it is likely that the time a map() function spends processing its block of data is greater than the time it takes to create all supervisors. It therefore follows that the time efficiency for alternative B tends to the following:

$$\mathcal{O}(f_{3a})$$

4.4 Monitoring and Cluster Cleanup

An essential operation which facilitates judgement of the relative performance of each Lean MapReduce alternative prototype is monitoring and data gathering. To obtain meaningful measurements, the process will be based on 3 key metrics: Time to Complete, percentage CPU and memory consumption. Going by the revelations in 4.3.5, it is also essential to track the amount of time it takes to prepare a cluster (create and configure supervisors).

To effectively measure job processing time, a monitoring tool will be run at the scheduler since it has the best perspective on start and finish times of a job. The same could be done at the other worker nodes, especially the supervisor, but their start/finish time perception is limited to the sub-set of the cluster directly downstream of them. The measurement tool is a loose coupling of independent lines of code which are positioned strategically in the program execution workflow to capture the start and finish times in different variables. The duration is arrived at by subtracting the start time from finish time. The result is then written to a file on disk. The file in question - one for each of the Lean alternative prototypes - is designated to hold a set of such measurements against the workload size, over several experiments. The following algorithm (algorithm 9) brings these parts together.

```
// at processing preamble ...
timer = data_size
start = get_timestamp()
.
.
.
isend = False
// The while loop is entered after all processing is triggered
while isend == False do
{
    if finalresult.txt exists then
    {
        finish = get_timestamp()
        interval = finish - start
        timer += "\tab" + interval
        open file (duration)
        file.write(timer)
    }
}
```

Algorithm 9: time_keeper commands capture the start and finish time of the entire job execution process

Reading percentage CPU and memory usage will take place at the hypervisor level - that is the supervisor and the scheduler (in the case of Lean alternative A). At each of these worker nodes, a custom script is called from within the main program as the first event in the preprocessing stage and stopped as the final event of the entire job execution process (after time_keeping components are exited). Since qemu emulates the CPU and memory for the overlying mappers, it is necessary to monitor the qemu child processes which support the spawned mappers. Same script will be used to read the system resource usage associated with the reduce() function. The script presents an abstraction which is adapted for a specific process ID. The abstraction is described by algorithm 10.

```
function resource(process_id)
{
        // executes as first event ...
        timer = 1 sec
        open file (resource_usage)
        while 1 do
        {
            CPU = read_cpu_usage(process_id, time)
            memory = read_memory_usage(process_id, time)
            entry = data + "\tab" + CPU + "\tab" + memory
            sleep(timer)
            file.write(entry)
        }
}
```

Algorithm 10: resource() collects the resource usage at supervisors and the scheduler

At the end of a job window, the cluster is taken down. A clean-up script is used to release all hardware resources used allocated to the cluster, back into the general resource pool. The cleanup operation can be conveniently performed centrally at the scheduler node. Relying on MLN's interface with the Openstack platform, all supervisors can be stopped with MLN stop and destroy commands. This in essence also destroys the IncludeOS instances (mappers) that they host. In the case of Lean alternative A where the scheduler hosts some mappers, killing the parent qemu process has a ripple effect on all its child processes. The script also stops and deletes utilities and objects which facilitated the job processing operation, such as the tcp server socket and data files.

Chapter 5

Implementation

This chapter gives a description of the work done in creating working protoypes by following the technical designs and algorithms laid out in the design chapter. The staging environment is an Openstack cloud installation (Alto Cloud).

5.1 Components of the Toolkit

A few orchestration scripts have been created to facilitate properly functioning prototypes, including those with with integrated map() or reduce() functions. The key components of the scripts used are highlighted below. Together with other components, some of which are listed under the toolkit, they aid the end-to-end flow of a Lean MapReduce job.

5.1.1 MLN Template

An MLN template serves as a "boilerplate" which can be used as is, or modified with a different hostname, projectname, or filename.

Listing 5.1: MLN template

```
1
2  global {
3          project leanmr
4                  }
5          host downstream1 {
6               openstack {
7               image Ubuntu14.04
8               flavor m1.xlarge
9               user_data {
10                         wget https :// apt.puppetlabs.com/puppetlabs-release-
                               trusty.deb
11                         dpkg -i puppetlabs-release-trusty.deb
12                         apt-get update
13                         apt-get install -y puppet augeas-tools
14                         puppet module install maestrodev-ssh_keygen
15                         wget http :// student.cs.hioa.no/~s237677/sshkey.pp
16                         puppet apply sshkey.pp
17                         wget 10.1.0.33/ id_rsa.pub
```

```
18                          cat id_rsa.pub >> /home/ubuntu/.ssh/authorized_keys
19                          apt-get install -y python-dev
20                          apt-get install -y python-paramiko
21                              apt-get install -y make
22                              apt-get install -y git
23                              apt-get install -y bridge-utils
24                              apt-get install -y apache2
25                          }
26                  }
27      network eth0 {
28          net nsa_master_net
29          }
30  }
```

This provides a file system template which installs the baseline set of software needed for each virtual machine (supervisors) in the Lean MapReduce cluster. The baseline tools include puppet, which is not used in this context for configuration management but as a convenience tool with a feature (maestrodev-ssh_keygen module) that aids setting up ssh access to the new server; python paramiko module: a powerful tool for interacting with and configuring remote machines over an ssh connection; make: a Unix tool for simplifying building a program executable from an arbitrary module which could be written in C, C++, Java, etc. In this case, it is used in compiling and building an IncludeOS OS, and services bound for it; bridge-utils package which contains the brctl utility, used in creating bridge devices and setting up networking for hosted IncludeOS instances; and apache2: used in this capacity as a trivial network delivery mechanism - instead of serving up web pages, it serves other files which were difficult to deliver over alternative channels.

5.1.2 Create supervisor

The MLN template described above is utilized by a function which is responsible for creating virtual machines in the Openstack cloud. It modifies the template with a set of user supplied parameters so as to avoid identity conflict within the immediate namespace, builds the virtual machine, and returns its IP address to the calling program. The function implements the create_supervisor algorithm discussed in the design chapter.

Listing 5.2: CreateSupervisor

```
1  def prepRemote(localip):
2          os.chdir("/home/ubuntu/mln")
3          try:
4                  subprocess.check_call("rm␣/root/.ssh/known_hosts", shell=True)
5          except:
6                  print "no␣known_hosts␣file"
7          subprocess.call(["cp", "leanmr.mln", "leanmrtemp.mln"])
8          subprocess.call(["sed", "-i", "s/ipaddress/%s/g" % localip, "
               leanmrtemp.mln"])
9          subprocess.call(["sed", "-i", "s/leanmr/leanmr2/g", "leanmrtemp.mln"
               ])
10         subprocess.call(["sed", "-i", "s/downstream1/downstream2/g", "
               leanmrtemp.mln"])
```

```
11          fnull = open(os.devnull, 'w')
12          subprocess.call (["/usr/local/bin/mln", "build","-r","-f", "leanmrtemp
                .mln"],stdout=fnull, stderr=subprocess.STDOUT, close_fds=True)
13          subprocess.call (["/usr/local/bin/mln", "start", "-p", "leanmr2"],
                stdout=fnull, stderr=subprocess.STDOUT, close_fds=True)
14          status = False
15          ipaddress = None
16          reg_pattern = re.compile(r"\b\d {1,3}\.\ d {1,3}\.\ d {1,3}\.\ d{1,3}\b")
17          while status==False:
18                  mln_stat = subprocess.Popen(["/usr/local/bin/mln", "status", "
                        -p", "leanmr2", "-h", "downstream2"], stdout=subprocess.
                        PIPE)
19                  entry, error = mln_stat.communicate()
20                  ipaddress = reg_pattern.findall (entry)
21                  for i in entry.split ():
22                          if "downstream2" in i and ipaddress:
23                                  ipaddress = ipaddress[0]
24                                  ipaddress = ipaddress.strip ()
25                                  print "Address found! >> ", ipaddress
26                                  status = True
27                          else:
28                                  continue
29          return ipaddress
```

5.1.3 Run remote

When supervisors are created and work material sent to them, processing has to be triggered on them over an ssh connection. The python paramiko module is used for this purpose. The main orchestrator script which binds all functions into an end-to-end workflow is executed via this means. The ssh connections are based on ssh public-private key-pair and a username (Ubuntu) for a key-based login.

Listing 5.3: Run remote

```
1
2   import paramiko
3
4   def run(host, uname, pkey, command, other, port=None, localdir=None, remotedir
        =None):
5           key = paramiko.RSAKey.from_private_key_file(pkey)
6           ssh = paramiko.SSHClient()
7           ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
8           ssh.load_system_host_keys()
9           ssh.connect(host, username=uname, pkey=key)
10          if command == None:
11                  print "No command was supplied"
12          else:
13                  try:
14                          stdin, stdout, stderr = ssh.exec_command(command)
15                          if other != None:
16                                  time.sleep(2)
17                                  stdin.write(other)
18
```

```
19                  except:
20                          print "please cross-check the command you
                                attempted to execute", " \n <<<more info>>> \n\
                                n", stderr
21          ssh.close()
22          return
```

5.1.4 IncludeOS Binary

To use IncludeOS, a copy of the source code was cloned from its Github repository after which it was built into a program executable using the install_from_bundle.sh script which performs a fast build of the operating system. When a map() program (see below) is compiled with the IncludeOS executable, a runnable disk image is obtained. The compilation process is an aspect of the job processing flow.

5.1.5 Map()

The map() function is written in C++ because IncludeOS only supports native applications. The program parses a collection of words, which makeup the block of data given to an encapsulating mapper, and builds an index of the words. An elided version of the code is shown below:

Listing 5.4: Reduce()

```
1
2   #include <iostream>
3   #include <string>
4   #include <sstream>
5   #include <fstream>
6   #include <map>
7   #include <algorithm>
8   #include <os>
9   #include <net/inet4>
10
11
12  void Service::start() {
13          map<string, size_t> word_count;
14          string text = "bigfile";
15
16          istringstream single(text);
17          while (single)
18          {
19                  string word;
20                  single >> word;
21                  for (auto &x : word)
22                  {
23                          x = tolower(x);
24                          if (!isalpha(x))
25                          {
26                                  x='_';
27                          }
28                  }
29                  word = rtrim(word);
```

```
30                      ++word_count[word];
31
32  }
```

In IncludeOS parlance, any program which is ready for combination with the OS is referred to as a service. A file named service.cpp is used as a receptacle for this purpose.

5.1.6 Spawn Mappers

When an IncludesOS is compiled with a map() function and started, it is referred to as a mapper. The following code is a snippet of code which performs the task of spawning mappers:

Listing 5.5: Spawning Mappers

```
1   def spawn(count):
2           ips = 44
3           cpu = 0
4           for n in range(count+1):
5               if n == count:
6                   break
7               os.chdir("\%s/mapresult00\%d" \% (localdir,n))
8               subprocess.call(["sed", "-i", "/^$/d", "bigfile00\%d" \% n])
9               subprocess.call(["sed", "-i", r"s/$ /\/ g", "bigfile00\%d" \% n])
10              subprocess.check_call(["sed", "-i", r"$ s/\$ //g", "bigfile00\%d"
                    \% n])
11              subprocess.call("truncate --size=-1 bigfile00\%d" \% n, shell=
                    True)
12              subprocess.call(["perl", "-i", "-pe", "s/bigfile00\%d/`cat bigfile00
                    \%d`/ge" \% (n,n), "service.cpp"])
13              fnull = open(os.devnull, 'w')
14              subprocess.Popen(["make"], stdout=fnull,close_fds=True)
15              mac = randomMAC()
16
17              args = "taskset -c \%d /usr/bin/qemu-system-x86_64 --enable
                    -kvm -drive file=mapresult.img,format=raw,if=ide -device
                    virtio-net,netdev=net0,mac=\%s -netdev tap,id=net0,script=/
                    root/IncludeOS_install/etc/qemu-ifup -name includeOS\%d -
                    vga none -nographic -smp 1 -m 200" \% (mac,m,cpu)
18
19              args = shlex.split(args)
20              subprocess.Popen([args], stdout=fnull, close_fds=True)
21              cpu +=1
22              if cpu == 8:
23                  cpu = 0
```

To get data into a mapper, data is spliced into a specific location in the service.cpp file marked by a substitution flag. This is achieved by using a combination of the Unix stream editor, sed, and Perl programming language syntax for stream editing. Another part of the program code, aside from the map() function, acts as a network socket client which establishes a TCP connection and streams a map() functions computation result to a server.

The mappers are distributed evenly across all cores in round-robin fashion. This way the execution of a mapper is locked to a particular core using the Unix taskset command.

5.1.7 Mapper simulator

As a substitute for IncludeOS-based mapper, a simulator was built using Basic Calculator programming language (an arbitrary precision calculator programming language much like C) and a trivial network socket client based on the /dev/tcp/ device which is built in to most Linux distributions including Debian distributions. The simulator performs floating point arithmetic which is used to induce CPU overhead. At the end of computation, the simulator emits a result which looks exactly like that expected from an IncludeOS-based mapper. The result is a captured copy of the output of a map() function. The code has been redacted for better appreciation:

Listing 5.6: Mapper simulator

```
1
2 fnull=open(os.devnull, 'w')
3 subprocess.Popen(["/bin/bash", "OSsim.sh", "\%s" \% localip], stdout=fnull,
      close_fds=True)
```

The script, in essence, establishes a TCP connection to the listening network socket server associated with a reduce() function, and transmits its the result.

Listing 5.7: Mapper simulator

```
1
2 #!/usr/bin/bash
3
4 ip=$1
5
6 echo "9345678^123456" || time bc
7
8 #the number 1338 is the port number at which a server is supposed to be
      listening
9
10 exec 3<>/dev/tcp/$ip/1338
11
12 echo -e "some-data" >&3
```

5.1.8 Network Socket Server

At each reporting station - a supervisor or scheduler - a network socket server written in Python listens for connections from different hosts. Its code extends the Python Asyncore class which creates a non-blocking socket. For every connection, it writes all data received to a distinct file. A combination of a parameter which specifies the total number of connections to listen for and the files count serves as a control feature for determining when the server should stop waiting for connections and hand control to a reduce function.

Listing 5.8: Mapper simulator

```
1
2  def handle_read(self):
3          total_data=[]
4          data = None
5          while 1:
6              try:
7                  data = self.recv(1024)
8                  if data:
9                      total_data.append(data)
10                 else:
11                     time.sleep(6)
12             except:
13                 pass
14
15         count = 1
16         crtl = 0
17         total_data = "".join(total_data)
18         while os.path.isfile("/home/ubuntu/mapproject/results\%d.txt" \% count
               ):
19             count += 1
20             continue
21         datastr = open("/home/ubuntu/mapproject/results\%d.txt" \% count, "a+"
               )
22         datastr.write(total_data)
23         datastr.write("\n")
24         datastr.close()
25         for fil in os.listdir("/home/ubuntu/mapproject"):
26             if "results" in fil and os.path.isfile(os.path.join("/home/ubuntu/
                   mapproject",fil)):
27                 crtl +=1
28                 time.sleep(3)
29         if crtl >= controlvar:
30             result = []
31             for entrys in os.listdir("/home/ubuntu/mapproject"):
32                 if "results" in entrys and os.path.isfile(os.path.join("/home/
                       ubuntu/mapproject",entrys)):
33                     result.append("\%s/\%s" \% ("/home/ubuntu/mapproject",
                           entrys))
34             atexit.register(self.reduce, result)
35             sys.exit()
```

5.1.9 Reduce()

This component is written in Python as a method which extends the Python Asyncore class. It performs an operation similar to a mapper, but in its case, the input is a processed list of key-value pairs sent from multiple mappers and, possibly, supervisors. The results arrive in a json-like format over a tcp connection and are written to files on disk from where the reduce() function reads them. A key is a word which occurs in a data split, while the value is the number of times it occurs in that portion of data. The primary job of the reducer is to sum up the values of identical keys in the entirety of the result set, into a single value.

Listing 5.9: Map()

```
1
2  def reduce(self, dfiles):
3          result_dict = {}
4          for title in dfiles:
5              datain = open(title, "r")
6              for line in datain:
7                  line = line.replace("{", "")
8                  line = line.replace("}", "")
9                  try:
10                     line = line.replace("\"","")
11                 except:
12                     pass
13                 line = line.strip()
14                 line = line.split("*")
15                 for entry in line:
16                     if line=="":
17                         continue
18                     else:
19                         entry = entry.split(":")
20                         if entry[0].strip() in result_dict:
21                             result_dict[entry[0].strip()] += int(entry[1].strip
                                   ())
22                         else:
23                             result_dict[entry[0].strip()] = int(entry[1].strip()
                                   )
24             datain.close()
```

5.1.10 Data Generator

A short python script was created to generate any size of data by repeatedly writing a piece of text a number of times until the desired size is attained.

Listing 5.10: Generator.py

```
1
2  text = "original␣text␣replaced␣for␣brevity"
3  line = open("bigfile.txt", "a+")
4  for i in range(368000):
5      line.write(text)
6  line.close()
```

5.2 The Test Environment

At this point, working prototypes have been built, so reference shall be made to Lean Protoypes A and B as against the names of their corresponding designs:Lean alternatives A and B.

As alluded to in the introductory paragraph of this chapter, all experiments were performed on the Alto Openstack cloud platform. Each virtual server was assigned resources and an operating system suitable to their status. Virtual machines were

crafted from MLN templates with MLN commands. A summary of the resource profile of all virtual servers used is captured in table 5.1.

Table 5.1: Summary of system specifications

	Lean Prototype	
	A	B
Scheduler	m1.xlarge ; Ubuntu 14.04 Trusty	m1.medium ; Ubuntu 14.04 Trusty
Supervisor	m1.xlarge ; Ubuntu 14.04 Trusty	m1.xlarge ; Ubuntu 14.04 Trusty
Mapper	Mapper simulator	Mapper simulator

All network access were based on public/private key authentication. A scheduler node acted as a pseudo-gateway to a cluster and a potential base for configuring that cluster (when there are other virtual servers on the cluster). The path taken to a virtual server depended on established public/private key relationships (an ssh server must have the client's public key for successful authentication): for instance, a scheduler may only have transitive access to a supervisor if it does not have a direct key relationship with the latter but has such a relationship with an intermediate supervisor which has the desired relationship.

The files - scripts, temporary information files and sub-folders, and the text corpus file - are held in the a project folder called mapproject. This folder recurs at every non-mapper node. Files to be transferred to a downstream node are held in a sub-folder called Transfolder. At the successful completion of job processing, the intermediate or final result (depending on the category in focus) is written to a file in mapproject called, finalresult.txt. The text corpus file is named bigfile2.txt

5.3 Lean MapReduce Prototypes

The toolkit described above is used in slightly different ways to implement the Lean MapReduce alternative designs. The major difference lies in how supervisors are created, where mappers are hosted, and the nature of communication between nodes in the network hierarchy of a cluster.

5.3.1 Lean Prototype A

The step-by-step procedure which support the functionality of Lean prototype A includes the functions described under the toolkit subsection; these are structured in the main prototype A code file called maprunaltamain.py . The cluster is extended one level at a time, with the depth of the cluster based primarily on memory constraints. For a cluster with arbitrary number of levels (1 or greater), at each level a percentage of total available memory is carved out for use after factoring in the size of data each mapper should take, the memory size specification for each mapper (memory size should be greater than 3x the size of data), and other events - including caching and the computation needs of the reduce() function. The allocatable memory is then measured against the size of data.

Excess data above the capacity of the current node (scheduler or supervisor) is transported with Secure Copy (SCP) to the next downstream node (mandatorily a supervisor) which was created for the purpose of coordinating processing of this subset of data. The code block which implements this logic is presented below; again the code has been redacted for brevity:

Listing 5.11: Scoping the Cluster: Prototype A

```
1
2  to_alloc = int(math.ceil((availmem - (availmem *(2/10.0)))/3))
3
4  if to_alloc >= filesize :
5          numof_instances = int(math.ceil( float ( filesize ) / max_data_per_vm))
6          subprocess.check_call(["split","-d", "--number=%d" %
               numof_instances, "bigfile2.txt", "bigfile0"])
7  else:
8          subprocess.call (["split","-d", "--line-bytes=%d" % to_alloc, "
               bigfile2.txt", "tempbig0"])
9          numof_instances = int(math.ceil( float ( to_alloc )/max_data_per_vm))
10         filecnt = 0
11         for i in os. listdir ( localdir ):
12                 if "tempbig0" in i:
13                         excesscnt += 1
14         xdata = os.path.join( transfolder , " bigfile . txt")
15         os.chdir( localdir )
16         for m in range(1,excesscnt):
17                 if m < 10:
18                         subprocess.check_call("cat tempbig00%d >> %s" % (
                               m,xdata), shell=True)
19                 else:
20                         subprocess.check_call("cat tempbig0%d >> %s" % (m,
                               xdata), shell=True)
```

The spawn() function and prepRemote() functions are called where appropriate to create mappers and supervisors in the workflow. To deal with remote access challenges, ssh keys are distributed over http connections. At each host - except the mappers - a minimal webservice running on Apache2 was installed as part of the initial software configuration. When a growth threshold is detected at the scheduler (if there is yet to be a supervisor) or the supervisor furthest from it, the current system user's ssh public key file is written to the webservice folder (/var/www/html). During the creation of a new virtual machine, it is coerced to download the file from its creator node using the GNU wget tool, and subsequently parse the content into the authorized_keys under the home directory of the target user, which by default is Ubuntu.

To fully automate growth of a cluster, MLN was installed at each node and manipulated to create and remove virtual machines from the Openstack project account. For the purpose of of tracking the creation time of a supervisor at each level, a timing mechanism was coded into the prototype. When the height of the cluster is greater than 2, information about the time it takes to create a new node is stored locally by every creating node (the current node). The information is extracted by the next upstream node using SCP when processing is concluded at

the current node. A timing mechanism was also written into the program to track Time to Complete: the time interval from the point right where data is split to the point where the final result is written to disk.

5.3.2 Lean Prototype B

Lean prototype B retains the meaning and logic of the functions in the toolkit just like prototype A does. It determines the number of supervisors needed and splits the data to be processed evenly using this number. It uses the memory size for an m1.xlarge Openstack flavor, chosen for every supervisor, in carrying out memory calculation similar to that performed in prototype A. The code which does this, is a follows:

Listing 5.12: Scoping the Cluster: Prototype B

```
1  to_alloc = int(math.ceil((availmem - (availmem *(2/10.0)))/3))
2
3  if to_alloc >= filesize :
4          numof_supervisors = 1
5          subprocess.call(["cp", "-r", "transfolder", "transfolder0"])
6          subprocess.call(["cp", "bigfile2.txt", "transfolder0/bigfile.txt"])
7          startSupervisors(localip,numof_supervisors)
8          remoteExec(ip_list[0], 0)
9
10 else:
11         numof_supervisors = int(math.ceil(float(filesize)/to_alloc))
12         subprocess.call(["split","-d", "--number=%d" % numof_supervisors
               , "bigfile2.txt", "bigfile0"])
13         for x in range(numof_supervisors):
14                 subprocess.call(["cp", "-r", "transfolder", "transfolder%d" %
                       x])
15                 subprocess.call(["mv", "bigfile00%d" % x, "transfolder%d/
                       bigfile.txt" % x])
16         for k in range(numof_supervisors):
17                 startSupervisors(localip, k)
18         for hostip in ip_list :
19                 remoteExec(hostip, num)
20                 num += 1
```

To create any number of supervisors needed to process a workload, the core prototype B script named maprunaltbmain.py called an MLN template with a few unique parameters to generate unique identifiers for each virtual machine. Execution of remote commands was a lot easier than in prototype A since only one public/private ssh key-pair was needed, hence the http medium for key transport used in prototype A was not required.

The TTC and spawn time tracking mechanism used in prototype A were employed here; however, in this case the functionality was concentrated at the scheduler due to its global view of the cluster.

5.4 Pre-experiment Evaluation

To ensure that the prototypes work as expected, they were tested extensively under different workloads. The test looked at the results returned for a each workload in several iterations: the check confirmed that the results were as expected, and that there was no difference in results for each iteration.

For each prototype, the expected cluster roll out pattern and communication paths were verified using ssh access attempts (a supervisor cannot be accessed from any node other than that directly upstream of it) and tcp connections monitoring. The following are snippets of the expected intermediate and final processing results, respectively:

Listing 5.13: Sample of Intermediate Result

```
1 {{trump:12500∗a:250000∗about:37500∗absolutely:37500∗abuse:12500∗access
      :50000∗according:12500∗across:25000∗action:12500∗added:25000∗addiction
      :25000∗address:12500∗advance:12500∗advisers:12500∗after:12500∗aim:12500∗
      alaskan:12500∗alike:12500∗all:25000∗allocation:12500∗already:12500∗also
      :25000∗although:12500∗ambassador:37500∗among:25000∗amongst:12500∗an
      :37500∗and:262500∗angus:12500∗any:12500∗april:12500∗are:87500}
```

Listing 5.14: Snippet of Final Result

```
1
2  WORD            COUNT
3  −−−−−−−− −−−−−−−−−−
4  all             150000
5  month           75000
6  systemic        75000
7  oldest          75000
8  children        75000
9  issues          150000
10 suicide         450000
11 pledged         75000
```

Using timing values, the absence of systematic errors was determined at a 95% degree of confidence.

At the end of each job window during the tests, the clusters were effectively taken down by using a script which invokes MLN stop and destroy commands for the Openstack virtual machines, and the Unix kill command for processes forked during the job window. The cleanup operation also involved deleting temporary files which were used to capture different events and data, during the job processing window. The process is straightforward in prototype B as the script is the whole cluster can be erased at the schedular; but in prototype A the cluster is scaled back one level, each parent node deleting its child supervisor until only the scheduler is left. Variants of the following shell script are used in prototypes A and B for this purpose and are passed a level number as a parameter.

Listing 5.15: Clean-up script

```
1  #!/bin/bash
2
3  sleep 20
4  rm -rf bigfile0* screenlog.0 transfolder/ bigfile.txt mapresult0* results*
       finalresult.txt
5  sleep 6
6  mln stop -p leanmra$1 -h downstreama$1
7  sleep 6
8  mln remove -p leanmra$1 <<-EOF
9  y
10 EOF
```

Chapter 6

Measurement, Analysis and Comparison

This chapter describes the experiments conducted to capture empirical data about the performance of Lean MapReduce prototypes, as well the analysis which ensued. The experiments were performed to get a measure of the time it takes for prototypes A and B to process some unstructured data. Of particular interest was the amount of time spent adding requisitioned compute resources to an adhoc cluster, by way of creating and adding new supervisor nodes to it. The data obtained was analysed in the analysis section to make clear the timeliness of the operation of each prototype, and link into further analysis about the optimal and efficient resource utilization features of the prototypes in view.

Data block size was set at 200MB, while the available memory calculation formula was defined at each non-mapper node with operands which coerced a size of 1100MB of RAM. The data collected have been converted to graphs for ease of interpretation, the graphs as well as the experiment parameters and data gathering techniques are detailed below.

IncludeOS-based mappers were replaced with mapper simulators due to technical challenges which resulted in a preponderance of failures during testing.

6.1 The Experiments

Five categories of experiments were performed, each with a distinct text corpus size: 1026MB, 2052MB, 3078MB, 4104MB,and 5130MB. Two types of measurements were taken: measurements for the time it takes to prepare a cluster, particularly the time it takes to spawn all required supervisors if an extension threshold is reached; and the TTC for a complete job processing window (job duration).

The spawn time for 1 supervisor was measured from the point the preparation of an MLN template began to the point when all initial software packages had been successfully installed on the new node - this point was delineated by the ability to access the webservice installed on that node. The TTC time began to count from

the point the script was executed, and ended when the final result was produced at the scheduler.

Data collection was done slightly differently in prototypes A and B: At prototype B, the results were written at the scheduler node to two files named Duration.txt and Spawnup.txt, for TTC and spawn time, respectively. Here the spawn time comprised of the total continuous amount of time spent acquiring more supervisors for the cluster. At prototype A, the spawn time was recorded by every node which requisitioned a new supervisor downstream of it. Such records were then transferred over a secure file transfer channel from one node to the next, in the direction of the scheduler node. As it was passed along, the record was merged with the spawn time record of the recipient upstream node; a last merge operation took place at the scheduler where the result of the final summation was written to the Spawnup.txt file. The TTC was recorded in identical manner to the approach in prototype B.

Each experiment category was iterated 30 times, giving a total of 150 spawn time random samples and as many TTC entries for each Lean prototype. For proper distinction, the charts for prototype are in blue color, while prototype B charts are in brown.

6.1.1 Experiment 1: Processing 1026MB

In the first experiment, a text corpus of 1026MB was generated using the data generator script (refer to 5.1.10). The size of the corpus did not trigger the defined extension threshold in alternative A, but warranted the creation of a supervisor in prototype B; hence only TTC was recorded for the former, while prototype B generated both TTC and spawn time data.

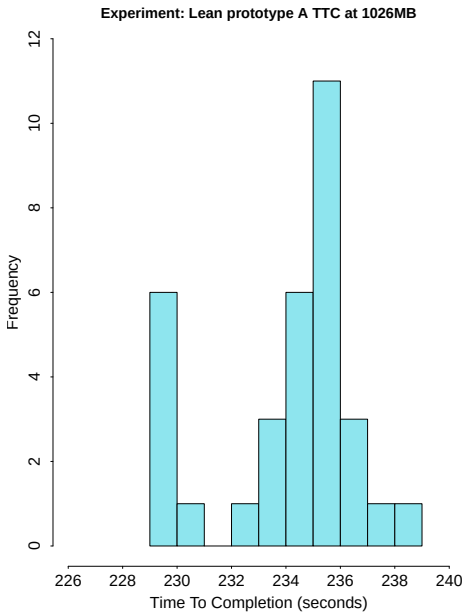

Figure 6.1: Prototype A: Time to Complete for 1026MB

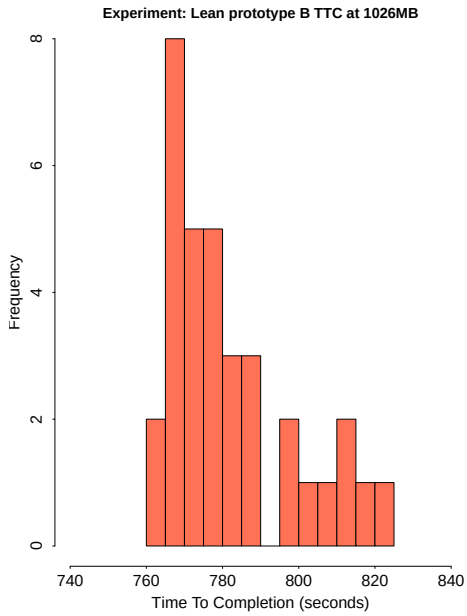

Figure 6.2: Prototype B: Time to Complete for 1026MB

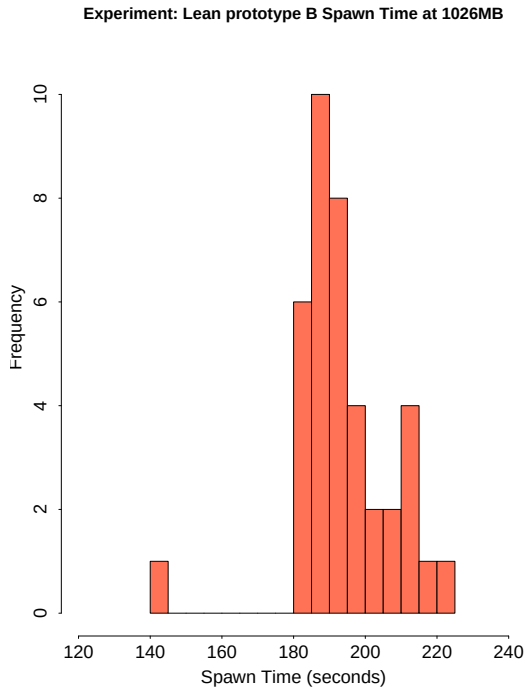

Figure 6.3: Prototype B: Spawn Time for 1026MB

The average spawn time in prototype B was about 25% of its TTC, while the average TTC of prototype A was about 30% of that of prototype B. The single left-oriented outlier (creation time for a one supervisor) in the spawntime distribution of protoype B could be attributed to a relatively quick response time by the cloud mechanism responsible for crafting virtual machines, and the time frame in which the baseline set of software packages were installed. The properties of the data sets are summarized in table 6.1:

Table 6.1: Statistical description of data: 1026MB

	Prototype A		Prototype B	
	TTC	Spawn Time	TTC	Spawn Time
Mean	234.46	N/A	782.56	194.10
Median	235.00	N/A	777.50	193.05
StdDev	2.76	N/A	16.52	13.82

6.1.2 Experiment 2: Processing 2052MB

In the second experiment, a text corpus of size 2052 was processed simultaneously by prototypes A and B. This lead to the creation of a supervisor in prototype A and 2 supervisors in prototype B.

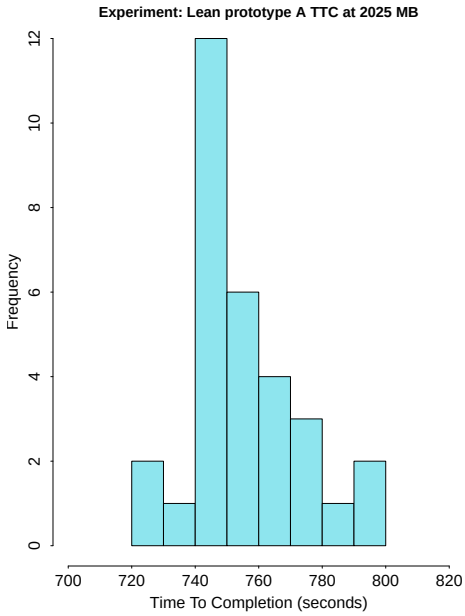

Figure 6.4: Prototype A: Time to Complete for 2052 MB

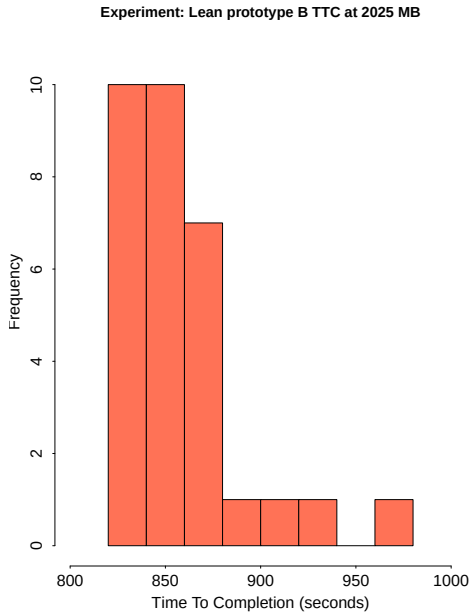

Figure 6.5: Prototype B: Time to Complete for 2052 MB

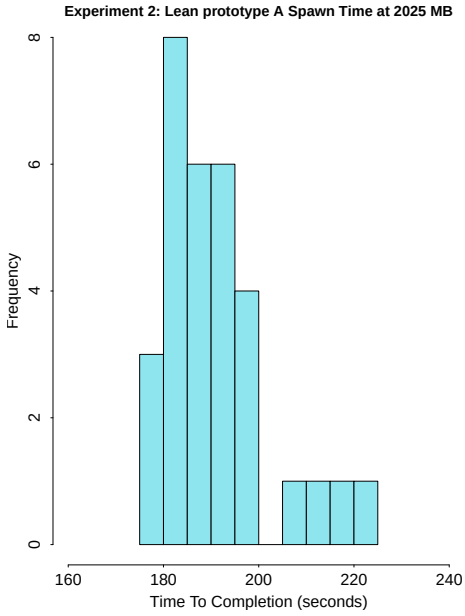

Figure 6.6: Prototype A: Spawn Time for 2052 MB

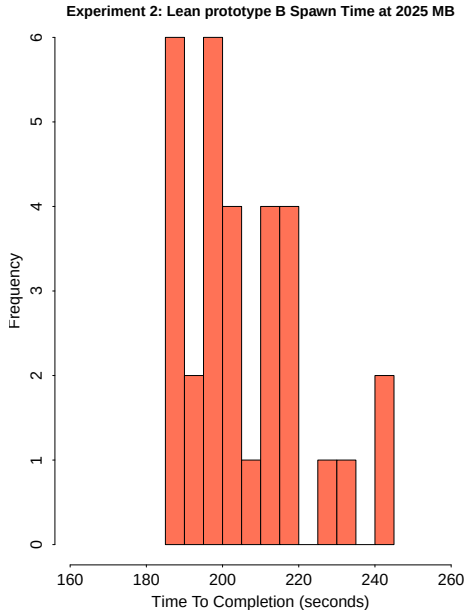

Figure 6.7: Prototype B: Spawn Time for 2052 MB

The spawn time of prototype A was 24 % of its TTC. This ratio was mirrored by prototype B although with slightly greater values. The extrema to the right of prototype B's distribution could be attributed to activity level and functional state of the underlying cloud platform. This is summarized in table 6.2

Table 6.2: Statistical description of data: 2052MB

	Prototype A		Prototype B	
	TTC	Spawn Time	TTC	Spawn Time
Mean	755.03	191.57	861.01	207.15
Median	751.00	190.00	857.00	201.00
StdDev	16.28	10.05	31.54	18.84

6.1.3 Experiment 3: Processing 3078MB

In experiment 3, two text corpus of size, 3078MB were generated. These were processed by in parallel by prototypes A and B. Prototype A created 2 supervisors to complete the job, while prototype B spawned 3 supervisors for the same purpose. TTC and spawntime for the job window are shown in the graphs below:

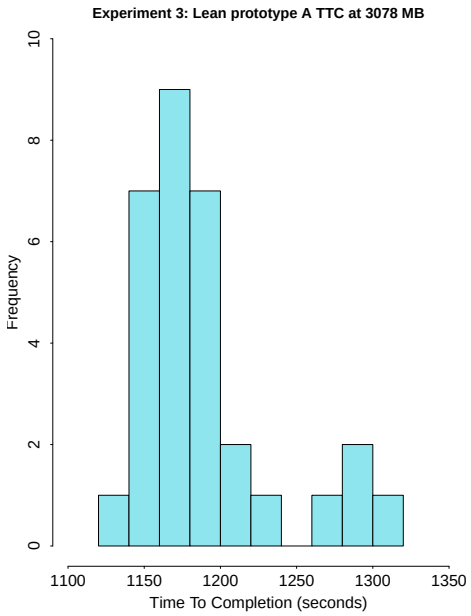

Figure 6.8: Prototype A: Time to Complete for 3078 MB

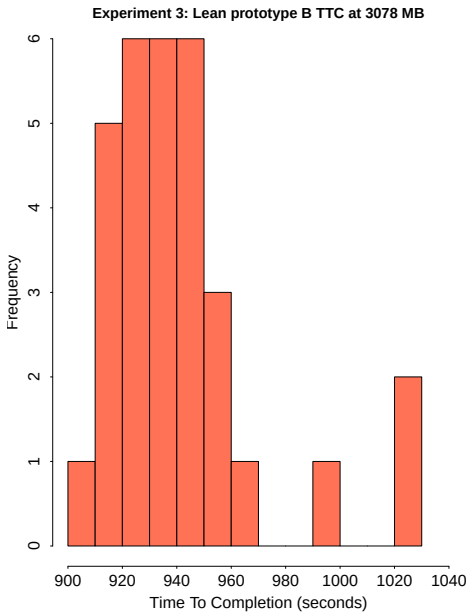

Figure 6.9: Prototype B: Time to Complete for 3078 MB

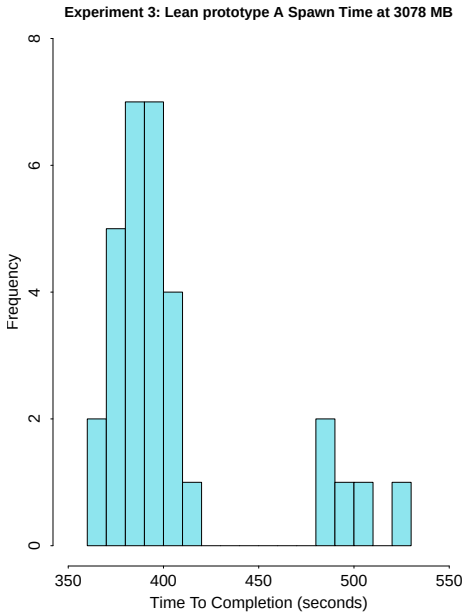

Figure 6.10: Prototype A: Spawn Time for 3078 MB

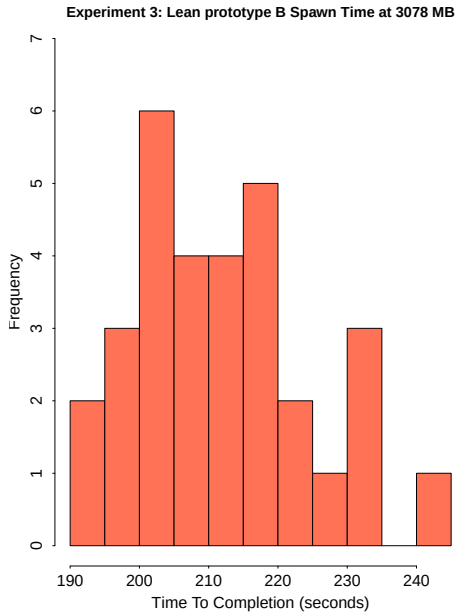

Figure 6.11: Prototype B: Spawn Time for 3078 MB

The spawn time for prototype A constituted 35% of its TTC, while the data for prototype B showed that spawn time made up 23% of the TTC. The wide variation in the distributions of TTC and spawn times were attributed to activity flux being experienced by the underlying cloud platform, which in some cases slightly degraded its performance. The data obtained at 3078MB is statistically summarized in table 6.3

Table 6.3: Statistical description of data: 3078MB

	Prototype A		Prototype B	
	TTC	Spawn Time	TTC	Spawn Time
Mean	1184.74	405.44	939.87	211.98
Median	1176.00	391.50	936.00	211.50
StdDev	39.16	41.95	26.11	11.67

6.1.4 Experiment 4: Processing 4104MB

In experiment 4, a text corpus of size, 4104MB was generated. This was processed in independently by prototypes A and B. Prototype A created 3 supervisors to complete the job, while prototype B spawned 4 supervisors for the same purpose. TTC and spawntime for the job window are shown in the graphs below:

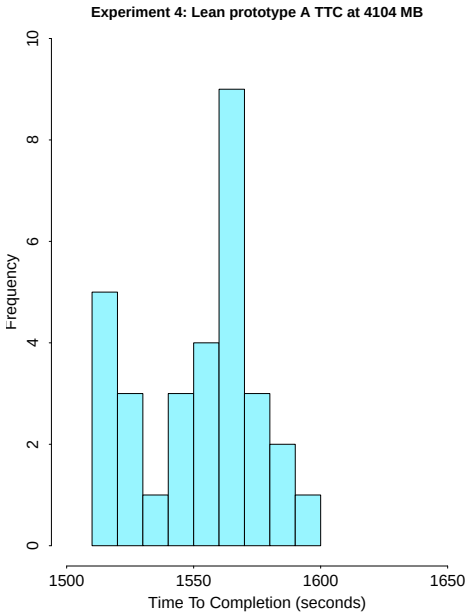

Figure 6.12: Prototype A: Time to Complete for 4104MB

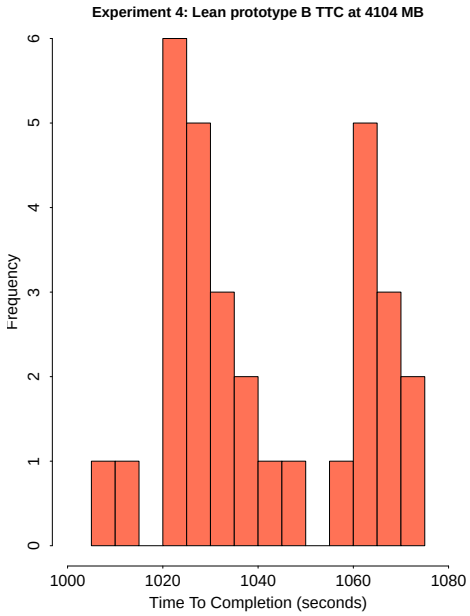

Figure 6.13: Prototype B: Time to Complete for 4104MB

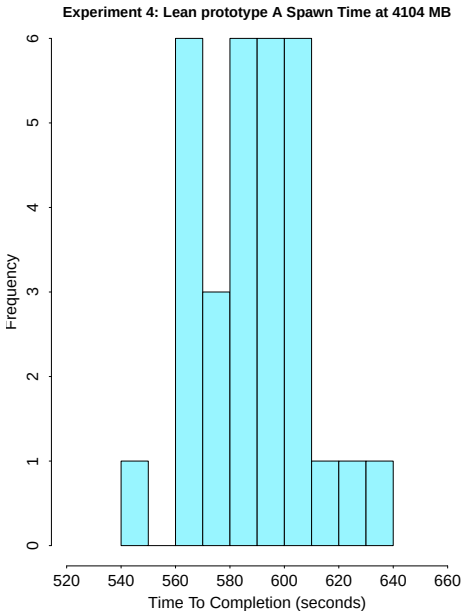

Figure 6.14: Prototype A: Spawn Time for 4104MB

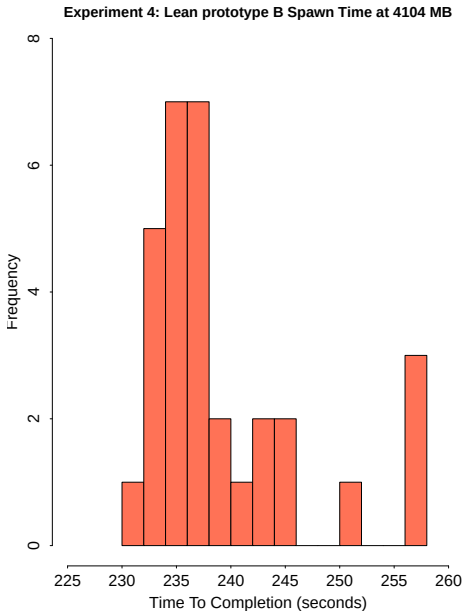

Figure 6.15: Prototype B: Spawn Time for 4104 MB

Average spawn time was 38% of average TTC in prototype A's operation, and stood at 23% in prototype B's operation.

Table 6.4: Statistical description of data: 4104MB

	Prototype A		Prototype B	
	TTC	Spawn Time	TTC	Spawn Time
Mean	1549.03	587.81	1038.77	238.22
Median	1551.00	590	1032.00	236.00
StdDev	23.58	18.95	18.04	6.97

6.1.5 Experiment 5: Processing 5130MB

A text corpus of size, 5130MB was generated in this round of experimentation. To process the data, prototype A spawned 4 supervisors, while prototype B needed 5 supervisors.

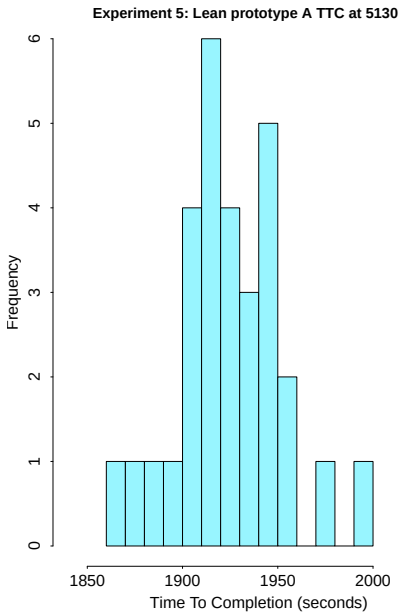

Figure 6.16: Prototype A: Time to Complete for 5130MB

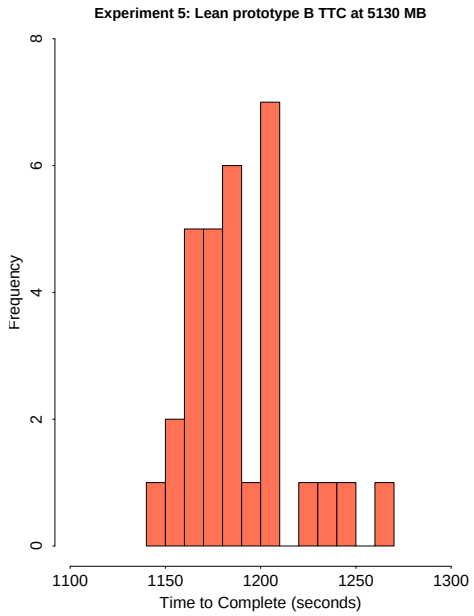

Figure 6.17: Prototype B: Time to Complete for 5130MB

Table 6.5: Statistical description of data: 5130MB

	Prototype A		Prototype B	
	TTC	Spawn Time	TTC	Spawn Time
Mean	1927.00	780.39	1190.39	292.76
Median	1925.00	776.00	1184.00	290.00
StdDev	28.51	24.22	26.77	11.46

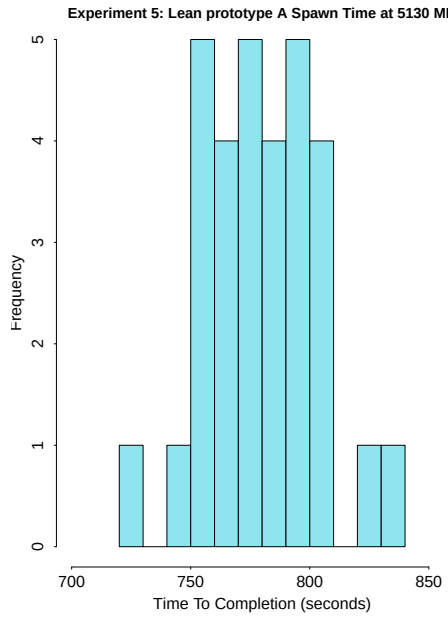

Figure 6.18: Prototype A: Spawn Time for 5130MB

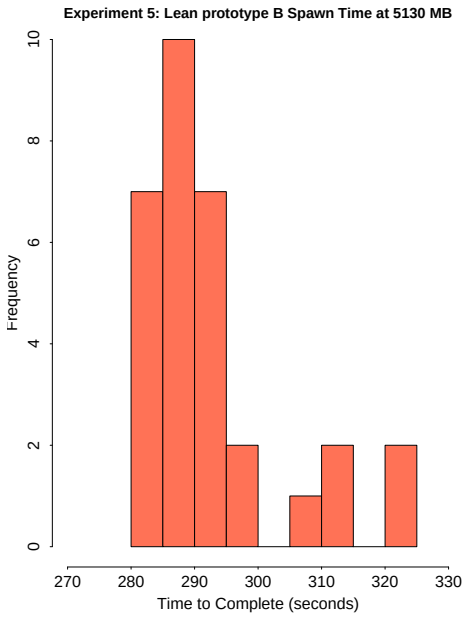

Figure 6.19: Prototype B: Spawn Time for 5130MB

The average spawn time in this round of experiments was approximately 41% of average TTC for prototype A, and stood at approximate 25% for prototype B.

6.2 Data Analysis

In this section, the results of the analysis of the data described in the experiments section is presented. The results obtained were as expected in terms of resource use, although there was slight deviation from notational approximation in terms of the amount of total and component times needed for one complete job window (refer to 4.3.5). These are highlighted in the following subsections. The outcome of the analysis will help determining to what extent the objectives of the thesis have been met: to design a adhoc MapReduce architecture which optimally and efficiently utilizes system resources to reliably process big data workloads and deliver results in a timely manner.

6.2.1 Interpreting Prototype A data

From the results, it can be seen that as capacity thresholds for nodes were reached, additional units of compute resources in the form of supervisors were added to the cluster until the workload was completely accommodated. As the cluster size grew, Time to Complete increased linearly with the workload size. In the first experiment where the text corpus size was 1026MB, there wasn't a need for additional compute resources hence the job was processed entirely by the scheduler. This resulted in a TTC of about 235 seconds. From the second experiment to the last, more compute resources were added according to the number of capacity thresholds reached in sequence, of which the impact was a considerable increase in TTC. In the second experiment their was a 320% increase , equivalent to 520 seconds, but the increase in TTC became roughly constant in the subsequent experiments with the increase hovering within the range of 360 to 430 seconds.

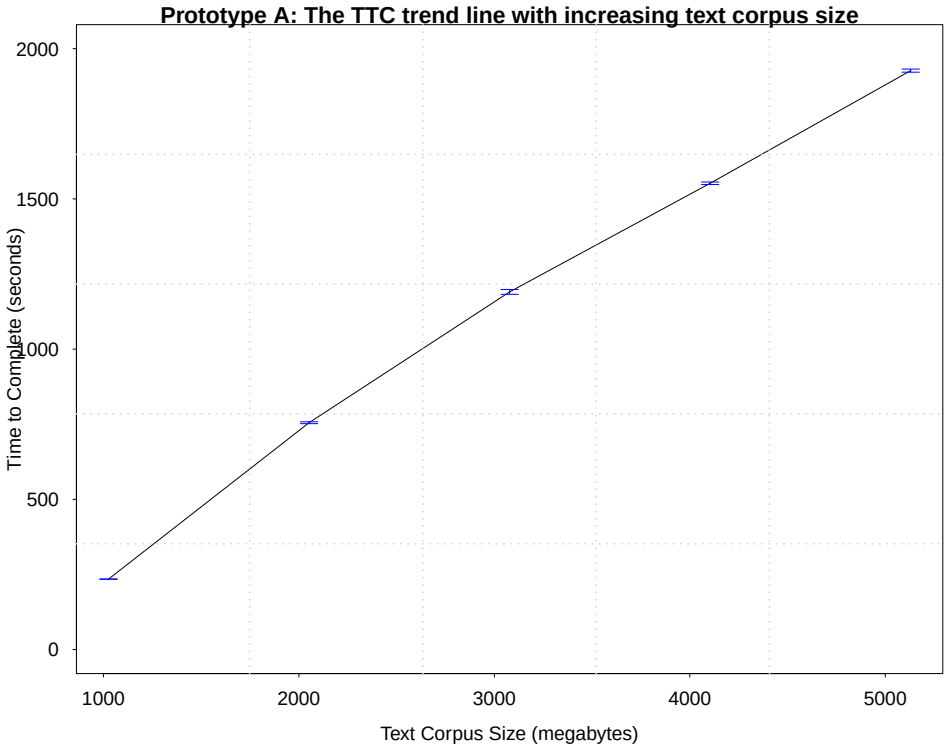

Figure 6.20: Prototype A: Trend line of TTC

Similarly, as the number of needed supervisors increased so did the amount of time spent spawning them. The roughly 100% increase of this time facet can be attributed to the perfectly sequential way compute resource sets were added to the cluster as it grew from the scheduler to the last needed supervisor. The spawn time was also an influential factor in the increase of average Time to Complete; as the workload size increased, the average spawn time increasingly became a dominant component of average TTC, growing from 24% for one supervisor to 41% when 4 supervisors were needed.

The factors which became less dominant aspects of TTC with increasing workload

size, were the processing time spent by the different levels of the map() function (distinguished by host node hierarchy), the time spent transporting and storing intermediate data, and the time used up by the reduce() function for its merge operation.

6.2.2 Interpreting Prototype B data

In the results obtained during the experiments involving prototype B it shows, by virtue of spawn time data being recorded from the very first experiment, that for a workload to be processed - irrespective of size - a supervisor or multiple supervisors must be created. The number of supervisors corresponded to the workload size divided by the predetermined capacity limit of each supervisor.

The TTC in the first experiment was relatively high when contrasted with that of prototype A. Average spawn time made up 25% of average TTC. The average spawn time to TTC ratio was about the same for all experiments, although TTC increased linearly with increase in workload size.

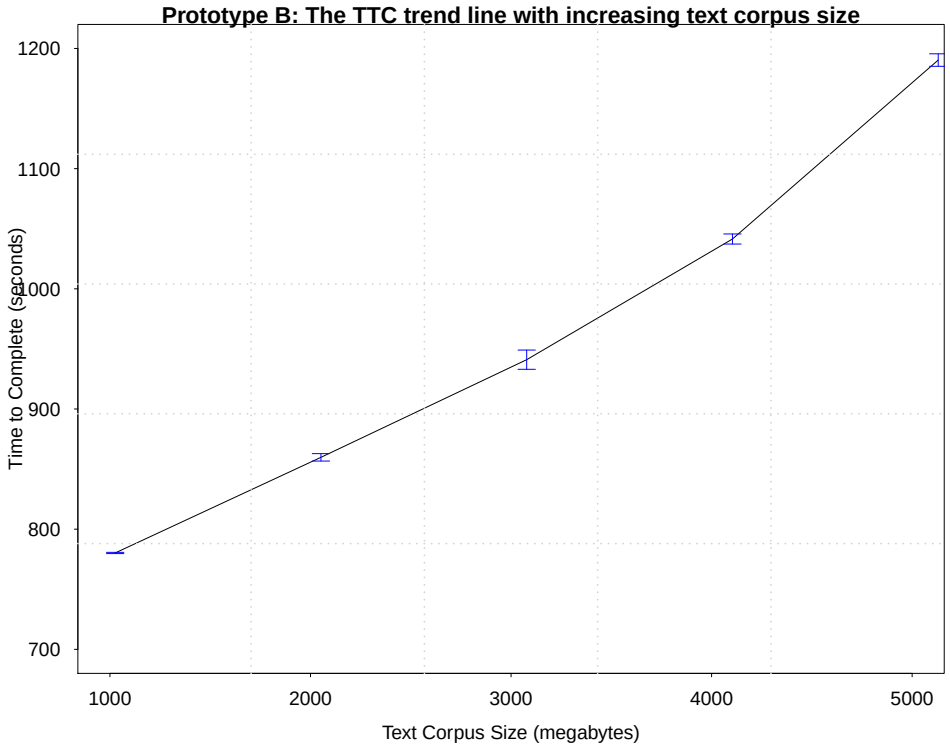

Figure 6.21: Prototype B: Trend line of TTC

The supervisors weren't created in a parallel order, but their periods of creation overlapped considerable. This played a big part in the noticeable increase in TTC with every next experiment. Although there were significant differences between the average TTCs posted per workload size, the differences between them could be described as small when compared with the differences observed in prototype A. This could largely be attributed to concurrent application of map() functions to their respective data splits.

6.2.3 Comparison of Prototypes A and B

Prototype A and B reacted differently when applied to different workloads. In the first experiment, prototype A requisitioned no additional compute resource sets for processing its workload and delivered results in an average time of 234.46 seconds ($\pm$0.98, 95% CI), which is significantly different from that of prototype B ($\pm$5.75, 95% CI) where an additional compute resource set was needed.

For the subsequent experiments, prototype A needed one compute resource set less than prototype B. The gap between individual TTCs was relatively not considerable in the second experiment, with prototype A delivering within less average time (755.81 $\pm$6.21 vs 861.01 $\pm$10.83, 95% CI); however going by their confidence intervals in fig 6.22, it can be seen that the difference is significant.

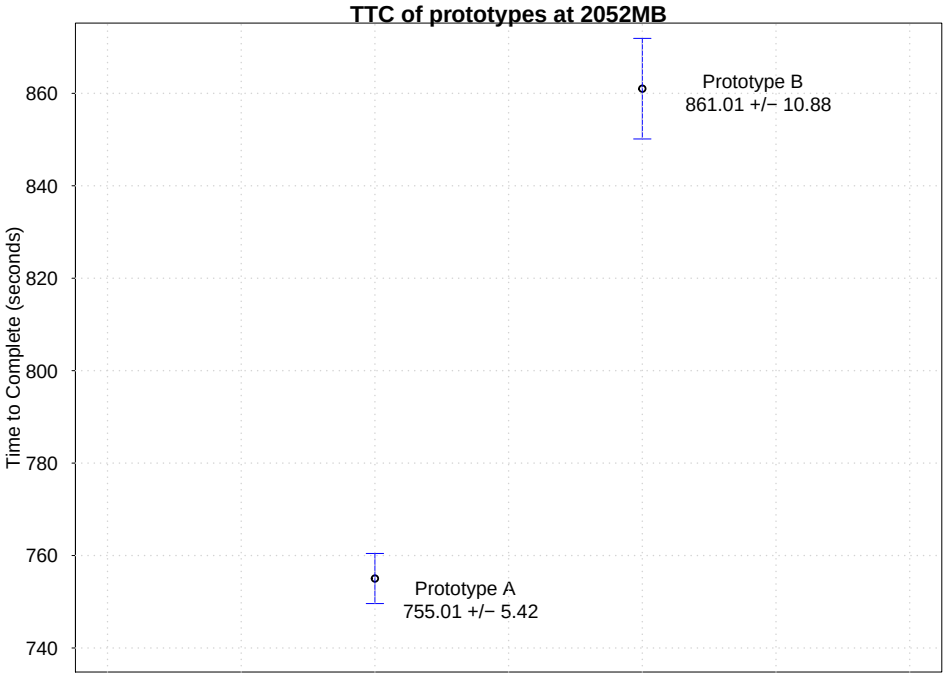

Figure 6.22: Comparing TTCs at 2052MB

As can be seen in fig 6.23, with increasing workload, the TTC of prototype B became better that of prototype A with the margin of difference widening and peaking in the final experiment (1190.39 $\pm$10.64 (prototype B) vs 1927 $\pm$9.82 (prototype A), 95% CI).

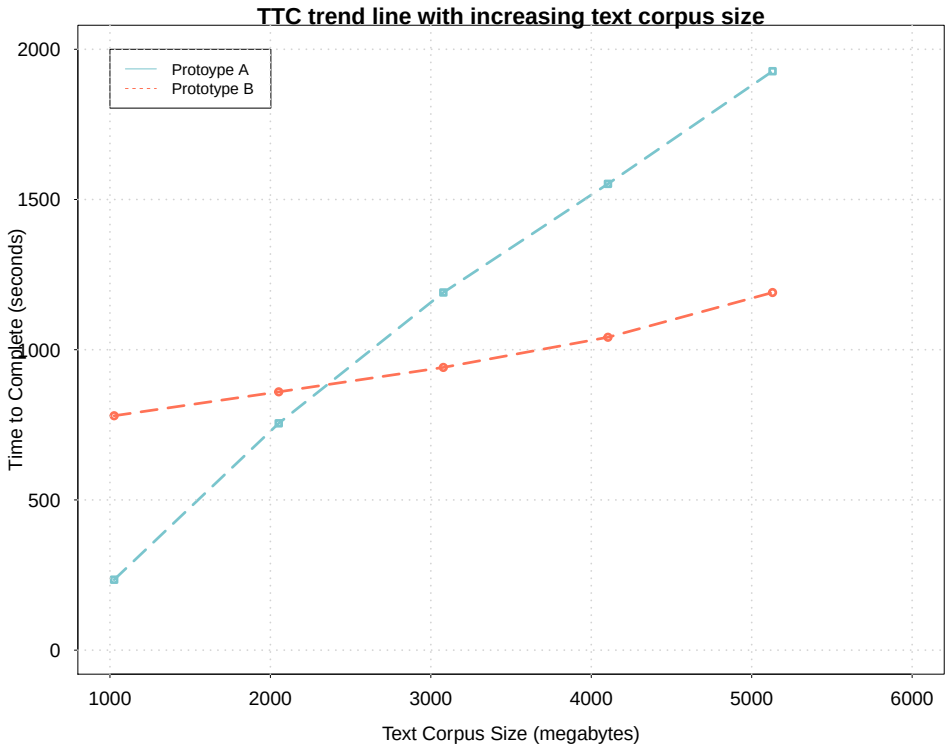

Figure 6.23: Comparison of TTCs

Based on the foregoing analysis, prototype A is efficient for instances of smaller workload which require just one additional unit of compute resources, but when the workload demands more supervisors, then prototype B is more efficient.

A deeper inspection of the results also revealed that for smaller workloads, prototype A required less network bandwidth for moving text corpus which were over the capacity of the current node to the next node; however, it required increasingly more bandwidth than prototype B to do the same for jobs where it required 3 or more supervisors.

In the experiment where prototype A processed 1026MB of text corpus, it expended zero network bandwidth as the workload was within the capacity of the scheduler, whereas prototype B had to transfer the text corpus to a supervisor. For 2052MB of text corpus, prototype A transferred 1026MB to its first level supervisor, while prototype B moved its corpus over 2 links to 2 supervisors. However, in the next experiment, the total amount of data transferred evened out: in prototype A, the scheduler moved 2052MB of text corpus to the first level supervisor, which in turn moved 1026MB to the second level supervisor; this adds up to the 3078MB of text corpus moved on to supervisors by the scheduler in prototype B. In the fourth experiment, prototype A transferred 6156MB as against 4104MB in prototype B.

The foregoing can be generalised as follows: for an instance where the size of the corpus is n, assuming a constant maximization factor m, and a Lean MapReduce

tree (cluster) of height, $h = \lceil \frac{n}{m} \rceil$, the approximate bandwidth, B, consumed by prototype A can be expressed thus:

$$B \leq \sum_{i=1}^{h} n - i * m$$

while that consumed by prototype B was approximately the same size, n, as the input data.

6.3 Design Iteration: A Cloud Tenant Constrained Lean MapReduce

The analysis from the preceding section has revealed that Lean Prototype B is more efficient than Lean Prototype A in terms of Time to Complete and network bandwidth, given a big data premise, but that the inverse is the case in terms of the amount of compute resources used (compute resource sets). A prominent constraint which influenced both these outcomes was scheduler and supervisor capacity.

The Lean MapReduce design is such that map tasks are run in mapper nodes (possibly IncludeOS instances) which are hosted on a QEMU-KVM hypervisor; in a cloud computing scenario, this introduces an additional layer of management. To break the performance limitation imposed by multiple layers of management, mapper nodes will run directly on the cloud platform, right beside supervisors - this is based on the assumption that the mapper has virtualization support on the cloud platform. This relieves the controllers (scheduler or supervisor) of the burden of hosting resource-demanding compute entities - in essence, breaks the individual node capacity constraint - and leverages the extent of a cloud project or tenant resource quota.

When the rules set which govern the creation of workers for each prototype is examined, as well as the amount of bandwidth needed in this respect, it is apparent that the design of prototype A (Lean alternative A) can be slightly modified for this purpose; in prototype A, a scheduler may have a direct relationship with mapper nodes, while prototype B enforces a transitive relationship (via supervisors). This policy still holds when tight coupling of controller and mapper nodes is relaxed.

Cloud architectures generally adopt a multi-tenant setup where each tenant has a negotiated resource quota. With this in mind, the maximization rule for prototype A can be evolved from an intra-node-based consideration to a tenant-quota-based consideration. In this state of design, a controller node is given charge of the available resources of a cloud tenant, non-monopolistically. Based on predetermined task slot size, it will create and run the necessary number of mapper nodes next to it. The controller still hosts the designated reduce() function for all mappers which sprung from it.

Typically, the size of a tenant is negotiated with proper accommodation of a target job in mind. In the immediate context, if the resource quota of the current tenant

in exhausted, the excess workload will be offloaded to another tenant which the current controller is privy to. A supervisor has to be created at the new tenant to manage the MapReduce affairs, this way the arborescence of the original design is maintained. Because tenants are not objects which are dynamically created, for additional tenant quotas to be used, a list of accessible tenants must be prepared prior to job execution and handed from controller to controller (typically from scheduler to first level supervisor, and further up the chain).

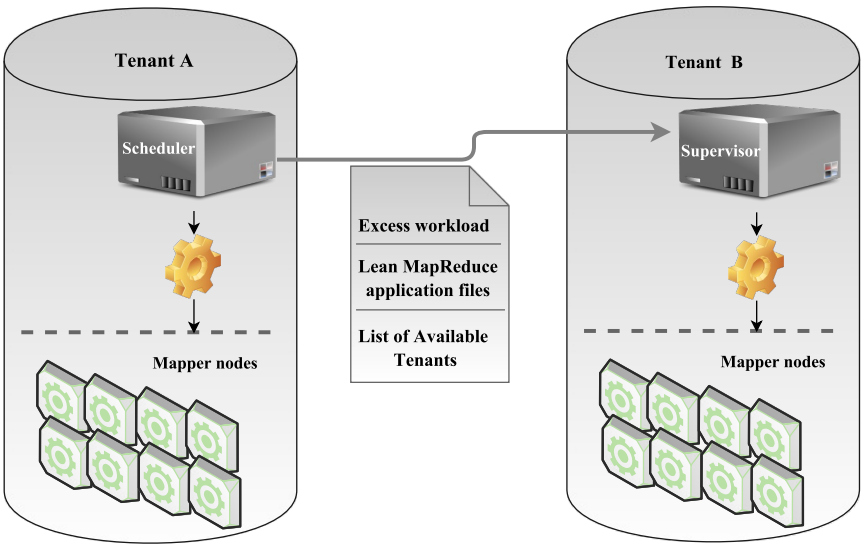

Figure 6.24: Tenant constrained Lean alternative A

Given that the size of a tenant usually considerably exceeds the resource specification of a single node, this design whittles capacity constraint which previously led to a lot of time being spent extending a cluster through the addition of compute resource sets. It also reduces the amount of data transferred between nodes. It however introduces a new performance limitation since a single reduce() function may have to merge intermediate results from relatively more map() functions. Where a virtual server may have 1 to 48 VCPUs with between 0.5 to 2000 gigabytes of memory, and 1 to 48000 gigabytes of storage in Amazon web services or an Openstack private cloud installation, a tenant is typically sized to contain enough system resources to create one or more of such servers.

So far, the focus has been on data-volume-driven job. For a compute-intensive job which involves far less data, the new design would also be a good fit since workloads are being executed directly on the cloud platform as against a nested virtualized arrangement. In addition, because in this scenario significantly less bandwidth is required, it opens up for cloudbursting, where instead of relying on multi-tenant cluster growth in the domain of a single cloud provider, multiple tenants on disparate cloud platforms are used. The decision to co-opt resources from a new cloud platform may be based on changing pricing realities if the platforms in picture are public clouds.

Chapter 7

Discussion

This chapter reflects on the results obtained in the course of the project work in relation to the problem domain and statement set forth as the motivation for this thesis. The reflection also encompasses viable alternative approaches, suggestions on improvements which could be co-opted in future work, and potential impact on the big data sphere.

7.1 Implementation of Alternative Lean MapReduce Designs

The goal of this paper was to explore a novel MapReduce architecture for optimal and efficient use of resources in timely analysis of big data. This was achieved through two architectural designs, inspired by the form and features of the B-Tree abstract data structure. These served as representational alternatives for how a MapReduce cluster may be grown into place for an adhoc arrangement. Based on these designs, two prototypes were developed, experimented upon, and their resource and time-oriented performance measured, analyzed and compared. Through this process, the propriety of each prototype for different workload magnitudes and resource requirements was discovered.

The result of experimentation was as expected, and the outcome of the analysis showed that prototype A was better than prototype B in terms of the amount of resources needed by one unit of compute resources (one supervisor) under all conditions. This difference is substantial when a few units are needed, but becomes increasingly negligible as the number of needed compute resource sets increases. Prototype B delivered a constant spawn-time to TTC percentage under different workloads and resources needs, while prototype A showed an increase for the same percentage as the workload and the attendant number of resources needed was scaled up. It was also observed that as the number of needed compute resource units gradually increased, the spawn time and TTC of prototype B increasingly became better than that of prototype A.

The general observations made supported the anticipated correlation of Spawn Time and TTC for both prototypes: prototype A showed a galloping increase in both performance axis, while prototype B demonstrated a slight increase over time

for both measurements. The results for prototype B were not entirely as expected, as it was anticipated that the TTC for a workload requiring several supervisors would be approximately equal to that for an instance requiring just one supervisor, instead there was a marked difference in TTC owing in part to a near-congruent behavior in spawn time.

Multiple iterations of each experiment category were conducted to get a good number of random samples for a tighter margin of error. Wide variations in results - abnormally long execution times, or unusually short execution times due to premature termination - were sometimes observed during experimentation. This was due to instability of the cloud, with a ripple effect across the entire system. Spawn time was usually the affected performance facet and this inadvertently swelled TTC. To obtain fair results, experiments were scheduled for periods when the platform was deemed to be in a normal functioning state.

For a finer grained characterisation of performance, metrics for the precise amount of time it took to process the input to a map() function, the counterpart time for applying the reduce() function to intermediate data, and the time spent transferring intermediate data, could have been splintered off from TTC. This will improve decisions about the viability of the prototypes for a deadline-bound MapReduce job, beyond the fact that prototype B spends roughly 25% of a job window preparing a Lean MapReduce cluster, while this factor grows from about the same percentage and tends towards a much higher value in prototype A. The outcome would be proper accounting of the sizable difference between TTC and spawntime. Another facet which could have been measured is inter-vm bandwidth (between a scheduler and a supervisor, or two supervisors) and consumption rates: although intra-cloud bandwidth is free, this would give an idea of the potential cost factor in a scenario where data has to be transported over a monetized infrastructure particular one provided by an Internet service provider.

7.1.1 Compute Resource Utilization and Measurement

CPU and memory measurements which were initially planned had to be ignored. While both measurements would help to validate the extent to which these system resources are utilized, they would not help in differentiating the performance of the prototypes. The same type of compute resource sets - except for the scheduler in prototype B - were used in both prototypes, and were loaded similarly using available memory in dynamically determining the number of task slots per node. The settings used during experimentation meant the number of task slots used were the same across board. CPU usage went along the same line: the execution of a map task was pegged to a particular CPU core, where the CPU cores allocation was done in round-robin style so that the execution was evenly distributed across all cores.

7.1.2 Relatedness of Project Outcome to Problem Statement and Definition of Lean

In the problem statement, three key requirements were established for a Lean MapReduce architectural design, these are:

- An architectural design which supports adhoc provisioning of a MapReduce system:

  Adhocness of the Lean MapReduce design is implied since the clusters in the showcased prototypes are acquired in a Just-in-Time manner for each new workload, and removed at the end of each job window.

- A design which leads to optimal and efficient utilization of compute resources:

  Following the discussion in 7.1 (paragraph 2), the optimality and efficiency of the designs is implied since the number of locally hosted tasks is dynamically established so that the allocatable memory in a system is completely utilized. However, this was not followed strictly - nor measured explicitly - during experimentation in order to expedite the process.

- Timely delivery of results:

  Provincial timeliness of the Lean MapReduce prototypes was established through a comparison of them, where the outcome revealed that prototype B delivered a better TTC for workloads requiring more than 1 additional compute resource set (supervisors) - a tendency which is far more likely in a big data context. To reiterate, spawn time was approximately 25% of TTC for prototype B.

  For a broader verification to be possible, Lean prototype B will have to be deployed on the same scale as an established MapReduce implementation such a Hadoop. Given that the established counterpart may typically involve a cluster which is in place for all jobs, it therefore could be said that Lean MapReduce is at best 75% as efficient as established MapReduce implementations, on the time performance axis. Depending on deadline requirements, this may be deemed sufficient or otherwise.

Considering the principles which should be met to satisfy "Leanness" in the context of this thesis (see 4.1, two of them were demonstratively met: eliminate waste and deliver fast. This was achieved through the use of adhoc clusters to which resources were provisioned using a Just-in-Time approach, and prototype B being able theoretically achieve 75% time efficiency of established MapReduce implementations.

7.1.3 Reproducibility of Prototypes

A number of tools were used in building the Lean prototypes and in measuring for performance metrics. A majority of the tools including the Python programming library, GNU stream editor (sed), taskset, and a host of others, are free and available out-of-the-box with most Linux distributions including Ubuntu 14.04. A few of the tools had to be installed (freely) - such as Puppet and Apache2 - while some of the tools had to be extended with third-party plug-ins to enable specialized functionality - among which are the Python paramiko remote access and puppet ssh keys generator modules. Recreating the environment on an Openstack cloud platform is fairly straightforward following the high-level steps set out in the design

phase, and using the code formations described in the implementation phase. On a non-Openstack platform such as a single physical server, the full potential of MLN might not be brought to bare for automated cluster roll-out; therefore, alternate configuration options may have to be used to arrive at the same cluster behavior. This may however affect performance measurements.

7.2 Implementation Challenges

Given that the project work culminated in working prototypes, the development process did not go without some challenges. These are outlined in the subsections below.

7.2.1 Integrating IncludeOS for Map Tasks Isolation

Top among the challenges experienced was the integration of IncludeOS instances into Lean MapReduce clusters as mapper nodes. IncludeOS was to be used in overcoming and simplifying the administrative overhead of determining the optimal number of task slots for each node (scheduler or supervisor), as the execution of each such application would be well isolated in each IncludeOS instance. Its lightweight nature also implies negligible kernel-size resource overhead. However, due to technical complications, which were yet to be understood at the time of writing this section of the report, a substantial percentage of experiments involving IncludeOS did not reach completion. As a workaround, a simple simulator was developed to generate a result similar to what was expected of an IncludeOS-based mapper.

7.2.2 Programming Complexity

Writing code for the functionality of the Lean prototypes was equally challenging. The prototypes were created from scratch and were facilitated by over a 1000 lines of a combination of C++ and Python code, across multiple files. There was a lot of trial-and-error, leading to a lot of debugging. The code covered map (C++) and reduce (Python) applications - which were the same in all occurrences - as well as files for preparing the compute environment (Python) which had to be subtly modified per prototype. The code for preparing new supervisors, setting up tasks and executing same tasks on that host was particularly challenging in A, as modifications had to be made in the code versions executed at each new level (new supervisor) reached in the cluster - a different version of the code file had to passed to a new supervisor. For prototype B, this was less challenging: the same code was simply replicated albeit with unique configuration parameters, and passed on to each new supervisor.

7.2.3 Control over Test Infrastructure

For testing, infrastructure sourced from Alto Openstack cloud had to be relied on entirely. During the process, wide variations were observed. Although multiple iterations of each experiment category were executed to obtain a fair

mean value, some outliers - especially left-oriented extremas - indicated that the system could perform better than majority of the random samples suggested in some categories. This could not be fixed due to lack of administrative access to the cloud infrastructure management layer. It would be interesting to see whether better performance can be obtained on a platform where administrative access is granted, using the same machine specifications as used on Alto cloud.

7.2.4 Implementing Planned Reliability and Cloudbursting Features

To bring Lean MapReduce into conformity with the fundamental design tenets of MapReduce, fault-tolerant features were planned for prototypes A and B. These could not be implemented due to time constraint. Further testing of the prototypes to obtain results for metrics (see fifth paragraph of preceding section) which were discovered to be relevant late into the project, was suspended for the same reason. In the same vein, the potential of stretching a cluster over multiple clouds for instances of cloud-bursting could not be explored.

7.3 Improvements to Lean MapReduce Designs

In this section, potential improvement which could be made to the Lean MapReduce designs are presented.

7.3.1 Adaptable Features from Related Work

The approach adopted in addressing the problem statement question is novel to the best of our knowledge, in that a compute cluster is not pre-provisioned for jobs. A minimal compute resource footprint (impressed by a singleton scheduler) is maintained outside of job windows, while the required capacity of the cluster is determined and requisitioned as an aspect of the execution window. The approach is however driven in part by the universal objective of optimal and efficient resource usage.

BASE (see 2.11.2) employs *resource stealing* in achieving a similar goal; however, BASE aims to improve the utilization levels for active jobs in extant Hadoop clusters. Lean MapReduce circumvents speculative resource provisioning, a pre-provisioning strategy which is not addressed but rather is implied in BASE and by extension, Hadoop. On its part, Lean MapReduce shows a weakness on the specific issue which BASE is designed to solve: using up resources assigned to empty task slots for the operations of currently running tasks. Although Lean MapReduce jobs are one-window only and do not have the notion of idle task slots, there is the propensity for it to let resources "idle" when the available resources at a node outsizes its assigned workload. Resource idling would be particularly substantial in long-running jobs. For this purpose, resource stealing could be employed in Lean MapReduce to engage the full capacity of a system, although this would entail introducing component parts which support such a behavior; IncludeOS-based mappers will have to be substituted with a resource-elastic compute entity like docker (with low-level manipulation of cgroups settings). Cgroups, short for

control groups, is a Linux kernel utility which facilitates resource management of a collection of processes.

7.3.2 Consolidating the Strong Features of Lean MapReducee Designs

The Lean MapReduce prototypes demonstrated individual advantages (section 7.1, 2^{nd} paragraph): where Lean prototype A constantly required one less resource unit than prototype B, prototype B delivered results in considerably better time than prototype A as the number of needed resource sets tended to infinity. Prototype A also showed a superior TTC for non-threshold-crossing workloads owing to the fact that the scheduler was able to accommodate and execute all map and reduce tasks in its task slots. Based on this, a hybrid of both prototypes could be proposed, where the right to host task slots at the scheduler (in force in prototype A) is introduced to prototype B. The original plan for prototype B was for it to be initiated from a scheduler which is used for other computer intensive but non-MapReduce operations, while prototype A would grow out of a server (a scheduler) meant exclusively for orchestrating a Lean MapReduce cluster. The hybrid design would make the resource needs of a job asymptotic to that of a prototype A workload as well as deliver the TTC expected of the minimal compute resource set needed by the same prototype, and also achieve TTC which approaches the expected value in prototype B.

In delivering intermediate results to a local reduce task, tcp connections were established from mapper nodes to a listening TCP server on the host, and the results streamed across. These results were then written to a file on the host system. Although the resource and time impact of the process was not measured - and should be in future work - it stands to reason that given the proximity of the communicating entities, that this two-legged strategy involves a redundancy. By relying on a functionality which enables IncludeOS to write objects to its memory-resident virtual file system - provided by an external disk device called a memdisk - the need for local TCP connections would be eliminated and provide an opportunity to shorten TTC.

7.4 Future Work

The two prototypes showcased in the course of exploring a novel way of driving optimal and efficient resource utilization for timely delivery of results in processing of big data, have provided initial answers which could be built upon. To improve enhance their reliability the proposed fault-tolerance features should be implemented. Following this, the tenant-constrained design advanced in 6.3 as an iteration of prototype A is a veritable next step. The design opens up a new pathway which could lead to the amalgamation of cloud tenants quotas from disparate cloud providers into a logical Lean MapReduce cluster. Such an arrangement would require payed access to compute resources on a public cloud. With the tendency towards energy efficiency in the modern era, cloud service prices are predominantly dictated by the cost of generating power. On this frontier, wind and

solar generated electricity are strongly correlated with some of the cheapest prices. The management logic of the new Lean design could co-opt logic for monitoring meteorological reports as well as constantly surveilling the pricing options for the cheapest rates on offer by the diversity of cloud providers. This would entail that the list of accessible tenants is constantly updated across an active cluster.

Such a design should have a better TTC advantage over original Lean prototypes A and B, since the need to increase resource capacity beyond a tenant's quota would be the exception rather than the norm, which results in nominal or no spawn time being expended - that is if the initial size of the tenant is relatively large. Even in cases where the tenant quota starts out small and is dynamically stretched, the dominant cluster preparation factor will be time spent creating mapper nodes rather than supervisors. The relative speed with which Unikernel VM instances can be built, as against general purpose operating systems, would still offer a performance boost over what the performance observed of the Lean prototypes in their present state.

The potential for live migration of entire clusters or component mappers could also be looked at, particularly in a situation where a more favourable pricing option is discovered. The absence of a underlying distribution file system means that the cluster is loosely coupled with the nodes enjoying some degree of autonomy, and that the mappers can be shuffled around to some degree. In this scenario, the mapper could be imbued with intelligence to self migrate to a newly acquired tenant if the price of the present tenant is deemed less favorable. The overall cost of long-running, compute intensive jobs could be made very cheap using such a scheme.

The model would hold an economic appeal to small and medium scale organizations whose ability to acquire their desired big data computing capacity is ordinarily constrained by limited budgets. It holds the potential of optimising value for money in the short and long term. It would also be a boon to start-up big data scientists, as well as researchers in developing countries, who would have cheaper access to compute resources for sustainable computing.

Chapter 8

Conclusion

The aim of this project was to investigate a way to design an adhoc MapReduce architecture which optimally and efficiently utilizes system resources to reliably process big data workloads and deliver results in a timely manner.

The key elements of the problem statement were addressed through the development of two alternative designs - Lean Alternatives A and B - based on 2 distinct adaptations of the B-tree abstract data structure: a flat tree structure, and a chain structure with hanging leaves. These guided the development of two prototypes, named Lean Prototypes A and B. The prototypes were experimented on (to obtain time and resource-oriented performance measurements), analyzed and compared.

Through analysis of the experiment results, it became apparent that the designs facilitated optimality and efficiency of resource utilization - a direct implication of the dynamic way in which Lean MapReduce-based clusters are grown into place, using compute resource capacity of provisioned nodes as extension thresholds. "Adhocness" was inherent feature due to the Just-in-Time and one-window resource provisioning approach to processing workloads. Comparative timeliness between both designs was established, where the better performing design for workloads requiring more than 2 resource sets (Lean alternative B) showed that it could theoretically achieve about 75% of the time efficiency of established Mapreduce implementations which make use of persistent compute clusters.

Appendix

All the scripts developed and used in this thesis have been uploaded to an on-line repository where they may be accessed with the following hyperlink: *https://iamiam@bitbucket.org/iamiam/leanmapreduce.git*

Bibliography

[1] James Manyika et al. “Big data: The next frontier for innovation, competition, and productivity.” In: (2011).

[2] Min Chen, Shiwen Mao, and Yunhao Liu. “Big Data: A Survey.” In: *Mobile Networks and Applications* 19.2 (2014), pp. 171–209. ISSN: 1572-8153. DOI: **10.1007/s11036-013-0489-0**. URL: **http://dx.doi.org/10.1007/s11036-013-0489-0**.

[3] Jinchuan Chen et al. “Big data challenge: a data management perspective.” In: *Frontiers of Computer Science* 7.2 (2013), pp. 157–164.

[4] Frank Würthwein. *Computing at the Large Hadron Collider*. 2013. URL: **https://www.rcc.uq.edu.au/filething/get/1140/W%C3%BCrthwein.pdf** (visited on 01/26/2016).

[5] Cian O’Luanaigh. *Computing*. 2012. URL: **http://cds.cern.ch/record/1997391** (visited on 01/26/2016).

[6] Adrian Diaconescu. *The US Government is building the fastest computer in the world*. 2014. URL: **http://www.digitaltrends.com/computing/the-us-will-build-the-worlds-fastest-supercomputers/** (visited on 01/28/2016).

[7] Christopher J Trezzo. “Continuous MapReduce: an architecture for large-scale in-situ data processing.” In: (2010).

[8] Margaret Rouse. *high-performance computing (HPC)*. 2007. URL: **http://searchenterpriselinux.techtarget.com/definition/high-performance-computing** (visited on 02/06/2016).

[9] Carlos P Sosa. *HPC: Past, Present, and Future*. 2011. URL: **http://www.msi.umn.edu/~cpsosa/HPC_Past_Preset_Futurre_web.pdf** (visited on 02/06/2016).

[10] Douglas Eadline. *HPC for dummies*. 2009. URL: **http://hpc.fs.uni-lj.si/sites/default/files/HPC_for_dummies.pdf** (visited on 02/07/2016).

[11] IBM, Paul Zikopoulos, and Chris Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. 1st. McGraw-Hill Osborne Media, 2011. ISBN: 0071790535, 9780071790536.

[12] Gil Press. *A Very Short History Of Big Datas*. 2013. URL: **http://www.forbes.com/sites/gilpress/2013/05/09/a-very-short-history-of-big-data/#5a016bb255da** (visited on 02/03/2016).

[13] FollowtheData. *Data size estimates*. 2014. URL: **https://followthedata.wordpress.com/2014/06/24/data-size-estimates/** (visited on 02/08/2016).

[14] Rick Delgado. *How Car Manufacturers are Using Big Data*. 2014. URL: **https://datafloq.com/read/car-manufacturers-are-using-big-data/1204** (visited on 02/07/2016).

[15] DeepWeb. *STRUCTURED VS. UNSTRUCTURED DATA*. 2012. URL: **http://www.brightplanet.com/2012/06/structured-vs-unstructured-data/** (visited on 02/07/2016).

[16] Pierre Dorion. *What is unstructured data and how is it different from structured data in the enterprise?* 2007. URL: **http://searchstorage.techtarget.com/feature/What-is-unstructured-data-and-how-is-it-different-from-structured-data-in-the-enterprise** (visited on 02/09/2016).

[17] Tim McGuire et al. *Why Big Data is the new competitive advantage*. 2012. URL: **http://iveybusinessjournal.com/publication/why-big-data-is-the-new-competitive-advantage/** (visited on 02/09/2016).

[18] Hugh J Watson. "Tutorial: Big data analytics: Concepts, technologies, and applications." In: *Communications of the Association for Information Systems* 34.1 (2014), pp. 1247–1268.

[19] Steve LaValle et al. "Big data, analytics and the path from insights to value." In: *MIT sloan management review* 52.2 (2011), p. 21.

[20] Scott Lowe. *Is data an asset or a liability?* 2009. URL: **http://www.techrepublic.com/blog/tech-decision-maker/is-data-an-asset-or-a-liability/** (visited on 02/08/2016).

[21] Doug Henschen. *Big Data Debate: End Near For ETL?* 2012. URL: **http://www.informationweek.com/big-data/big-data-analytics/big-data-debate-end-near-for-etl/d/d-id/1107641?** (visited on 02/11/2016).

[22] Intel. *Big Data meets High Performance Computing*. 2014. URL: **http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/big-data-meets-high-performance-computing-white-paper.pdf** (visited on 02/11/2016).

[23] Isaac Lopez. *IDC talks convergence in High Performance Data Analytics*. 2013. URL: **http://www.datanami.com/2013/06/19/idc_talks_convergence_in_high_performance_data_analysis/** (visited on 02/14/2016).

[24] Margaret Rouse. *big data analytics*. 2014. URL: **http://searchbusinessanalytics.techtarget.com/definition/big-data-analytics** (visited on 02/10/2016).

[25] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: **10.1145/1327452.1327492**. URL: **http://doi.acm.org/10.1145/1327452.1327492**.

[26] Avita Katal, Mohammad Wazid, and RH Goudar. "Big data: issues, challenges, tools and good practices." In: *Contemporary Computing (IC3), 2013 Sixth International Conference on*. IEEE. 2013, pp. 404–409.

[27] Jorda Polo et al. "Resource-aware adaptive scheduling for mapreduce clusters." In: *Middleware 2011*. Springer, 2011, pp. 187–207.

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google file system.” In: *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 29–43.

[29] Apache.org. *Welcome to Apache™ Hadoop*. 2016. URL: **http://hadoop.apache.org/** (visited on 02/01/2016).

[30] Hortonworks.com. *HDFS*. 2016. URL: **http://hortonworks.com/hadoop/hdfs/** (visited on 02/20/2016).

[31] saphanatutorial.com. *How YARN Overcomes MapReduce Limitations in Hadoop 2.0*. 2014. URL: **http://saphanatutorial.com/how-yarn-overcomes-mapreduce-limitations-in-hadoop-2-0/** (visited on 02/28/2016).

[32] Magaret Rouse. *Apache Hadoop YARN (Yet Another Resource Negotiator)*. 2013. URL: **http://searchdatamanagement.techtarget.com/definition/Apache-Hadoop-YARN-Yet-Another-Resource-Negotiator/** (visited on 03/11/2016).

[33] David Marshall. *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*. 2007.

[34] Oracle Corporation. *Introduction to Virtualization*. 2011. URL: **https://docs.oracle.com/cd/E20065_01/doc.30/e18549/intro.htm** (visited on 02/14/2016).

[35] Michael Fenn et al. “An evaluation of KVM for use in cloud computing.” In: *Proc. 2nd International Conference on the Virtual Computing Initiative, RTP, NC, USA*. 2008.

[36] Peter Mell and Tim Grance. “The NIST definition of cloud computing.” In: (2011).

[37] Paulo Neto. “Demystifying cloud computing.” In: *Proceeding of Doctoral Symposium on Informatics Engineering*. 2011.

[38] Alfred Bratterud. *IncludeOS*. 2015. URL: **https://github.com/hioa-cs/IncludeOS/wiki** (visited on 03/15/2016).

[39] A. Bratterud et al. “IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services.” In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. Nov. 2015, pp. 250–257. DOI: **10.1109/CloudCom.2015.89**.

[40] E Knuth Donald. “The art of computer programming.” In: *Sorting and searching* 3 (1999), pp. 426–458.

[41] Dave Perrett. *CompSci 101 - Big-O Notation*. 2010. URL: **http://www.daveperrett.com/articles/2010/12/07/comp-sci-101-big-o-notation/** (visited on 05/02/2016).

[42] Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley, 2013.

[43] William Shields. *Plain English Explanation of Big O Notation*. 2009. URL: **http://www.cforcoding.com/2009/07/plain-english-explanation-of-big-o.html** (visited on 05/02/2016).

[44] Anca Iordache et al. “Resilin: Elastic MapReduce over multiple clouds.” In: *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE. 2013, pp. 261–268.

[45] Zhenhua Guo et al. “Improving resource utilization in mapreduce.” In: *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE. 2012, pp. 402–410.

[46] Kyrre M Begnum. “Managing Large Networks of Virtual Machines.” In: *LISA*. Vol. 6. 2006, pp. 205–214.

[47] Kelly Waters. *All About Agile*. 2010. URL: **http://www.allaboutagile.com/lean-principles** (visited on 05/11/2016).