DEGREE PROJECT IN COMPUTER SCIENCE AND COMPUTER ENGINEERING,
SECOND LEVEL
STOCKHOLM, SWEDEN 2016

A Coordination Framework for Deploying Hadoop MapReduce Jobs on Hadoop Cluster

ANITHA RAJA

KTH ROYAL INSTITUTE OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY

A Coordination Framework for Deploying Hadoop MapReduce Jobs on Platforms

Anitha Raja

2016-11-27

Master's Thesis

Examiner
Prof. Gerald Q. Maguire Jr.

Academic Supervisor
Assoc. Prof. Anders Västberg

Industrial Supervisor
Mr. Yue Lu (Ericsson AB)

KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
Department of Communication Systems
SE-100 44 Stockholm, Sweden

Abstract

Apache Hadoop is an open source framework that delivers reliable, scalable, and distributed computing. Hadoop services are provided for distributed data storage, data processing, data access, and security. MapReduce is the heart of the Hadoop framework and was designed to process vast amounts of data distributed over a large number of nodes. MapReduce has been used extensively to process structured and unstructured data in diverse fields such as e-commerce, web search, social networks, and scientific computation. Understanding the characteristics of Hadoop MapReduce workloads is the key to achieving improved configurations and refining system throughput. Thus far, MapReduce workload characterization in a large-scale production environment has not been well studied.

In this thesis project, the focus is mainly on composing a Hadoop cluster (as an execution environment for data processing) to analyze two types of Hadoop MapReduce (MR) jobs via a proposed coordination framework. This coordination framework is referred to as a workload translator. The outcome of this work includes: (1) a parametric workload model for the target MR jobs, (2) a cluster specification to develop an improved cluster deployment strategy using the model and coordination framework, and (3) better scheduling and hence better performance of jobs (i.e. shorter job completion time). We implemented a prototype of our solution using Apache Tomcat on (OpenStack) Ubuntu Trusty Tahr, which uses RESTful APIs to (1) create a Hadoop cluster version 2.7.2 and (2) to scale up and scale down the number of workers in the cluster.

The experimental results showed that with well tuned parameters, MR jobs can achieve a reduction in the job completion time and improved utilization of the hardware resources. The target audience for this thesis are developers. As future work, we suggest adding additional parameters to develop a more refined workload model for MR and similar jobs.

Keywords

Hadoop, Workload Characterization, Parametric Modeling, Coordination framework, OpenStack, Workload deployment

Sammanfattning

Apache Hadoop är ett öppen källkods system som levererar pålitlig, skalbar och distribuerad användning. Hadoop tjänster hjälper med distribuerad data förvaring, bearbetning, åtkomst och trygghet. MapReduce är en viktig del av Hadoop system och är designad att bearbeta stora data mängder och även distribuerad i flera leder. MapReduce är använt extensivt inom bearbetning av strukturerad och ostrukturerad data i olika branscher bl. a e-handel, webbsökning, sociala medier och även vetenskapliga beräkningar. Förståelse av MapReduces arbetsbelastningar är viktiga att få förbättrad konfigurationer och resultat. Men, arbetsbelastningar av MapReduce inom massproduktions miljö var inte djup-forskat hittills.

I detta examensarbete, är en hel del fokus satt på ”Hadoop cluster” (som en utförande miljö i data bearbetning) att analysera två typer av Hadoop MapReduce (MR) arbeten genom ett tilltänkt system. Detta system är refererad som arbetsbelastnings översättare. Resultaten från denna arbete innehåller: (1) en parametrisk arbetsbelastningsmodell till inriktad MR arbeten, (2) en specifikation att utveckla förbättrad kluster strategier med båda modellen och koordinations system, och (3) förbättrad planering och arbetsprestationer, d.v.s kortare tid att utföra arbetet. Vi har realiserat en prototyp med Apache Tomcat på (OpenStack) Ubuntu Trusty Tahr som använder RESTful API (1) att skapa ”Hadoop cluster” version 2.7.2 och (2) att båda skala upp och ner antal medarbetare i kluster.

Forskningens resultat har visat att med vältrimmad parametrar, kan MR arbete nå förbättringar dvs. sparad tid vid slutfört arbete och förbättrad användning av hårdvara resurser. Målgruppen för denna avhandling är utvecklare. I framtiden, föreslår vi tilläggning av olika parametrar att utveckla en allmän modell för MR och liknande arbeten.

Nyckelord

Hadoop, Arbetsbelastning Karakterisering, Parametrisk Utformning, Koordinations system, OpenStack, Arbetsbelastnings Utplacering

Acknowledgments

I would like to offer my sincere thanks to my thesis supervisor Associate Prof. Anders Västberg and my thesis examiner Prof. Gerald Q. “Chip” Maguire Jr. (both of the School of Information and Communication Technology, KTH Royal Institute of Technology, Stockholm, Sweden) for their valuable suggestions and indispensable recommendations.

My heartfelt gratitude to my thesis supervisor at Ericsson Mr. Yue Lu, my manager Ms. Azimeh Sefidcon, and my advisor Mr. Joao Monteiro Soares for sharing their valuable ideas and also personally helping me settle down in Ericsson AB and successfully complete my work.

My gratitude to Ms. May-Britt Eklund Larsson for the continuous support she provided during my course work at KTH.

Finally, I thank my parents, husband, and in-laws for their uninterrupted affection and moral support throughout the period of my study and all through my life. I would like to thank friends, family members, and everyone else who supported and inspired me during my whole life.

Stockholm, November 2016
Anitha Raja

Table of contents

List of Figures

List of Tables

List of acronyms and abbreviations

App Mstr	Application Master
API	Application programming interface
HDFS	Hadoop Distributed File System
HMM	Hidden Markov Model
JCT	Job Completion Time
MR	MapReduce
NM	Node Manager
REST API	Restful API
RM	Resource Manager
S3	(Amazon) Simple Storage Service
vCores	virtual cores
VM	Virtual Machine
WL	Workload
XML	eXtensible Markup Language

1 Introduction

This chapter describes the specific problem this thesis addresses, the context of the problem, the goals of this thesis project, and outlines the structure of this report.

The cloud computing concept has been researched over many years with different dimensions, especially with regard to its pay per use and flexible business models. Many cloud service providers need to process huge amounts of data. At present, the amount of data that is processed by a single user has increased from Terabytes to Petabytes and the expected future demand is for much larger amounts of data (this is often referred to as “Big Data”). One solution for data mining with such huge amounts of data is MapReduce systems. MapReduce systems are the main framework used today for processing big data. This framework minimizes communication and data movement by performing computation local to the data. There are two major steps in MapReduce: map and reduce. The map step divides the workload into smaller tasks and distributes them to worker nodes as a map task. The reduce step gathers output data from each worker node and creates the final job output. Additionally, it is highly desirable to predict workloads in advance, so that a series of processing steps (with dependencies) can be scheduled and executed in the appropriate order to deliver refined data by the required time.

In this thesis project we focus on workload modeling to provide an interpretation layer to translate input user workloads (WLs) into a specification for deployment (of these WLs). Given this specification we develop a deployment strategy on top of a logical server. We want to identify *implicit* characteristics of the WLs that will assist us in finding a good deployment strategy. We evaluate the resulting strategy using OpenStack [1], an open source cloud computing software that controls large pools of compute, storage, and networking resources. OpenStack is used to realize a logical cluster for a given WL.

Figure 1-1 is a system diagram of a cloud system. The input to the data center are WLs. In this thesis project, these WLs are translated into a more refined WL specification with extra parameters in order to deploy the WLs in an optimized manner.

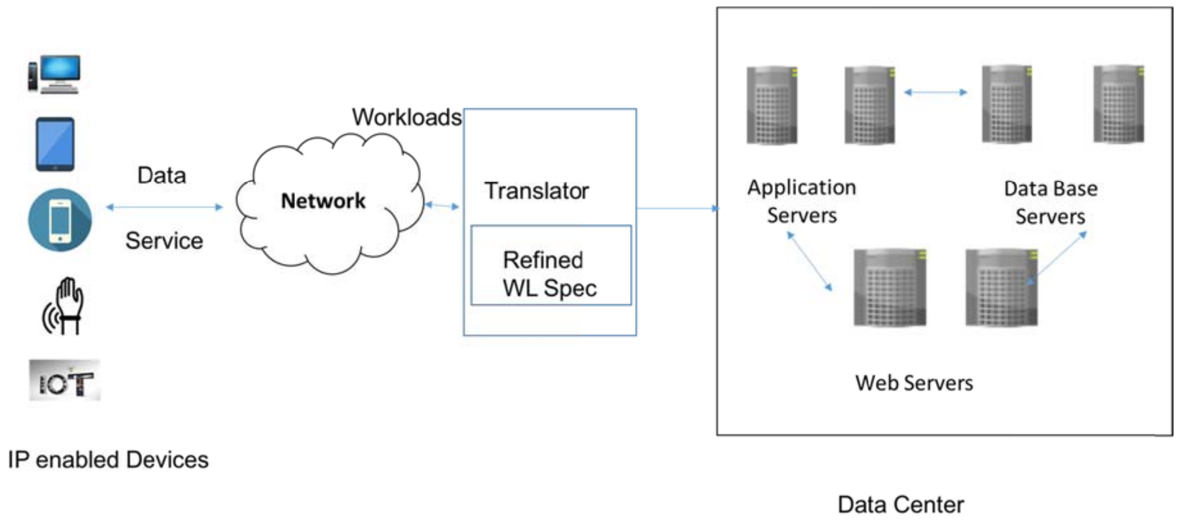

Figure 1-1: System Overview

1.1 Background

Hadoop MapReduce (MR) [2] is a programming framework for parallel processing which can be used to write applications that will process huge amounts of data. The MapReduce Next generation architecture called YARN Cluster [3] has separated the two main functions of a Job Tracker (resource management and job scheduling) into two separate components (shown in Figure 1-2). This is done by having a global Resource Manager (RM) and a per-application Application Master (App Mstr). The RM arbitrates requests for resource allocations by the applications running in the system. RM has two components: (1) a scheduler that allocates resources among the running applications and (2) an App Mstr that accepts job submissions, negotiates an initial container[*] for application execution, and provides services to restart the App Mstr container if it fails. The RM communicates with a Node Manager (NM) to track the allocation of containers. A NM is a per machine slave that launches applications and their containers and monitors resource usage and availability of resources. The App Mstr is responsible for negotiation and tracking of the resource containers allocated by the scheduler.

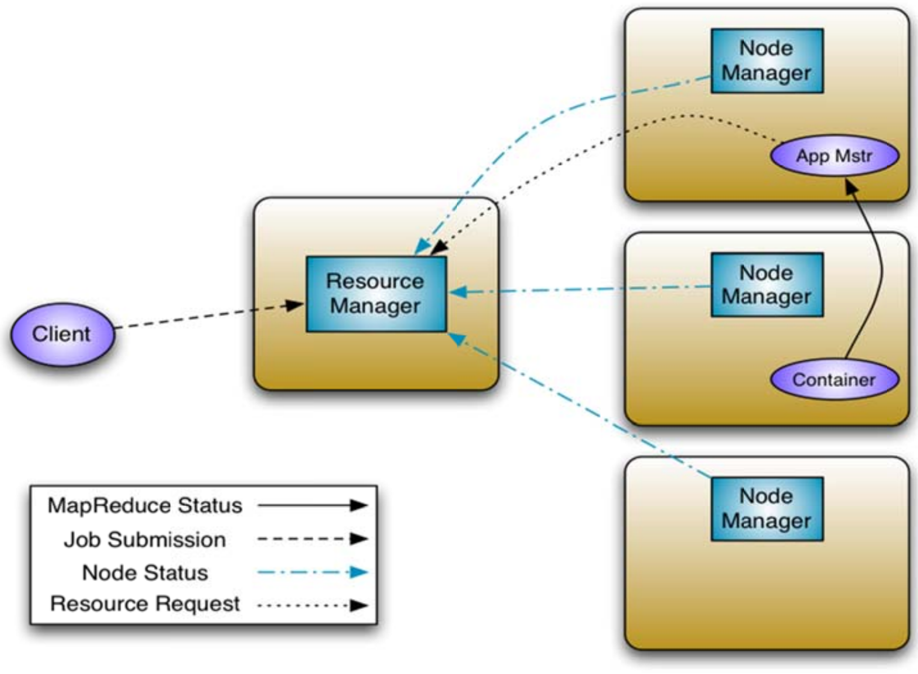

Figure 1-2: Hadoop architecture

When a developer submits a MapReduce job in YARN, the WL consists of the following: a configuration file, a jar file with the implementation of MapReduce, the input directory path where the files to be processed are stored, and the output directory path where the results will be stored.

In this thesis project we will derive a more refined specification from each WL in order to better describe each Hadoop MR job. This improved description will facilitate our development of a deployment strategy to efficiently deploy the WLs on logical platforms (specifically a logical cluster).

[*] A container is a unit of allocation to execute an application specific task.

1.2 Problem definition

This thesis addresses three problems:

1. How to develop a parametric model to describe a Hadoop MR job?
2. How to develop a deployment strategy?
3. How to develop a coordination framework to compose a logical cluster and deploy the MR workload?

1.3 Purpose

The purpose of this thesis project is to develop a parametric model and a prototype coordination framework to realize the developed model in order to dynamically compose a logical cluster for the incoming MR WL.

1.4 Goals

The goal of this project has been divided into the following four sub-goals:

1. Characterize WLs and then refine the WL specifications to facilitate deployment;
2. Develop WL deployment strategies;
3. Find a light-weight means to perform logical server/cluster composition (this provides the coordination framework); and
4. Demonstrate the achievement of the three earlier goals through a prototype implementation of a coordination framework.

1.5 Research Questions

The main research question for this thesis is: "**How to deploy diverse Hadoop MR workloads on a data center?**"

This question leads to the following sub-questions:

Q1 What are the characteristics of WLs?

Q2 What is a suitable parametric model for these WLs?

Q3 What deployment strategy performs best in handling Hadoop MR WLs *within* a data center?

1.6 Research Methodology

We use quantitative methods in this research to understand a Hadoop job. We also use qualitative methods to understand deployment strategies and optimization techniques when setting up a logical cluster.

1.7 Delimitations

We concentrate on Hadoop MR jobs for our WL analysis. Other types of WLs are not analyzed in this research. For simplification, we assume the WL information is provided either by users or available as prior knowledge *before* modeling. Although we will do some extra work to represent

WLs at the level of resource demands at the task level, we sought to minimize the number of parameters. Also, we focus only on *dimensioning* the logical cluster's size (in number of nodes) in order to limit the scope of this thesis project.

1.8 Structure of the thesis

The layout of the rest of this thesis is as follows: The next chapter presents relevant background information about a distributed cloud data center and its problem areas. Chapter 3 describes the methodology used to solve the problem. Chapter 4 discusses and evaluates the results, while Chapter 5 analyzes these results. The final chapter provides the conclusion of this thesis and suggests potential future work.

2 Background

This chapter provides basic background information about cloud computing. Additionally, this chapter describes existing workload modeling techniques used in cloud computing.

Cloud computing enables large-scale services without requiring a large up-front investment. In contrast, the traditional computing model has two common problems: under provisioning and over provisioning of resources. An infrastructure for cloud computing is called a "cloud". In cloud computing under/over provisioning is avoided by *dynamically* provisioning resources.

There are three categories of cloud services: infrastructure as a service, platform as a Service, and software as a service. Additionally, there are four cloud deployment models: public cloud, private cloud, community cloud, and hybrid cloud. Public clouds are owned by cloud service providers who charge on the basis of resource usage. These clouds are characterized by providing a homogeneous infrastructure, common policies, shared resources and multi-tenancy, and leased or rental infrastructure. Examples of public clouds are Amazon's AWS/EC2 [4], Microsoft's Azure [5], Google's compute Engine [6], and Rackspace [7]. In contrast, private clouds are owned and operated by a single organization. Their basic characteristics include heterogeneous infrastructure, customized policies, dedicated resources, and in-house infrastructure. Examples of software for realizing private clouds include Eucalyptus Systems [8], OpenNebula [9], and OpenStack.

The cloud computing paradigm has spread widely in the market and become successful in the past few years. Although the adoption of cloud computing and its success has been rapid, characterizing cloud WLs is still not (yet) completely clear. Understanding WLs has become an important research area for those seeking to improve the performance of cloud systems. Improving the performance of cloud systems is necessary since the cloud computing paradigm is becoming a major environmental polluter due to consuming enormous amounts of energy (especially electrical power) [10] Additional reasons for optimization are that demand has continued to grow exponentially, while performance (of the computing, network, and storage) has not grown at such a rate, and because cloud infrastructures have portions of the infrastructure that are underutilized.

Cloud WLs frequently and repeatedly originate from web services, such as search and retrieval queries, online documentation, data mining (such as MapReduce jobs), etc. In practice, many WLs have short duration and are submitted at very frequent intervals. These WLs are frequently latency-sensitive WLs, hence their scheduling has to be carefully addressed. In contrast, batch WLs, some of which are computation intensive (i.e., with greater processing requirements - but smaller storage requirements), memory intensive (larger storage requirements - but lesser processing requirements), or require both greater processing and larger storage. A mix of batch WLs and latency-sensitive workloads lead to mixed WLs. These mixes arise from most online services as these service involve interactive servicing of user requests *and* processing (often a large amount of) data in the background. Deploying such mixed workloads on a data center requires a good understanding of the diverse WLs and a suitable deployment strategy.

Apache Hadoop [11] is open source software that provides reliable, scalable, and distributed computing. It is a framework that permits distributed storage and distributed processing of huge data sets across computer clusters using a simple programming model. The basic Hadoop components are Hadoop Distributed File System (HDFS), where the data is stored, and MapReduce, which processes the data stored in HDFS. HDFS is a distributed file system that provides built in redundancy, scalability, and reliability. HDFS is the foundation of the Hadoop stack. On top of this is the MapReduce processing framework. This framework is responsible for resource management and data processing in the cluster. On top of MapReduce, all kinds of applications are used, such as Pig [12] (a platform for analyzing large data sets and parallel processing), Hive [13] (a data warehouse platform that provides large dataset management using SQL), and Spark [14] (a fast and general engine for processing large scale data). These applications manipulate the data through the MapReduce process on top of the distributed file system.

The Hadoop cluster building blocks are as follows:

NameNode	The NameNode is the centerpiece of HDFS as it stores file system metadata and is responsible for all client operations.
Secondary NameNode	The Secondary NameNode synchronizes its state with the active NameNode in order to provide fast failover if the active NameNode goes down.
ResourceManager	The global ResourceManager is a scheduler that directs the slave NodeManager daemons to perform the low-level I/O tasks.
DataNodes	The DataNodes (also known as slaves) store data in the HDFS. These nodes host a NodeManager process (i.e., acts as a slave NodeManager daemon) which performs the actual processing of the data stored in the nodes. Each NodeManager process communicates with the ResourceManager to get instructions about how to process the local data.
History Server	The History Server provides REST APIs for the end users to get a job's status and other job information.

Hadoop is a black box which can accept various types of jobs. The Hadoop framework contains over 190 parameters [15]–[18] that can be configured. Some of these parameters play a significant role in the performance of a Hadoop job. However, it is a challenging and time consuming task to manually identify and configure each of these performance tuning parameters for each incoming job. By developing a parametric model, it is possible to find the optimum values for these parameters (or at least a subset of them). Unfortunately, creating a mathematical model that represents the correlation among these parameters is extremely difficult.

This thesis project focuses on Hadoop. The main goal is to understand different WLs behaviors in order to develop a parametric model to describe Hadoop MR jobs. The first step was to propose a parametric model for Hadoop MR jobs and then to develop a deployment strategy in order to set up the Hadoop cluster based on the refined specifications of the incoming WLs.

Section describes what a Hadoop MR job is. Section describes the organization of a disaggregated data center. Section 2.3 describes the concept of logical server platforms. Section 2.4 concerns characterization of WLs. Finally, Section 2.5 contains some additional background information and summarized related work.

2.1 A Hadoop MR job

A MR job consists of a map function, a reduce function, and input data[*]. First, the input data is split into multiple splits. Then, for each map split, a "map task" runs which applies the map function to the map split. The resulting output of each map task is a collection of key-value pairs. The output of all map tasks are shuffled, that is, for each distinct key in the map output, a collection is created containing all corresponding values from the map output. For each key-collection resulting from the shuffle phase, a "reduce task" runs which applies the reduce function to the collection of values. The resulting output is a single key-value pair. The aggregation of all key-value pairs resulting from the reduce phase is the output of the MR job.

[*] The data is typically stored in a *Hadoop Distributed File System* (*HDFS*).

2.2 Disaggregated data center

Traditional data centers have a relatively fixed computing infrastructure that is used for the data center's operation. Having all of the resources in one place enables high utilization during both peak and non-peak demand conditions (as during the later period portions of the resources can be powered off). Data center operators invest in servers which are kept in an active pool to ensure that their customers have sufficient resources even during high demand conditions. A disaggregated data center separates the resource components, such as CPU, Memory, and I/O, into logical pools of these resources. The aim of this separation by resource type is to offer greater flexibility while ensuring more optimal resource utilization (as resources are less likely to be stranded in a discrete physical server that is allocated for computing, but has limited memory needs – while another physical server could utilize this memory). Table 2-1 summarizes the characteristics that distinguish a disaggregated data center from a traditional data center.

Table 2-1: Disaggregated data center characteristics

Characteristic	Description
Disaggregated resources	CPU, Memory, and I/O resources are separated into pools of a single type of resource.
Composing systems	Using these disaggregated resources, the data center operator can compose different sized and configured logical clusters (systems).
On-demand resource creation	Depending on a WL's demand a suitable logical cluster can be composed from the disaggregated resources.
Interconnection	Unfortunately, the physical distance between disaggregated resources are usually several meters, hence communication between these components is much slower than in a traditional data center. High speed interconnection fabrics are used for communication among the components.

2.3 Logical server platforms

In a disaggregated data center, there are multiple resource pools, containing resources provided by the different server blades that are are mounted in racks. This environment allows more fine-grained allocation of resources than a traditional blade-oriented architecture. From these resources we compose a logical server based upon storage/computing/networking from one or more physical server blades. Ideally we should select memory resources from those server blades that have the lowest I/O delay to the other server blades which provide the logical cluster with CPU and networking resources.

2.4 Workload characterization

WL characterization is one of the primary goals of this thesis project. We extracted some of the WL parameters to develop a parametric model that best describes a Hadoop MR job. This characterization serves as a basis to understand what a Hadoop MR jobs look like before actually deploying it in the data center. Also, it is a time consuming process to identify the best performance tuning parameters from Hadoop, as it has over 190 configuration parameters. We identified a few tuning parameters for our parametric model (as described in detail in Section 3.2).

2.5 Background and Related work

Some research has already been done on characterizing cloud WLs [19]–[22]. However, these studies have focused on statistically understanding and recreating computing tasks (e.g., MR tasks) that are scheduled on a cloud.

2.5.1 HCloud

Christina Delimitrou and Christos Kozyrakis proposed HCloud [23], a hybrid provisioning system that determines whether the jobs should be mapped to reserved or on-demand resources based on overall load and resource unpredictability. They showed that HCloud increases performance by 2.1 times that of fully on-demand resources and increases cost efficiency by decreasing cost by 46% compared to fully reserved resources.

2.5.2 HUAWEI HTC-DC

HUAWEI [24] proposed a high throughput data center architecture called HTC-DC which is designed to meet the high throughput demands of big data. HTC-DC supports Petabytes (PB)-level data processing capability, intelligent manageability, high scalability, and high energy efficiency. However, it is still being developed, but it could be a promising candidate in the future.

2.5.3 Energy efficiency for MR WLs

Feng et al. [25] conducted an in depth study of the energy efficiency of MR WLs and identified four factors that affect the energy efficiency of MR. They found that with well tuned system parameters and adaptive resource configurations, MR cluster can achieve both performance improvement and energy efficiency in some instances. However, their solution has to be verified with large cluster sizes.

2.5.4 Actual cloud WLs

Panneerselvam et al. [10] researched actual cloud WLs. They categorized and characterized WLs to help predict user demand using a parametric modeling technique. Using their model, CPU intensive WLs show a higher percentage of prediction errors than memory intensive WLs in experiments conducted by evaluating the performances of two prediction techniques (Markov modelling and Bayesian modelling).

2.5.5 Characterizing and predicting WL in a Cloud with incomplete knowledge of application configuration

Khan et al. [16] introduced a new way to characterize and predict WL in a cloud system when complete application configurations of customers' VMs are unavailable to the cloud providers. They identified repeatable WL patterns within groups of VMs that belong to a cloud customer. They employed a Hidden Markov Model (HMM) to capture the temporal correlations and to predict changes in WL pattern based on co-clusters discovered using clustering. This method showed higher prediction accuracy than traditional methods. However, these studies only examined repeatable WL patterns and did not examine periodic daily patterns.

2.5.6 Statistical analysis of relationships between WLs

Yang et al. [19] proposed a statistical analysis approach to identify relationships among WL characteristics, Hadoop configurations, and WL performance. They applied principal component analysis and cluster analysis to 45 different metrics and revealed that they could accurately predict the performance of MR WLs under different Hadoop configurations. However, these studies show that their proposed predictive model can be difficult to apply when there is dynamic profiling of Hadoop configurations for optimizing workloads.

2.5.7 Analysis of virtualization impact on resource demand

Wang et al. [26] conducted an in depth analysis of WL behavior from web applications, specifically the Rice University Bidding System (RUBiS) benchmark application. They also analyzed the impact of virtualization on the resource demands of cloud applications by profiling WL dynamics on both virtualized and non-virtualized servers. Their experimental comparison results help in predicting Service Level Agreement (SLA) compliance, evaluating the application's performance, and deciding upon the right hardware to support applications. In the future, they plan to characterize other cloud application's WL, such as big data using the MapReduce framework.

2.5.8 Methodology to construct a WL classification

Mishra et al. [20] developed a methodology to classify WLs and applied it to the Google Cloud Backend. They used the concept of qualitative coordinates to gain several insights into the Google Cloud Backend. Their results can guide system designers to improve task scheduling and capacity planning. In the future, they plan to extend their study to consider job constraints and to address task arrival process characterization.

2.5.9 Matching diverse WL categories to available cloud resources

Mulia et al. [27] developed a common set of definitions of WLs to reduce the difficulties in matching customers' requirements with available resources. They proposed diverse cloud WL categorizations from different customers and then matched these categories with the available resource.

2.6 Summary

Although the adoption of cloud computing and its success has been rapid, characterizing cloud WLs is still not (yet) completely clear. This thesis will focus on characterizing one type of WL, specifically MR WLs, and will define a parametric model which will describe such a WL. The results of this model are used to develop an improved deployment strategy.

3 Methodology

The purpose of this chapter is to provide an overview of the research method used. Section 3.1 describes WL characterization and representation. Section 3.2 describes WL modeling. Section 3.3 explains the deployment strategy using the parametric model. Section 3.4 describes the research process. Section 3.5 describes the experimental design. Section 3.6 explains the techniques used to evaluate the reliability and validity of the data collected.

3.1 WL Characterization and Representation

Understanding WL characteristics is one of the primary research areas needed to improve the performance of cloud systems. If we possess some prior knowledge about the characteristics of the WLs, then we can set up the underlying platforms appropriately. It is too late to characterize WLs when the WL actually arrives at the data center. Understanding WLs by identifying some extra requirements using *implicit* constraints plays an important role in our research, as we need to understand each WL's characteristics before we can proceed toward our next goal. In order to characterize WLs, we categorize WLs into periodic, aperiodic, and sporadic WLs based on their job arrival rate, frequency of jobs submitted, and nature of the jobs. As noted earlier some WLs are computationally intensive, some memory intensive, and some WLs require both [25]. An in depth analysis of WLs and each WL's properties (including job duration, frequency of jobs submitted, resource utilization, etc.) are important in WL characterization. In [28], WL characteristics were observed by conducting a comprehensive WL trace analysis at job and task level granularity.

A WL may have consistent behavior in one context, but not in another. For example, if the WL consists of a sequence of web requests and the system is a web server with a single disk that serves requests in their arrival order, then the distribution of response times might be the relevant performance metric. However, this characterization does not apply when the server stores data on an array of disks and requests are served based on the requested page's size. Restricting the WL to a specific context can improve our WL model. In our model, we consider only MapReduce (MR) WLs as input. The basic details of how MapReduce works were given in Section 2.1 on page 6.

3.2 WL Modeling

We suggest a parametric model for MR WLs to find the implicit characteristics of the WLs that should be identified in order to make deployment decisions. Using this model, we seek to identify a deployment strategy in order to deploy these WLs on logical server platforms using disaggregated resources.

When a developer submits a Hadoop MR job to the YARN cluster we utilize the names of the input and output directories and the given java file in our analysis. The number of map tasks (corresponds to the number of splits) and the number of reduce tasks might be suggested by our deployment strategy using prior knowledge. For example, when submitting a MR job in YARN, users (e.g. a developer) provide (at least) the following:

- **A configuration file** (often in its default setting) – which can be used to select our deployment strategy, as the configuration file contains the values and intervals of parameters of the YARN components (e.g., parameters for the YARN schedulers and node monitors) with respect to amount of memory and number of virtual cores (vCores).

- **A jar file** containing the implementation of an MR model including a combiner.

- **Input directory** specifying the path in HDFS to the input files. The number of files stored in HDFS or Amazon Simple Storage Service (S3) determine the number of Map tasks (as an optional parameter).
- **Output directory** specifying the path in HDFS where output files should be written; the number of output file(s) in HDFS may determine the number of reduce tasks

Given these WLs we will augment them with a higher-level description in order to represent the WLs at the level of resource requests. This leads to the creation of an interpretation layer (translator) to translate a given user WL to the more elaborated WL description subsequently used for deployment. This elaborated WL description makes some of the *implicit* characteristics of the WLs that should be identified to facilitate the deployment *explicit*, and more precisely, these characteristics can be used to define a logical server (composed on top of the disaggregated resources). For instance, a MR job requires reading and writing data in both of its stages, knowing this facilitates optimization with respect to these operations in their own stage. A result of WL modeling is a new specification/representation of the input WLs. These steps are shown in Figure 3-1. Our parametric WL model for Hadoop MR jobs makes some assumptions. The underlying assumptions are: (1) all the tasks in a single job require the same amount of the different resources and (2) all the tasks are indivisible, i.e. each task is considered as one individual task and cannot be combined with another.

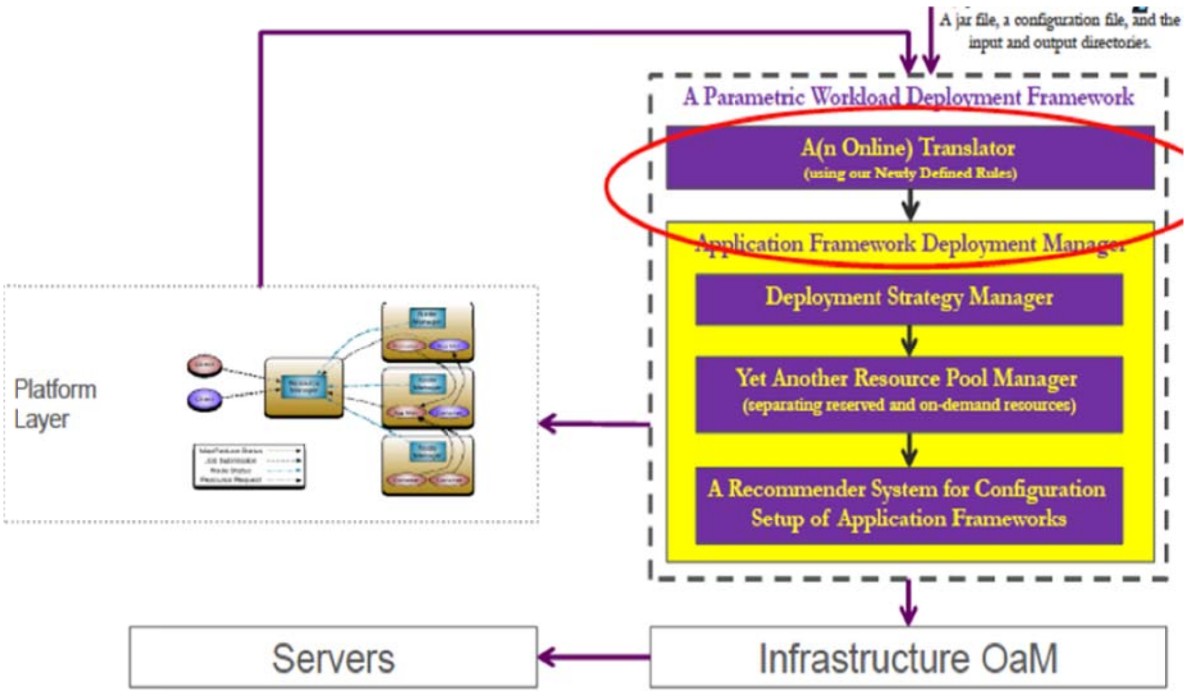
A jar file, a configuration file, and the input and output directories.
A Parametric Workload Deployment Framework
A(n Online) Translator
(using our Newly Defined Rules)
Application Framework Deployment Manager
Deployment Strategy Manager
Yet Another Resource Pool Manager
(separating reserved and on-demand resources)
A Recommender System for Configuration Setup of Application Frameworks
Platform Layer
Servers
Infrastructure OaM

Figure 3-1: Parametric framework architecture Overview

On the task level, our parametric WL model contains both basic and optional parameters. The basic parameters are:

- **A**(pplication) and Tenant identifier (**U**),
- **S**(cheduler),
- **C**(PU),
- **Mem**(ory) (Mem_M and Mem_R are two more detailed parameters decomposed from Mem),
- **DFS B**(lock size), or Input Splits ($\boldsymbol{I_s}$) → mapper instances,
- $\mathbf{N_r}$ →Reduce instances,
- **R**(eplica),
- Period (**T**), and
- Data locality (**L**) - the data locality of a task is more about IO waiting time of reading the source, and HDFS/S3 data and writing the output HDFS/S3 data: $\mathbf{L_r}$ and $\mathbf{L_w}$.

The Application ID is the global unique identifier of each submitted application. The Tenant identifier is a unique identifier for each tenant that describes all account information and user privileges in the system. The scheduler allocates the resources to run applications and monitors the status of the applications. This scheduler consists of two types of pluggable schedulers: CapacityScheduler and FairScheduler. By default Hadoop YARN is configured to use the CapacityScheduler as it allows multi tenancy[*] and sharing a large cluster by maximizing the throughput and cluster utilization. The FairScheduler allows YARN applications to share resources fairly in a large cluster (all applications get an equal share of resources on average over time). However, choosing the best scheduler for our parametric model is not yet finalized. The CPU parameter is the total processing time of a particular job in the Hadoop cluster. Memory is the total memory required by the Hadoop MR job. This memory parameter is further decomposed into Mem_M and Mem_R. Mem_M is the total memory needed to perform one map task of a job, whereas Mem_R is the total memory needed to perform one reduce task of a job. The DFS Block size refers to the block size in bytes of new files that will be created. This parameter plays a key role in calculating the number of mapper instances or input splits. The number of Reduce instances refers to Mapred.Reduce.Tasks parameter in Hadoop. The default value is 1. Increasing this value improves the utilization of hard disk I/O on a large input dataset, whereas with a small input dataset keeping this value small decreases the overhead in setting up tasks. The optimum value of reduce instances is not yet investigated, but is expected to be based on the input dataset size. The Replica parameter is the replication factor in the Hadoop cluster. The default value is 3. This is an important parameter to set in Hadoop in order to avoid data loss due to a failure. The Period parameter is the periodicity of the task in a job and has the values: periodic, sporadic, or aperiodic. The Data locality (L) parameter of a task is related to its replication factor.

The optional parameters are:

- The minimum **N**(etwork bandwidth) (between nodes),
- **D**(eadline) can be given either explicitly or implicitly,
- **RW**(Read/write ratio of data files),
- The size of requests ($\mathbf{R_s}$),
- The size of the input data ($\mathbf{I_d}$), and
- The size of the output data ($\mathbf{O_d}$).

There is a correlation (r) between some parameters, for example: r (L, N) = -1[†], e.g. a better L reduces the value of N.

[*] Multi tenancy refers to a single instance of software servers' multiple tenants.

[†] A correlation -1 refers here is that for every positive increase of 1 in data locality, there is a negative decrease of 1 in the Network bandwidth.

The configured block size and replication factor in HDFS play a major part in WL modelling. Blocks are replicated several times to ensure high data availability, as HDFS was intended to be fault-tolerant, while running on commodity hardware. A typical HDFS block size is 64MB. All blocks are of same size, except the last block in a file. When a user stores a file in HDFS, the Hadoop system decomposes it into set of blocks and store these in various worker nodes in the Hadoop cluster. The number of individual blocks is based on the block size set in a given Hadoop cluster. We can modify this block size within the Hadoop Cluster. If a user wants to change the block size for the entire cluster, he or she needs to add a property called **dfs.block.size** in the *hdfs-site.xml* file. Changing the block size affects *only* new files that are created and does *not* affect existing files in HDFS.

File blocks are replicated for fault-tolerance. The replication factor is also configurable in a Hadoop Cluster. An application can specify the replication factor of each file. This value can be set at the time of creation, but can also be modified later. All of the files in HDFS are write once and strictly limited to one writer at a time. We can adjust the global replication factor for the whole cluster or change the replication factor for each file that is created. There will be n-1 duplicate blocks distributed across the cluster for each block stored in HDFS. The property **dfs.replication** is set in *hdfs-site.xml* to adjust the replication factor for the whole cluster. To change the replication per file, we need to first create the file in HDFS, then set the replication by setting **hdfs dfs –setrep –w X <file-path>** where X is the replication factor[*]. In this thesis, we set the block size to 64 MB and replication factor to 2 for the default configuration of a Hadoop cluster. However, our analysis shows that increasing the block size to 512 MB and replication factor 3 improves the resource utilization and decreases the job completion time. We developed a deployment strategy based on both the default configuration and an extended configuration as explained in detail in Section 3.3.

3.3 Deployment Strategy

Our proposed parametric model helps to analyze the importance of each of the parameters. This analysis helps identify a good deployment strategy that can deploy diverse WLs. The basic deployment strategy is based upon two dedicated resource pools: (1) one that handles long running services and (2) another that handles latency-critical services. If we use a single powerful server, then new jobs will experience increased waiting times (as each job will need to wait for earlier jobs to terminate). In contrast, if we use multiple less powerful servers rather than one powerful machine, then the instantiation overhead will be greater because of more frequent setting up of the platforms. This suggests that we want to define a combined deployment strategy. However, in this thesis project we focus on dimensioning the size of a cluster when using similar configurations for all of the nodes in the cluster. This means that we assume that we do **not** have heterogeneous servers in the resource pools. As a result, each resource pool is assumed to be composed of servers that are homogenous (i.e., they have identical hardware configurations). Homogenous servers are used for demonstration purpose only. Ideally, deployment should work for heterogeneous servers in which the cluster will be composed based on the resource required (calculated from the deployment strategy). We tried to collect datasets of various real-time workloads for heterogeneous servers within the Ericsson environment. However, they were inaccessible due to political and security reasons; hence, we were constrained to perform experiments and collect the data using sample workloads. If we had been given access to real workloads and datasets, then the deployment strategy would cover many of the parameters needed for a more refined solution.

As a result of the above limitations, we define a deployment strategy for Hadoop MR jobs using the parametric model we propose, while assuming the size of the underlying physical servers is fixed

[*] Note: Replication of individual files takes time and it varies depending on the number of replicas, file size, and DataNode hardware. Hence you should only change the replication factor per file if you really need to.

(i.e., unvaried over time). Our deployment strategy includes only few of the performance tuning parameters from the basic parametric model (described in Section 3.2). We require a jar file as input, a directory path where the input data resides, and a directory path where the output of the job should be stored. Using the specified input directory, the size of the input data is calculated. The input data size and HDFS block size are used to calculate the number of map splits needed. Based on the number of splits, the maximum memory required for the WL can be estimated as described in the following sections, given:

$D_s \longrightarrow$ Data size of input in GB	$Mem_N \longrightarrow$ Total RAM per server/node
$B \longrightarrow$ Block Size	$Mem_C \longrightarrow$ Total Memory per container
$R \longrightarrow$ Replication	$M_p = I_s \star Mem_C \longrightarrow$ Physical memory
$I_s \longrightarrow$ No of input splits	$M_v \longrightarrow$ Virtual Memory
$N_s \longrightarrow$ No of servers/Data nodes	$E_s \longrightarrow$ Execution time of job in seconds

3.3.1 Default Configuration

With the following default configuration:

$$B = 64\,MB$$

$$R = N_s$$

$$Mem_N = 2\,GB$$

$$set\ Mem_C = 180\,MB$$

Given D_s = 1 GB,

Assign I_s = 16, then $M_p = 16 \star Mem_C$

$$\Rightarrow M_p \simeq 3\,GB$$

We know $Mem_N = 2$ GB.

$$\Rightarrow N_s = 2$$

This shows that to process 1 GB input data requires two servers that collectively provide 4 GB of memory capacity to execute the job in a better execution time (E_s). This could also be achieved by one server with 4GB of memory. However, since all our servers have a fixed 2GB memory configuration, this job requires two servers.

3.3.2 Extended Configuration

With the following Extended Configuration:

$$B = 512\,MB$$

$$R = N_s$$

$$Mem_N = 2\,GB$$

$$set\ Mem_C = 180\ MB$$

Given D_s = 1 GB,

Assign I_s = 2, then $M_p = 2 \star Mem_C$

$$\Rightarrow M_p \backsimeq 1\ GB$$

We know Mem_N = 2 GB.

$$\Rightarrow N_s = 1$$

This shows that 1 GB data input needs just one server which provides 2 GB of memory capacity to execute the job in a better execution time (E_s). Our deployment strategy gives the maximum resources required for a particular WL and assumes that the actual resource usage will not exceed this value.

Setting up of a cluster with the servers required to handle the WL occurs after this stage.

3.4 Research Process

The overall research process (shown in Figure 3-2) consists of:

- Understand the different types of WLs in Hadoop MR applications,
- Define a parametric model that describes the Hadoop MR jobs,
- Create Hadoop multi node cluster to setup test execution environment,
- Experiment with executing famous examples of Hadoop MR jobs, such as wordcount and grep search with varying input data size, block size, number of nodes in the cluster to observe the different behaviors and patterns,
- Collect data from the experimental evaluation,
- Analyze the collected data to find the deployment strategy in order to set up the logical platform (i.e., choosing the number of slave nodes), and
- Implement a deployment manager to find and use the deployment strategy based on incoming WLs to the datacenter.

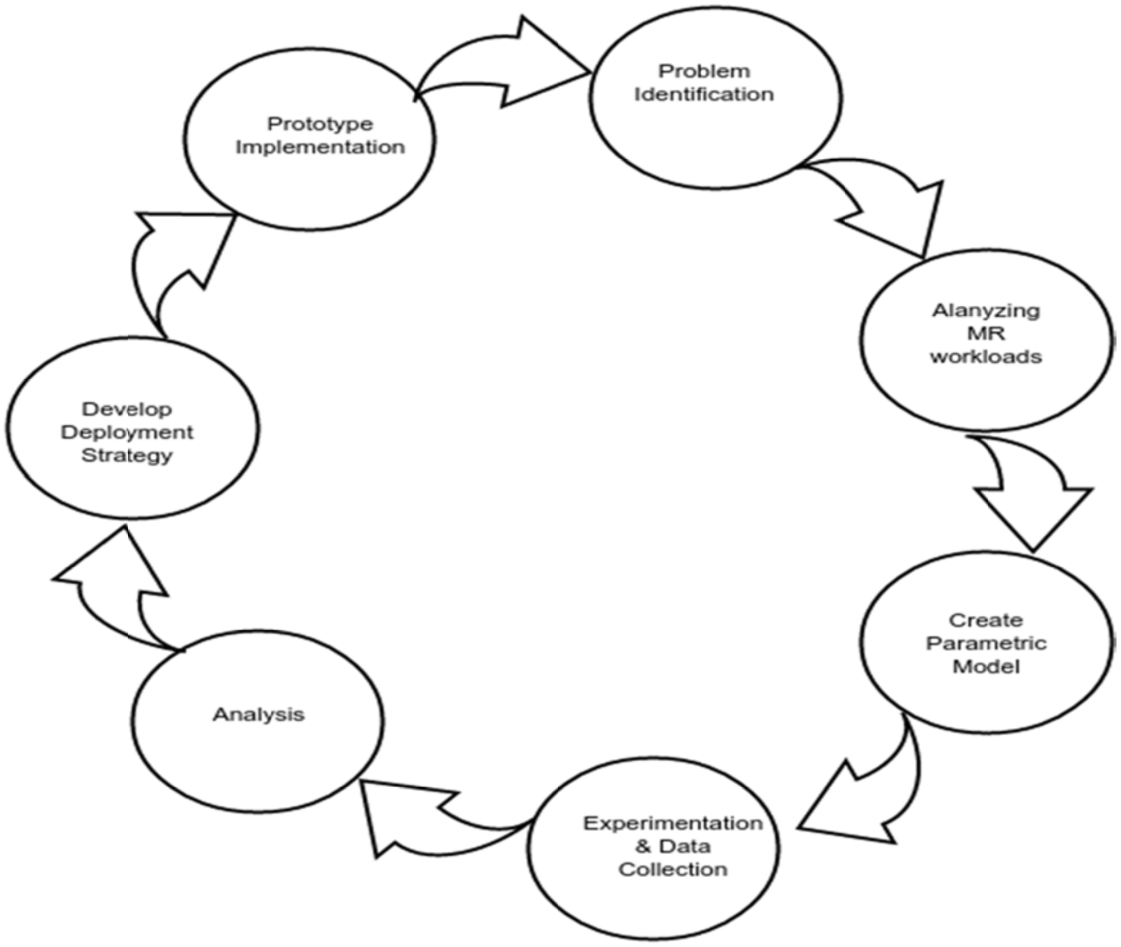

Figure 3-2: Research Process

3.5 Experimental Setup

This section describes the experimental test environment and the software/hardware configurations used.

3.5.1 Test environment: Hardware/Software to be used

A Hadoop cluster was setup using OpenStack on an underlying server whose specification is shown in Table 3-1. Five virtual machines (VMs) were configured on the server. The hardware and software configuration for each of the VM is shown in Table 3-2. Each VM is assigned 1 vCPU core, 2 GB RAM, and 20 GB of hard disk storage.

Table 3-1: Hardware **configuration of the server.**

Hardware	CPU Model	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
	Number of Cores	40
	Hard disk	4 TB
	Memory	158 GB

Hadoop-2.7.2 was used with a single VM configured as the NameNode and the remaining four VMs as DataNodes. The NameNode was not used as a DataNode. The replication level of each data block was set to 3. Two typical Hadoop MapReduce applications (i.e., wordcount [29] and grep [30]) were run as Hadoop YARN jobs. The TeraGen application [31] available as part of the Hadoop distribution was used to generate different sizes of input data.

Table 3-2: Software and Hardware configuration of each VM.

Software	Operating System	Ubuntu 14.04.3 LTS
	JDK	OpenJdk 1.7
	Hadoop	2.7.2
	OpenStack	Nova
Hardware	CPU	1 vCPUs
	Processor	Intel Xeon
	Hard disk	20 GB
	Memory	2 GB

3.6 Assessing reliability and validity of the data collected

This section describes the reliability and validity of the data collected. Section 3.6.1 describes the reliability of the data and Section 3.6.2 describes the validity of the data.

3.6.1 Reliability

The experiments will be tested within Ericsson's lab infrastructure. The results need to be consistent over multiple iterations. The OpenStack engine used to set up the VMs ensures that the shared resources availability is guaranteed according to the configuration. So if the results are consistent over multiple iterations, this ensures their reliability.

3.6.2 Validity

The experiments are done in a cluster of VMs in a private cloud using OpenStack. The validity of the collected data is assessed by comparing the results obtained from the experiment with that of measurements obtained from real clusters in a data center. The measurements obtained in the experiments are explained in Section 5.1.

4 Evaluation

The deployment strategy was evaluated based on experimental results from a Hadoop cluster using OpenStack VMs. The following steps were followed to identify a deployment strategy:

- Created a Hadoop multi node cluster as the test execution environment.
- Executed well-known examples of Hadoop MR jobs *wordcount* and *grep* search with different combinations of input data size and block size by scaling up and scaling down nodes in the cluster to observe the different behaviors and patterns.
- Collected data from the experimental evaluation.
- Analyzed the experimental results to find a deployment strategy to set up the logical platform (size and configuration of the servers).

4.1 Expected Results

The final outcome of this research is a characterization of a small number of cloud WLs and a refined WL specification. This characterization was used to configure a better logical server for deployment. The outcome of this thesis will be:

1. A parametric model that best describes the Hadoop MR jobs,
2. A deployment strategy to deploy the refined WLs on logical platforms, and
3. A prototype co-ordination framework which refines the incoming WL based on the parametric model.

4.2 Experimental Test Description

A cluster with the configuration specified in Section 3.5 was setup in the OpenStack Virtualization environment. The job configurations listed in Table 4-1 were tested with different block sizes: 64, 128, 256, and 512 MB.

Table 4-1: Job configurations tested

Job ID	Description
1	Simple test with single node cluster
2	Test with multi node cluster
3	Test MR job(wordcount) with 1 GB input and default configuration
4	Test MR job(wordcount) with 1 GB input and 64MB block size
5	Test MR job(wordcount) with 1 GB input and 512MB block size
6	Test MR job (wordcount) with 2 GB input and default configuration
7	Test MR job(wordcount) with 2 GB input and 64MB block size
8	Test MR job(wordcount) with 2 GB input and 512MB block size
9	Test MR job(grep) with 1 GB input and default configuration
10	Test MR job(grep) with 1 GB input and 64MB block size
11	Test MR job(grep)with 1 GB input and 512MB block size
12	Test MR job(grep) with 2 GB input and default configuration
13	Test MR job(grep)with 2 GB input and 64MB block size
14	Test MR job(grep) with 2 GB input and 512MB block size
15	Test MR job(wordcount) with 3 GB input and default configuration
16	Test MR job(wordcount) with 3 GB input and 64MB block size
17	Test MR job(wordcount) with 3 GB input and 512MB block size

4.3 Implementation

This section describes an implementation of a prototype coordination framework/WL translator that accepts user input and dimensions the cluster according to the deployment strategy selected by the framework.

The WL translator is implemented as REST web services using a REST API similar to the Hadoop Resource Manager REST web services. This translator is a separate component located in between the user and the Hadoop cluster. The user provides input to the WL translator as an XML file specifying the MapReduce implementation jar file location, a directory path where the input data is located, and a directory path where the output should be stored. See Appendix A for a sample WL. To setup the translator, configure SSH access from the translator node to the running Hadoop cluster (see Appendix C for instructions to set up Hadoop cluster). To submit a WL to the translator, a running Hadoop cluster and the URL link to communicate the cluster using REST API are required. A sample POST request to a Hadoop cluster running in local machine will look like

curl -X POST -H 'Accept: application/xml' -H 'Content-Type: application/xml' http://localhost:8080/rest/translate/apps -d @workload.xml

When the user submits a WL in the specified format, the translator calculates the input data size. After this calculation, the deployment manager is called to identify the size of the servers. The deployment manager uses the parametric model defined in Section 3.3 to return the amount of memory required to process the job with a minimal job completion time. Based on the amount of memory required, the translator calls the OpenStack REST services to create the number of instances needed to satisfy the specified resource requirements. Hadoop has to be installed and configured on each of these instances to setup the cluster. To avoid unnecessary installation of Hadoop at every call, we created a template VM image in OpenStack with Hadoop installed and use this template to launch an instance whenever required.

The submission of a Hadoop MR job involves two steps. The first step is to get the application ID. This is followed by the actual job submission. The translator removes this multi-step overhead as the translator automatically scales up or down the instances in the Hadoop cluster based on the WL resource demands as estimated by the translator. When scaling up the Hadoop cluster, no additional configuration is required. For scaling down the Hadoop cluster, we need to gracefully remove the data nodes from the cluster to avoid the risk of data loss. As a result, we decided that the minimum cluster size was one master and two slave nodes as core instance groups (each with a DataNode daemon running on it). Any slave nodes added to this core cluster are referred as spot instances (and operate without a DataNode daemon). These nodes will not store any HDFS data for the job, but are used as computing resources to execute the MR job. As a result, we can add and delete spot instances to and from the cluster without affecting the HDFS data. Each spot instance has to be configured in the *dfs.exclude* file under the Hadoop configuration directory to exclude it from storing data. The core instance groups have to be configured in the *dfs.include* file and the nodes have to be refreshed by using the command "*hdfs dfsadmin -refreshNodes*". There is an alternative method of performing scaling down without maintaining a minimum cluster size. In this method, the cluster can scale up as per resource demands and if the resource demand is less than the available cluster's resources, then the extra nodes/resources can be gracefully decommissioned (as explained in previous method). By doing this, the data that were present in the decommissioned nodes will be recreated within the active cluster's resources. Ideally, this method is only used when there is a node failure in the cluster.

Since the translator needs to perform lots of computation (such as finding the input data size, applying a deployment strategy, identifying the required resources to execute the particular job, and finally composing a cluster and deploying the job) from the initial job submission to generating a response, there was a time out error thrown after exceeding 1 minute from the the POST request. The time taken by the translator for different jobs with different input size and block size are available in Table 5-4, Table 5-5, and Table 5-6. An asynchronous web service was implemented to overcome this timeout issue. In such a service the first step is to first send an acknowledgement of the request and then to process the request. This enables the client application to continue its work and later handle the response.

Figure 4-1 illustrates the data flow in the translator according to the following steps:

1. The user submits the Hadoop MR job using the REST API.
2. The translator receives the request and stores it in the request.
3. The translator returns an acknowledgement to the user.
4. The Message driven bean (MDB) listener on the request queue receives the message and initiates processing of the request. In this scenario, a single MDB is associated with the request queue which handles both (request and response) processing.
5. The Request MDB calls the required method in the translator.
6. The translator calls the deployment manager to dimension the cluster according to the parametric model for the given WL.
7. The translator calls the running cluster and configures the resources for the request.
8. The Hadoop cluster returns a response.
9. The translator deploys the WL to the Hadoop cluster using the REST API
10. The translator returns a response to request MDB.
11. The request MDB, acting as a callback client, returns a response to the callback service.
12. The callback service returns a receipt confirmation message.
13. The request MDB returns a confirmation message to the request queue to terminate the process.

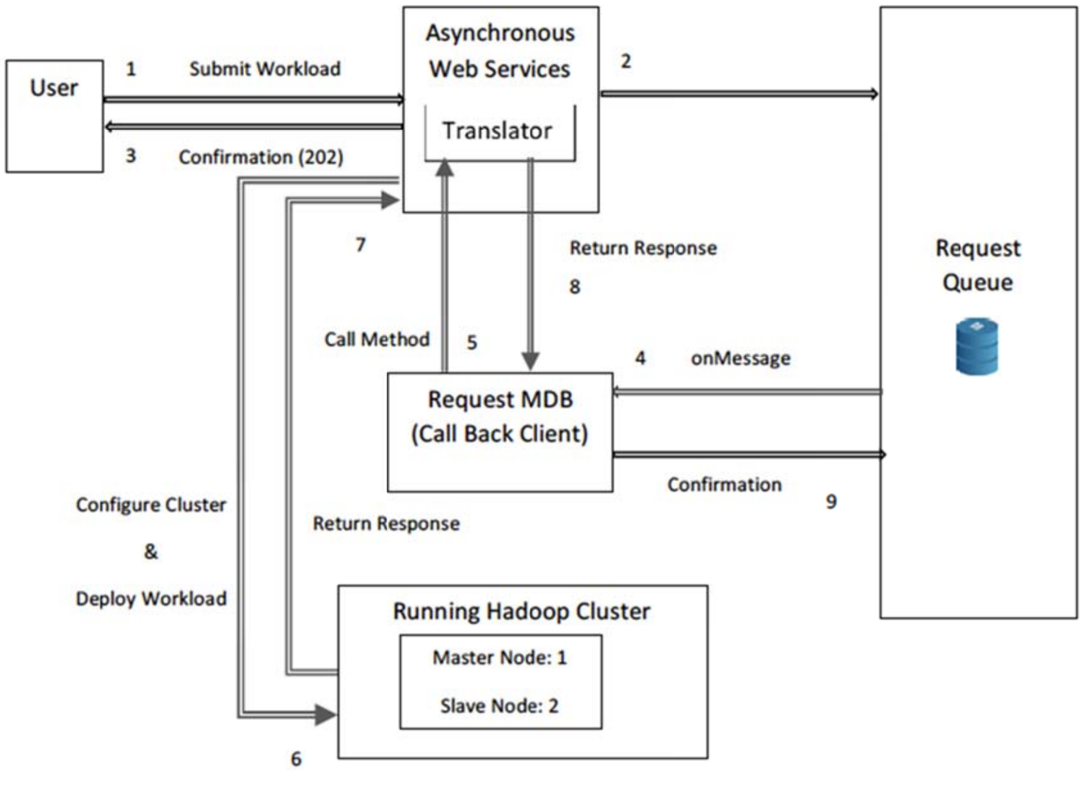

Figure 4-1: **Data flow diagram of translator**

5 Analysis

This chapter presents an analysis of the evaluation described in the previous chapter. This analysis serves as the basis to find a deployment strategy – specifically dimensioning the cluster. The metrics used in our analysis are memory usage (further split into physical memory and virtual memory), CPU processing time, and execution time. All of these metrics are analyzed on a per job basis.

5.1 Major results

The results of the experiments are shown in Table 5-1, Table 5-2, and Table 5-3. With 1 GB input data and 64 MB block size, the amount of memory required to process the WL is 3 GB. This is clear from the measurement data. However, the memory required is the same irrespective of the number of slave nodes in the Hadoop cluster. However, we evaluate the best deployment strategy based upon a combination of memory required, job completion time, and number of slave nodes. Providing the slave nodes with 3 GB RAM gives the best job completion time. Since each slave node in our cluster has the same memory capacity, i.e., 2 GB RAM, it is always better to provide two slave nodes (giving a total capacity of 4 GB) which is more than the required 3 GB memory.

The job completion time for 1 GB data is not reduced by much when varying the number of slave nodes. In the case of 2 GB data, the job completion time is improved with three slave nodes rather than when only one or two slave nodes are in the cluster. In the case of 3 GB of data, the job completion time is improved with four slave nodes rather than one, two, or three slave nodes are utilized.

The same WL works better with an extended configuration, such as a 512 MB block size. From Table 5-1, we can see that with a 512 MB block size, the amount of memory required is reduced to 1/6th of the memory required with a 64 MB block size and in turn reduces the number of nodes needed in the cluster to execute the job in a shorter job completion time. This is because the number of input splits are reduced when the block size is higher. The extended configuration not only decreased the amount of memory required, but also decreased the job execution time as is evident from Table 5-2 and Table 5-3. These experimental results confirmed the deployment strategy we defined in the Section 3.3. From our analysis, we observe that the DFS block size, replica, number of mappers, and number of reducers plays an important role in modeling the WL. However, we defined a simple model and limited the parameters in the deployment strategy. However, the WL model can be further refined in future by adding additional parameters. Figure 5-1, Figure 5-2, Figure 5-4, Figure 5-5, and Figure 5-6, presents the experimental results of Table 5-1, Table 5-2, and Table 5-3.

Table 5-1 Results obtained from 10 iterations with 1 GB input size

WL Type	Block Size (MB)	Input Splits	Average Physical Memory (GB)	Average Virtual Memory (GB)	Average CPU Processing Time (sec)	Number of Slave Nodes	Average Job Completion Time (sec)	Standard Deviation JCT
Wordcount	64	16	3.0098	12.50426	2.48	4	177	1.5837
	64	16	2.967	12.50413	2.37	3	173	1.3672
	64	16	2.97718	12.50414	2.38	2	171	0.9756
	64	16	3.03069	12.50411	2.33	1	285	1.0132
	512	2	0.47316	2.21021	1.38	1	141	2.1134
	512	2	0.47694	2.21021	1.39	2	135	0.7384
Grep	64	16	3.00988	12.50426	2.48	4	177	1.6593
	64	16	3.169	12.46296	2.45	3	176	1.0039
	64	16	2.97718	12.50414	2.38	2	172	2.3901
	64	16	3.09934	12.46394	2.34	1	298	0.8495
	512	2	0.47316	2.21021	1.38	1	143	0.4291

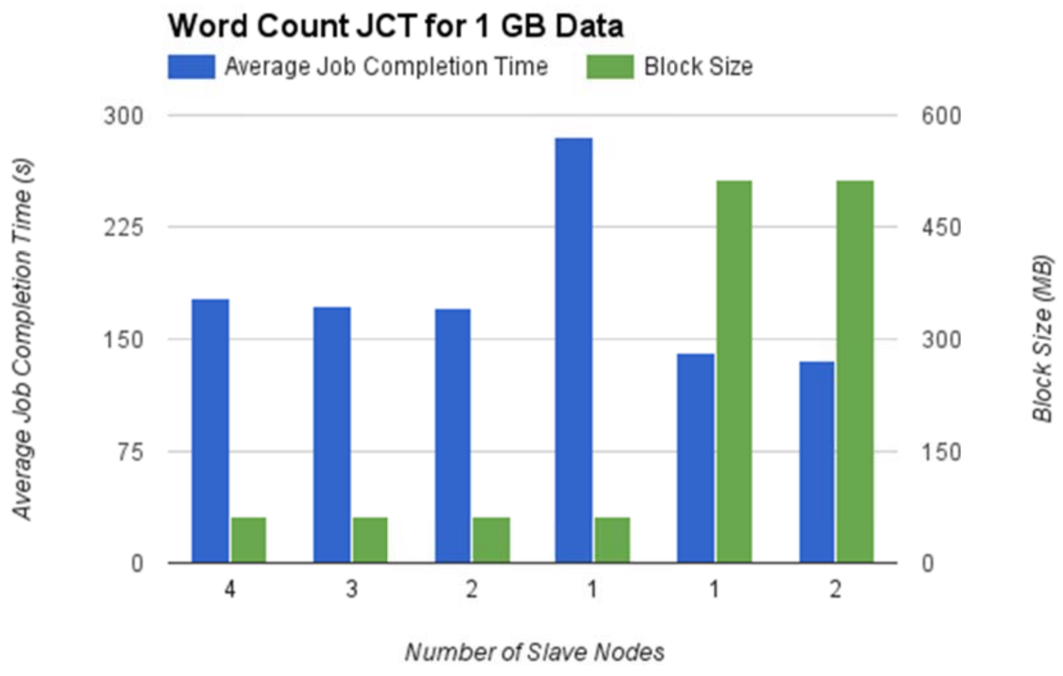

Figure 5-1: Word Count Job Completion Time for 1 GB data

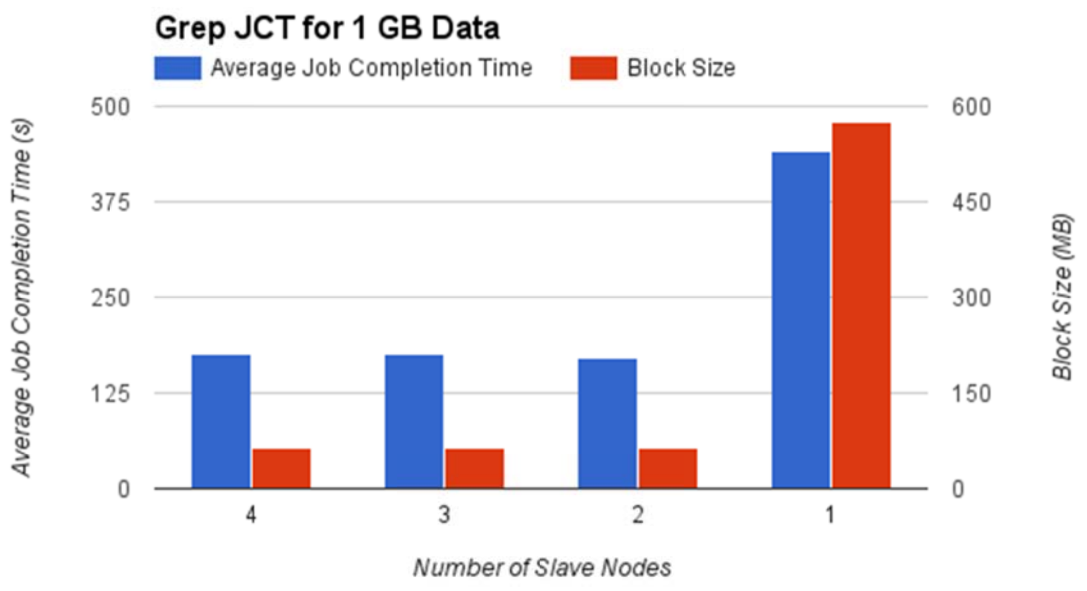

Figure 5-2: **Grep Job Completion Time for 1 GB data**

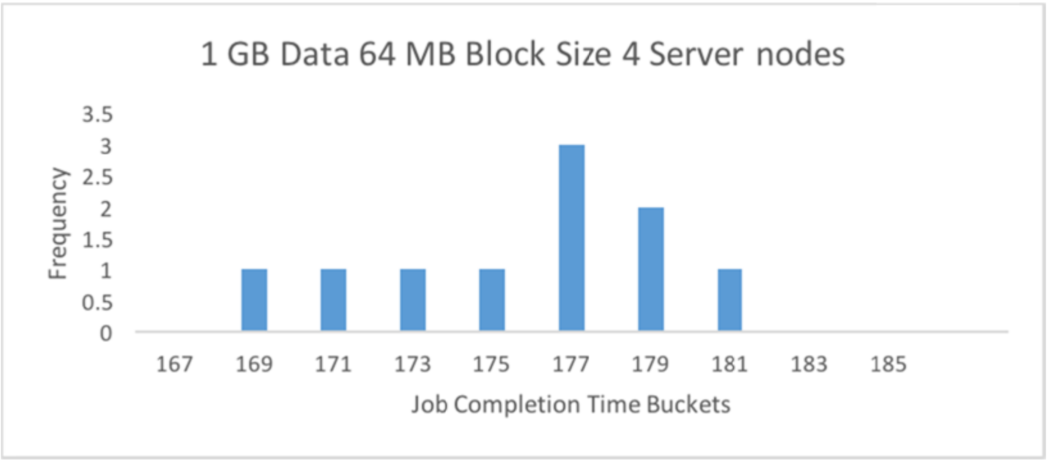

Figure 5-3 **Distribution of JCT for 1 GB data with 64 MB Block Size and 4 server nodes**

Table 5-2 Results obtained from 10 iterations with 2 GB input size

WL Type	Block Size (MB)	Input Splits	Average Physical Memory (GB)	Average Virtual Memory (GB)	Average CPU Processing Time (sec)	Number of Slave Nodes	Average Job Completion Time (sec)	Standard Deviation JCT
Wordcount	64	30	5.49	22.80	5.749	4	254	2.1248
	64	30	5.48	22.80	4.81	3	262	1.9530
	64	30	5.46	22.80	4.70	2	317	2.0485
	512	4	0.834	3.68	2.73	1	263	1.4927
	512	4	0.818	3.68	2.76	2	265	1.6642
Grep	64	30	5.46	22.80	4.71	2	320	1.5492
	64	30	5.48	22.80	4.81	3	267	1.1094
	64	30	5.49	22.80	5.75	4	260	1.8539
	512	4	0.92	3.94	2.80	1	267	2.0159

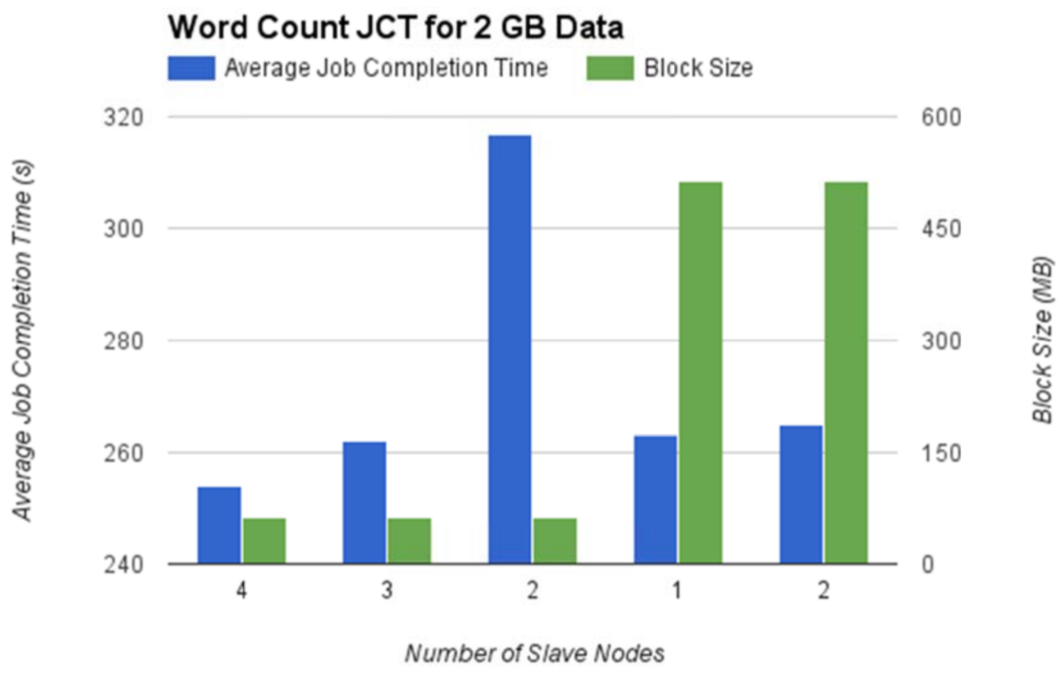

Figure 5-4: Word Count Job Completion Time for 2 GB data

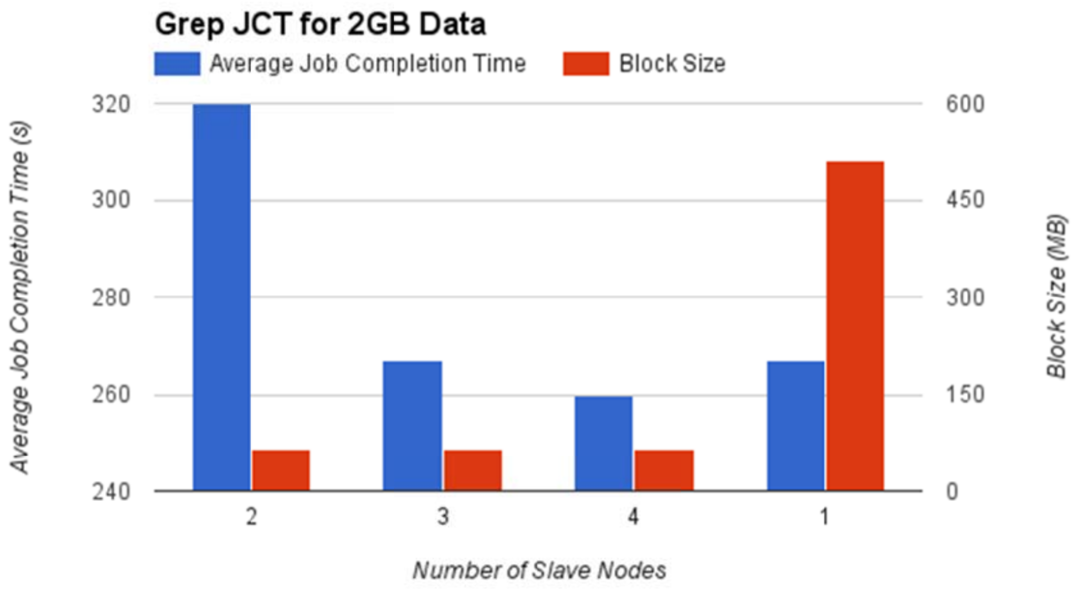

Figure 5-5: **Grep Job Completion Time for 2 GB data**

Table 5-3 **Results obtained from 10 iterations with 3 GB input size**

WL Type	Block Size (MB)	Input Splits	Average Physical Memory (GB)	Average Virtual Memory (GB)	Average CPU Processing Time (sec)	Number of Slave Nodes	Average Job Completion Time (sec)	Standard Deviation JCT
Wordcount	64	46	8.52	34.57	7.34	4	330	2.0193
	64	46	8.39	34.57	7.28	3	382	0.9034
	64	46	8.42	34.57	7.22	2	458	1.2823
	64	46	8.38	34.57	7.28	1	471	1.4934
	512	6	1.19	5.15	4.16	1	393	2.1167
	512	6	1.18	5.15	4.31	2	381	0.9589
	512	6	1.18	5.15	4.30	3	415	1.6635
Grep	64	46	8.52	34.60	7.35	4	330	2.0698
	64	46	8.39	34.57	7.30	3	385	2.0432
	64	46	8.43	34.60	7.23	2	462	1.7803
	512	6	1.26	5.18	4.17	1	381	1.8580

Table 5-1, Table 5-2, and Table 5-3 contains the data gathered from the Hadoop cluster executing Word Count and Grep workloads with different input data size, block size, and different size of the slave nodes. The workloads were executed ten times with each of the possible combinations above and the average values were calculated for Physical Memory consumed, CPU processing time, Virtual memory used, and Job Completion time. The standard deviation of the JCT indicates that the cluster was consistent in using the hardware resources and job completion time is normally distributed. As shown in the Figure 5-3, the Job Completion Time for 1 GB data with 64 MB Block Size and 4 server nodes configuration is normally distributed. The distribution is the same for all the other combinations too.

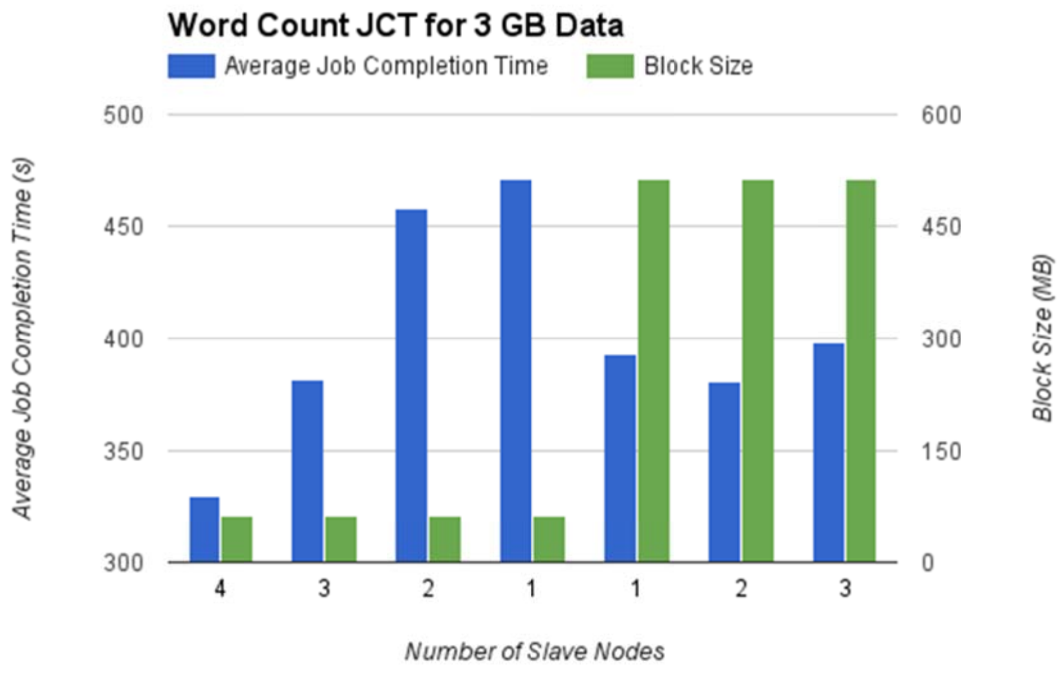

Figure 5-6: **Word Count Job Completion Time for 3 GB data**

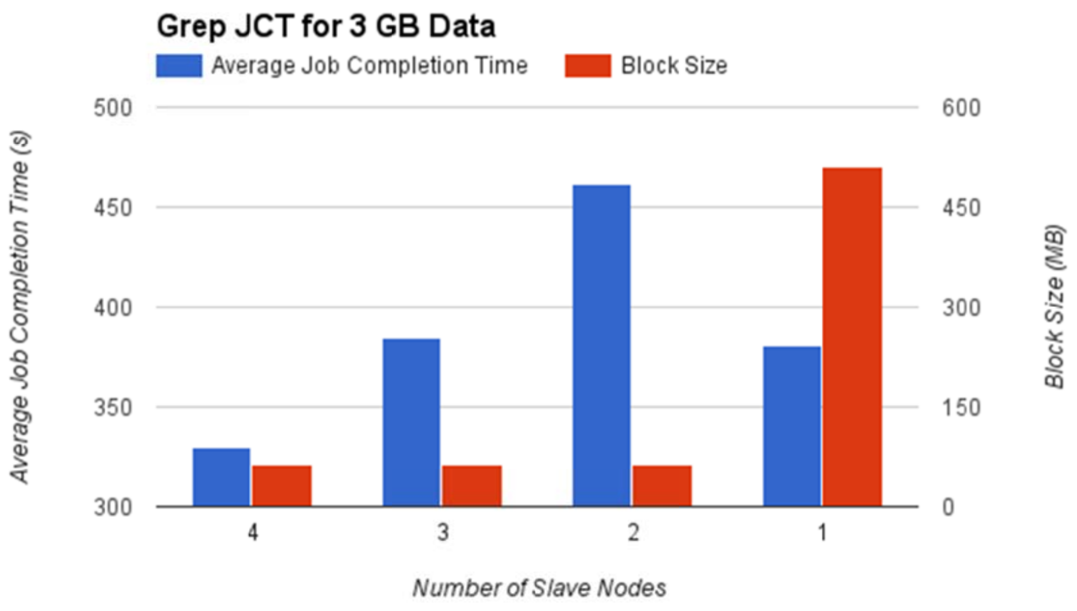

Figure 5-7: **Grep Job Completion Time for 3 GB data**

The above results were gathered by manually setting up the cluster with necessary slave nodes and block size. The average execution time obtained in the above tables is the time to execute the job after the cluster is setup according to the deployment strategy. The WL translator helps the user remove the manual steps of calculating the necessary slave nodes needed and launching the new instances and decommissioning the excess spot instances instantiated for the job after job completion. Table 5-4, Table 5-5, and Table 5-6 prove that the overhead introduced by the translator is negligible when compared to the manual intervention and time needed for executing each job.

Table 5-4 **Translator measurement for 1 GB Data**

WL Type	Block Size (MB)	Number of Slave Nodes	Average Completion Time including cluster setup (sec)	Standard Deviation
Wordcount	64	4	188	1.9375
	64	3	179	2.1582
	64	2	172	1.8529
	64	1	287	1.7458
	512	1	141	2.0834
	512	2	135	1.8974

Table 5-5 Translator measurement for 2 GB Data

WL Type	Block Size (MB)	Number of Slave Nodes	Average Completion Time including cluster setup (sec)	Standard Deviation
Wordcount	64	4	266	2.6327
	64	3	270	1.5082
	64	2	319	2.8570
	512	1	264	2.1839
	512	2	267	1.8328

Table 5-6 Translator measurement for 3 GB Data

WL Type	Block Size (MB)	Number of Slave Nodes	Average Completion Time including cluster setup (sec)	Standard Deviation
Wordcount	64	4	340	2.4782
	64	3	387	2.9381
	64	2	460	2.0062
	512	1	394	3.2701
	512	2	383	2.7192
	512	3	417	2.6910

5.2 Reliability Analysis

The data collected is reliable since it was gathered from experimental tests (conducted in Ericsson's lab infrastructure) rather than via simulation tool. Also, each test was executed a minimum of ten times to check whether we receive consistent performance metrics. The metrics are shown in Section 5.1. The standard deviation is very small for the sample workloads executed a minimum of ten times with different Block Size and input data size.

The possibility of skew in the measurements due to usage of virtual machines for the experimental setup is ignored because of the consistent results obtained. If the shared resources were not guaranteed to the VMs, the job completion time would have changed a lot. The consistent results confirm that the shared resources were guaranteed to the VMs by the OpenStack engine.

5.3 Validity Analysis

The experiments were done on a cluster of VMs set up in a private cloud (OpenStack) and automated using the implemented WL translator. The validity of the results could not be checked since Ericsson could not provide real workloads and physical machines to redo the experiments in a real environment.

6 Conclusions and Future work

In this chapter, we present the conclusions of our work, limitations of the results obtained, and suggest future work to help others to follow up this work.

6.1 Conclusions

In this thesis, we conducted an in depth investigation of MapReduce WLs in an open source implementation of Hadoop. In order to describe what Hadoop MR jobs look like, we identified some implicit parameters and defined a parametric model for Hadoop MR jobs. Next we chose a small number of performance tuning parameters and then we attempted to identify the best deployment strategy in terms of dimensioning the Hadoop cluster. The experimental results showed that with well tuned parameters, MR jobs can achieve an improvement (i.e., a reduction) in the job completion time and the utilization of the resources are improved. This research satisfies all the elements described in the problem statement. However, this research has alternate approach in the beginning which is described more in Appendix D.

6.2 Limitations

Though we improved the job completion time and resource usage, we only used a small number of parameters due to the need for extensive testing of various configurations on different cluster sizes. The cluster consisted of a set of VMs which shared the same underlying hardware. Also the WLs tested were only those sample WLs available as part of Hadoop distribution, i.e., there were no real WLs available for testing. Conducting the thesis at Ericsson and with limited time and resources forced the above two limitations of hardware and real workloads.

6.3 Future work

The parametric model we proposed is a simple model. However, it could be further refined to optimize the usage of disaggregated resources. The results of this thesis should be verified in larger sized clusters. Furthermore, more types of WLs should be introduced in the experiments to refine the model, rather than being specific to the few types of WLs available in the Hadoop distribution. In addition, the effects of changing other parameters, such as data locality, number of mappers and reducer slots in Hadoop cluster, and file buffer size should be studied. Choosing the best scheduler for our parametric model will be investigated. The optimum value of reduce instances will be investigated in future, but is expected to be based on the input dataset size.

6.4 Reflections

This research proposes new solutions to workload characterization aimed at performance optimization. The optimization potentially has some environmental implications as increased performance reduces the time that these jobs need to execute, hence reducing power consumption and makes the underlying resources available to execute other task. However, during the course of this project we have not encountered any major issues that have new ethical or social implications. The research implications are essentially the same as for all the existing work in each of the areas.

References

[1] "Home » OpenStack Open Source Cloud Computing Software." [Online]. Available: https://www.openstack.org/. [Accessed: 28-May-2016].
[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[3] V. K. Vavilapalli *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, New York, NY, USA, 2013, pp. 5:1–5:16.
[4] "Elastic Compute Cloud (EC2) Cloud Server & Hosting – AWS," *Amazon Web Services, Inc.* [Online]. Available: http://aws.amazon.com/ec2/. [Accessed: 28-May-2016].
[5] "Microsoft Azure: Cloud Computing Platform & Services." [Online]. Available: https://azure.microsoft.com/en-us/. [Accessed: 28-Sep-2016].
[6] "Google Compute Engine Documentation | Compute Engine | Google Cloud Platform," *Google Developers*. [Online]. Available: https://cloud.google.com/compute/docs/. [Accessed: 28-Sep-2016].
[7] "Rackspace: Managed Dedicated & Cloud Computing Services," *Rackspace Hosting*. [Online]. Available: https://www.rackspace.com/. [Accessed: 28-Sep-2016].
[8] D. Nurmi *et al.*, "The Eucalyptus Open-Source Cloud-Computing System," in *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, 2009. CCGRID '09*, 2009, pp. 124–131.
[9] "OpenNebula | Flexible Enterprise Cloud Made Simple." [Online]. Available: http://opennebula.org/. [Accessed: 28-May-2016].
[10] J. Panneerselvam, L. Liu, N. Antonopoulos, and Y. Bo, "Workload Analysis for the Scope of User Demand Prediction Model Evaluations in Cloud Environments," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, 2014, pp. 883–889.
[11] "Welcome to Apache™ Hadoop®!" [Online]. Available: http://hadoop.apache.org/. [Accessed: 08-Jul-2016].
[12] "Welcome to Apache Pig!" [Online]. Available: https://pig.apache.org/. [Accessed: 08-Jul-2016].
[13] "Apache Hive TM." [Online]. Available: https://hive.apache.org/. [Accessed: 08-Jul-2016].
[14] "Apache Spark™ - Lightning-Fast Cluster Computing." [Online]. Available: http://spark.apache.org/. [Accessed: 08-Jul-2016].
[15] "core.xml." [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/core-default.xml. [Accessed: 27-Nov-2016].
[16] "hdfs.xml." [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml. [Accessed: 27-Nov-2016].
[17] "yarn.xml." [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-common/yarn-default.xml. [Accessed: 27-Nov-2016].
[18] "mapred.xml." [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml. [Accessed: 27-Nov-2016].
[19] H. Yang, Z. Luan, W. Li, D. Qian, and G. Guan, "Statistics-based Workload Modeling for MapReduce," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, 2012, pp. 2043–2051.
[20] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 4, pp. 34–41, Mar. 2010.
[21] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload Analysis and Demand Prediction of Enterprise Data Center Applications," in *2007 IEEE 10th International Symposium on Workload Characterization*, 2007, pp. 171–180.

[22] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, “Statistics-driven workload modeling for the Cloud,” in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*, 2010, pp. 87–92.

[23] C. Delimitrou and C. Kozyrakis, “HCloud: Resource-Efficient Provisioning in Shared Cloud Systems,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2016, pp. 473–488.

[24] Q. Hao, “Keynote: #x201C;High throughput computing data center #x201D;,” in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2014, pp. 1–1.

[25] B. Feng, J. Lu, Y. Zhou, and N. Yang, “Energy Efficiency for MapReduce Workloads: An In-depth Study,” in *Proceedings of the Twenty-Third Australasian Database Conference - Volume 124*, Darlinghurst, Australia, Australia, 2012, pp. 61–70.

[26] X. Wang, S. Huang, S. Fu, and K. Kavi, “Characterizing Workload of Web Applications on Virtualized Servers,” *arXiv:1402.3549 [cs]*, Feb. 2014.

[27] W. D. Mulia, N. Sehgal, S. Sohoni, J. M. Acken, C. L. Stanberry, and D. J. Fritz, “Cloud Workload Characterization,” *IETE Technical Review*, vol. 30, no. 5, pp. 382–397, Sep. 2013.

[28] Z. Ren, X. Xu, J. Wan, W. Shi, and M. Zhou, “Workload characterization on a production Hadoop cluster: A case study on Taobao,” in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, 2012, pp. 3–13.

[29] “MapReduce Tutorial.” [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html#Example%3A+WordCount+v1.0. [Accessed: 28-Sep-2016].

[30] “Grep - Hadoop Wiki.” [Online]. Available: https://wiki.apache.org/hadoop/Grep. [Accessed: 28-Sep-2016].

[31] “TeraGen (Hadoop 1.0.4 API).” [Online]. Available: https://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/examples/terasort/TeraGen.html. [Accessed: 28-Sep-2016].

[32] “google/cluster-data,” *GitHub*. [Online]. Available: https://github.com/google/cluster-data. [Accessed: 23-Nov-2016].

[33] “Apache Hadoop 2.7.2 – Docker Container Executor.” [Online]. Available: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/DockerContainerExecutor.html. [Accessed: 27-Nov-2016].

[34] “The CLOUDS Lab: Flagship Projects - Gridbus and Cloudbus.” [Online]. Available: http://www.cloudbus.org/cloudsim/. [Accessed: 23-Nov-2016].

Appendix A: Workload Samples

Workload sample 1:

```
<? xml version="1.0" encoding="UTF-8" standalone="yes"?>
<workload>
    <application-name>test</application-name>
    <jar-path>{{HADOOP_COMMON_HOME}}/share/hadoop/mapreduce/hadoop-mapreduce-
examples-2.7.2.jar</jar-path>
    <args>
            <arg>wordcount</arg>
            <arg>/user/hduser/input</arg>
            <arg>/output/output5</arg>
    </args>
</workload>
```

Workload sample 2:

```
<? xml version="1.0" encoding="UTF-8" standalone="yes"?>
<workload>
    <application-name>test</application-name>
    <jar-path>{{HADOOP_COMMON_HOME}}/share/hadoop/mapreduce/hadoop-mapreduce-
examples-2.7.2.jar</jar-path>
    <args>
        <arg>grep</arg>
        <arg>/user/hduser/input</arg>
        <arg>/output/grep6</arg>
            <arg>'dfs[a-z.]'</arg>
    </args>
</workload>
```

Appendix B: Submit Application Example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application-submission-context>
  <application-id>application_1421661392788_0040</application-id>
  <application-name>test</application-name>
  <queue>default</queue>
  <priority>3</priority>
  <am-container-spec>
    <environment>
      <entry>
        <key>CLASSPATH</key>
        <value>{{CLASSPATH}}&lt;CPS&gt;./*&lt;CPS&gt;{{HADOOP_CONF_DIR}}&lt;CPS&gt;{{
HADOOP_COMMON_HOME}}/share/hadoop/common/*&lt;CPS&gt;{{HADOOP_COMMON_HOM
E}}/share/hadoop/common/lib/*&lt;CPS&gt;{{HADOOP_HDFS_HOME}}/share/hadoop/hdfs/*&lt;
CPS&gt;{{HADOOP_HDFS_HOME}}/share/hadoop/hdfs/lib/*&lt;CPS&gt;{{HADOOP_YARN_HO
ME}}/share/hadoop/yarn/*&lt;CPS&gt;{{HADOOP_YARN_HOME}}/share/hadoop/yarn/lib/*&lt;C
PS&gt;./log4j.properties</value>
      </entry>
    </environment>
    <commands>
      <command>{{HADOOP_COMMON_HOME}}/bin/hadoop jar
{{HADOOP_COMMON_HOME}}/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar
wordcount /user/hadoop/input /output/output</command>
    </commands>
  </am-container-spec>
  <unmanaged-AM>false</unmanaged-AM>
  <max-app-attempts>2</max-app-attempts>
  <application-type>MAPREDUCE</application-type>
  <keep-containers-across-application-attempts>false</keep-containers-across-application-
attempts>
  <application-tags/>
</application-submission-context>
```

Appendix C: Steps to setup Running Cluster

- Launch instances from OpenStack required for configuring multi-node cluster.
- Hadoop requires a working java 1.5+. However, java 1.6 or higher is recommended for running Hadoop.
- Install java on all the nodes.
- Add a dedicated user for Hadoop system, for example: hduser.
- Configure password-less SSH as Hadoop requires SSH access to manage nodes.

```
hduser@ubuntu:~$ ssh-keygen -t rsa -P ""

Generating public/private rsa key pair.

Enter file in which to save the key (/home/hduser/.ssh/id_rsa):

Created directory '/home/hduser/.ssh'.

Your identification has been saved in /home/hduser/.ssh/id_rsa.

Your public key has been saved in /home/hduser/.ssh/id_rsa.pub.

The key fingerprint is: 9b:82:ea:58:b4:e0:35:d7:ff:19:66:a6:ef:ae:0e:d2

hduser@ubuntu The key's random art image is: [...snipp...]
```

- Keep one node as master and rest of the nodes as slaves.
- Make sure all the machines are able to reach each other over the network. (i.e hduser user on the master must be able to connect to its own host and also to hduser user on the slave node via a password-less SSH login)
- Download Hadoop from the Apache Download Mirrors and extract the contents of the Hadoop package to a location of your choice. I used/usr/local/hadoop. Make sure to change the owner of all the files to the hduser user and hadoop group, for example:

```
$ cd /usr/local

$ sudo tar xzf hadoop-2.7.2.tar.gz

$ sudo mv hadoop-2.7.2 hadoop

$ sudo chown -R hduser:hadoop hadoop
```

- Update ~/.bashrc with path set to Hadoop installation directory.
- Configured hadoop-env.sh, core-site.xml, hdfs-site.xml, yarn-site.xml and mapred-site.xml on all the nodes. See Appendix A for configuring these files.
- Additionally, configured master and slave file only on master node. On master file, master's hostname should be added. On slave file, slave nodes' hostname should be added. If master node has to be used as slave node, then master's hostname should also be added in slave file.

```
hduser@anitha-1:/usr/local/hadoop$ vi /etc/hadoop/master

master

hduser@anitha-1:/usr/local/hadoop$ vi /etc/hadoop/slave

master

slave1
```

slave2

- Before starting the cluster, format the HDFS file system via NameNode. Run the below command to format the NameNode. (Note: Do not format the NameNode when the cluster is running as you will lose all the data in HDFS that are available at present).

  ```
  hduser@ubuntu:~$ /usr/local/hadoop/bin/hadoop namenode -format
  ```

- Run `$HADOOP_INSTALL/sbin/start-dfs.sh` and `$HADOOP_INSTALL/sbin/start-yarn.sh` on the master node.
- If everything is configured correctly, you can see a list of processes running on all the nodes by issuing `jps` command.

The following java processes should run on master.

```
hduser@anitha-1:/usr/local/hadoop$ jps
4065 Jps
820 NameNode
1390 ResourceManager
1545 NodeManager
1235 SecondaryNameNode
992 DataNode
```

The following java processes should run on slaves.

```
hduser@anitha-2:/usr/local/hadoop$ jps
14897 DataNode
15053 NodeManager
16298 Jps
```

- Like starting the cluster, stopping the cluster also done by `stop-dfs.sh` and `stop-yarn.sh` script inside `$HADOOP_HOME/sbin`.

Appendix D: Other Approaches

In this master thesis project, initially our main focus was to develop a coordination framework (as local-level resource coordinators) for server creation based upon disaggregated resources available in various resource pools, e.g. CPU, memory, storage, network bandwidth and data locality-aware consideration. Some tasks considered in specific were:

1) a holistic means to server creation based upon requests derived from different application demands (e.g. CPU-, memory-, IO-intensive applications). We researched and investigated to get real dataset to analyze different application workloads. We collected data from Google Data set [32]. Though the data set contains has lots of parameters including job level and task level data, we could not derive some specific workloads such as Hadoop MR workloads from the data set. Hence, we used the sample workloads that were available as part of the Hadoop distribution and limited the scope of the thesis to these MR workloads. To analyze the MR workloads, we planned to create a Hadoop Cluster. Initially were playing around with the Yarn cluster for quite some time, in order to get a basic understanding of launching Docker Container Executors(DCE) [33] at Yarn node managers. The purpose of this practice is not to set up a testing environment in Yarn, but to see if there will be some extra parameters and resources required by launching the DCE on a Yarn cluster. However, The machine was not powerful enough to host the cluster. So we created an experimental setup using OpenStack. Due to the time consumption in setting up the Hadoop cluster, the experimental core parameters were reduced to 3. They are HDFS block size, Input Splits and Input data size. They are the significant parameters in improving the performance of the job in terms of Job completion time and Memory utilization. Out of these three parameters, HDFS block size is the most significant parameter as increasing the block size to 512MB almost reduced the Job completion time and memory utilization by 1/8th comparing to 64 MB default block size. This was also because the time constraint was critical for implementing the Translator part using RESTFUL APIs.

2) An algorithm for matching requests and available disaggregated resources on hosts to be adopted by our per-request, shared-state coordinators. Although this was the initial plan, since there were no actual disaggregated resources, we created a co ordination framework to compose the server and dimension it (choosing the number of slave nodes in cluster) according to incoming WL resource requirement.

3) Evaluation and performance analysis of the proposed solution. Initially we planned to evaluate the workloads using a tool called CloudSim [34], which is used to model and simulate customized cloud infrastructure and services. Due to the difficulty in using the tool, OpenStack was used for the purpose.

TRITA-ICT-EX-2016:172