Abstract

In this era of BigData, designing a workflow to gain insights from the vast amount of data has become more complex. There are several different frameworks which individually process the batch and streaming data but coordinating the jobs between the engines in the workflow creates a performance penalty and other performance issues. Current workflow systems typically run only on one engine and do not offer the versatility required for today’s workflows. The process of submitting the jobs on different engines manually is not only time consuming, but also requires the expertise of working on these engines.

In this thesis, we have overcome the above mentioned issues by proposing a MEWSE - Multi Engine Workflow Submission and Execution on Apache YARN. It should also have design with plug and play functionalities to allow the inclusion of new engines. MEWSE has been tested on Amazon EC2 with a sample workflow which requires the following engines, Hadoop, Mahout, java and some scripts to process the data.
Acknowledgements

I would like to sincerely thank my supervisor, Prof. Patrick Martin, for guiding and supporting me throughout my time at Queen’s university. His expertise in the fields of Cloud Computing and BigData Analytics systems had encouraged me to complete this research successfully.

I would like to thank Wendy Powley, who patiently reviewed my thesis and her valuable feedback made me complete this thesis successfully.

I would like to thank our IBM colleagues, my lab mates and colleagues for their valuable comments in all the phases of the projects I have gone through.

Finally, I would like to thank the School of Computing and School of Graduate Studies at Queen’s University for providing me with the opportunity to pursue my master’s degree. I am thankful to IBM Canada Ltd. and NSERC CREATE in Ultra Large Scale System program for their financial support to the Database Systems Lab at Queen’s University.
# Contents

Abstract i

Acknowledgements ii

Contents iii

List of Tables v

List of Figures vi

List of Listings vii

Chapter 1: Introduction 1
  1.1 Motivation ........................................... 3
  1.2 Research Statement .................................. 4
  1.3 Thesis Organization .................................. 4

Chapter 2: Background 5
  2.1 Apache Hadoop ........................................... 5
  2.1.1 Apache Hadoop Architecture .......................... 5
  2.2 Apache Yet Another Resource Negotiator ................. 9
  2.2.1 Architecture of Apache YARN ........................ 9
  2.2.2 Apache YARN Ecosystem .............................. 13
  2.3 Workflow .............................................. 15
  2.3.1 Workflow Pattern .................................... 16
  2.3.2 Big Data Analytic Workflow and Its Representation .... 17
  2.4 Workflow Systems ...................................... 18
  2.4.1 Apache Oozie ........................................ 18
  2.4.2 Wings/Pegasus ....................................... 19
  2.4.3 IBM InfoSphere ..................................... 20
  2.4.4 Kepler ............................................... 21
Chapter 3: Multi Engine Workflow Submission and Execution 23
  3.1 Proposed Architecture ........................................... 23
    3.1.1 Workflow Submitter ....................................... 24
    3.1.2 Workflow Execution Manager .............................. 27
  3.2 Representation of Workflow .................................... 28
  3.3 Life Cycle of a job ........................................... 29
  3.4 Workflow Submission .......................................... 31
  3.5 Adding Frameworks ........................................... 33

Chapter 4: Test Case Workflow 38
  4.1 Example Workflow Representation .............................. 38
  4.2 Experimental Setup ........................................... 41
  4.3 Submit and monitor the Workflow ............................. 43
  4.4 Discussion ................................................... 43

Chapter 5: Conclusions and Future Work 48

References 50

Appendix A: Apache Hadoop 55

Appendix B: Apache YARN 57

Appendix C: Representation of Jobs 59

Appendix D: Test Case Workflow Representation 62

Appendix E: Injecting New Frameworks 66
List of Tables

3.1 Properties of the job object ........................................... 26

4.1 Operations involved in jobs in the example workflow ............... 40
4.2 Configuration of Amazon EC2 Machines ............................ 41
List of Figures

1.1 Big Data Analytic Workflow System ............................................ 2

2.1 Apache Hadoop Architecture .......................................................... 6
2.2 Apache Hadoop Distributed File System ........................................... 7
2.3 Apache YARN Architecture .............................................................. 10
2.4 Apache YARN Ecosystem ................................................................. 13
2.5 Example Analytic Workflow in Graphical Representation .................... 15

3.1 Proposed Architecture ................................................................. 24
3.2 Graphical representation of the workflow ........................................... 25
3.3 Life cycle of a job ........................................................................... 30

4.1 Example Workflow in Graphical Representation .................................... 39
4.2 The configurations of Hadoop Distributed File system .......................... 45
4.3 Before executing the sample workflow .............................................. 46
4.4 Execution of the sample workflow ................................................... 47

A.1 MapReduce Job in Hadoop ............................................................... 56

B.1 Apache YARN walk-through with an example ..................................... 57
List of Listings

3.1 Representation of the example workflow in XML language ........................ 28
3.2 Algorithm that submits Workflow’s jobs on Apache YARN ........................ 32
3.3 Parse method of Hadoop MapReduce job .................................................. 33
3.4 Submit method of Hadoop MapReduce job ................................................. 34
3.5 isJobComplete method of Hadoop MapReduce job ...................................... 35
3.6 afterJobCompletion method of Hadoop MapReduce job ............................... 36
C.1 Representation of Mahout KMeans Clustering Job ....................................... 59
C.2 Representation of Mahout Build Forest Job ................................................ 60
C.3 Representation of Mahout Test Forest Job .................................................. 60
C.4 Representation of Java job .......................................................................... 61
D.1 Representation of the test case workflow .................................................... 62
E.1 Representation of Spark job in XML ............................................................ 67
E.2 Parse method of Spark job .......................................................................... 67
E.3 Submit method of Spark job ........................................................................ 68
E.4 isJobComplete method of Spark Job ........................................................ 69
Chapter 1

Introduction

From the advent of Web 2.0, the rate of growth of data has been exponentially increasing. Companies like Google, Facebook, IBM, etc are using this data to gain insights for marketing purposes or to provide better services for their customers. Analysing data requires the creation and management of workflows which, given the vast amount of data and the numerous analysis tools available, can be a complex process. Workflows connect a series of business activities that execute on a set of resources. Modern workflow management systems require several different engines to process the huge amount of data. There are many issues in the existing Workflow management systems, such as data movement cost across the different engines used by the workflow or the inability of the file system to support different engines.

Analytic workflows are the workflows which analyse and gain insights from the vast amount of data. MapReduce [10] is a paradigm, which has mapper and reducer phases to process huge amount of batch data in a distributed manner which is published by Google. The MapReduce paradigm has been implemented in Apache Hadoop [11], which is an open source framework. Apache Hadoop is designed to process huge amount of batch data on commodity hardware with the use of the
Hadoop Distributed File System [22]. After the wide use of Hadoop framework in several organizations, it has been found that Apache Hadoop is restricted to MapReduce paradigm. The shortcomings of Hadoop framework are discussed in Section 2.2. Dealing with these shortcomings led to the invention of Apache Yet Another Resource Negotiator (YARN) [26], which is designed to support for non MapReduce applications on the Hadoop Distributed File System.

![Diagram of Big Data Analytic Workflow System](image)

**Figure 1.1: Big Data Analytic Workflow System**

Figure 1.1 represents a typical Big Data analytic workflow system where end users and business analysts submit queries to developers and data analysts from which they would like to gain insights from the data. Developers and data analysts collaborate with each other and design a workflow to answer the user-submitted queries. Then the designed workflow is submitted to the workflow scheduler. The workflow scheduler is responsible for submitting every individual job in the workflow on the execution layer. It is also responsible for monitoring the status of jobs which are submitted on
the execution layer. The execution layer consists of Big Data engines to process the data and file system to store the actual data. After the successful execution of all jobs in the workflow, the execution layer will send the results to end users and business analysts.

In this research, we present a framework that allows users to develop workflows that involve a variety of different engines on distributed resource scheduler framework like Apache YARN.

The remainder of the chapter is organized as follows; Section 1.1 explains the motivation of the research, Section 1.2 defines the proposed research statement and Section 1.3 explains the organization of the thesis.

1.1 Motivation

Analytic workflow management systems available today do not generally support different varieties of engines and typically require a great deal of expertise. Examples of workflow management systems include Apache Oozie [15], Wings/Pegasus [8], IBM InfoSphere [3], Kepler [27]. Apache Oozie is designed to run Hadoop jobs on a Hadoop cluster while systems like Wings/Pegasus have an expensive data movement cost across different engines. Hence, the motivation of this research is to coordinate between the different engines involved in the workflow and to execute the workflow on the distributed resource scheduling framework like Apache YARN. The main challenging part of this research is to implement a generic framework that supports a wide variety of engines in the workflow and schedules the jobs on Apache YARN.
1.2 Research Statement

The main goal of this research is to build a generic tool to allow users to submit the workflow’s jobs on the distributed resource scheduling framework like Apache YARN. Jobs which require different engines to process the data must be able to run on a single cluster with a distributed file system.

The major contributions of this research are,

- To provide a submission framework that submits jobs of a workflow on multiple engines running on Apache YARN.

- To provide a framework that should be adaptable to allow the inclusion of multiple analytic engines.

1.3 Thesis Organization

The thesis is organized as follows; Chapter 2 explains the background for the research; Chapter 3 describes Multi Engine Workflow Submission and Execution (MEWSE) framework; Chapter 4 presents a use case study with an implementation of MEWSE and Chapter 5 summarizes the thesis and discusses future work.
Chapter 2

Background

This research is in the field of Big Data processing and Workflow Analytics. In this chapter, we discuss the related work pertaining to our research. This chapter is organized as follows: the Hadoop ecosystem is presented in Section 2.1, the Apache Yet Another Resource negotiator is discussed in Section 2.2, Section 2.3 defines the workflow and its representation and existing workflow systems are described in Section 2.4.

2.1 Apache Hadoop

Apache Hadoop [11] is an open source implementation of MapReduce. MapReduce is a paradigm in which data are processed in two phases such as, Map and Reduce. A detailed explanation of MapReduce is given in subsection 2.1.1. Apache Hadoop is designed to process large amount of batch data with high data availability and has a fully fault tolerant mechanism.

2.1.1 Apache Hadoop Architecture

Apache Hadoop’s Architecture is divided into two layers,
2.1. APACHE HADOOP

- The storage Layer (Hadoop Distributed File System) which stores the actual data in the physical nodes.

- the MapReduce Engine which processes the batch data.

Figure 2.1 illustrates the components of the Apache Hadoop architecture.

![Figure 2.1: Apache Hadoop Architecture](image_url)

**Hadoop Distributed File System**

The Hadoop Distributed File System (HDFS) [22] is designed to store vast amounts of data in a distributed manner with high availability and efficient fault tolerance. It is designed to run in commodity hardware. Figure 2.2 illustrates the HDFS components.
Figure 2.2: Apache Hadoop Distributed File System

HDFS contains two important components: the NameNode and the DataNode. The NameNode is a master daemon service, whereas the DataNode is a worker daemon service. The NameNode stores the metadata of the data and the DataNodes store the actual data. The NameNode holds the details of the Inodes (like in the UNIX file system) of the files which contains permission, unique block number, disk space, etc. DataNodes store the files in block level storage. The size of a block (64MB, 128MB, 512MB, etc.) is customizable. The DataNodes register with the NameNode using a unique ID called a storage ID. If the cluster nodes restart, the change in the IP address will therefore not affect the NameNode service. The DataNodes periodically send heartbeat signals to the NameNode and the heartbeat signal holds the information about the DataNode such as, disk space, data transfers currently in progress and so on. If a DataNode fails to send heartbeat signals to the NameNode for 10 minutes, the NameNode assumes that the DataNode is down and the NameNode replicates the data onto other DataNodes to ensure data availability.
2.1. APACHE HADOOP

MapReduce Engine

The MapReduce Engine is used to process a large amount of data in a distributed manner with the help of two phases; the Map phase, and the Reduce phase.

In the Map phase, the given input data will be transformed into a <key, value> pair and the Reduce phase processes the intermediate result, which is generated in the Map phase and emits the final result. The MapReduce programming paradigm is inspired from the functional programming paradigm. In Hadoop, the MapReduce engine contains two major components: the JobTracker, and the TaskTracker.

The JobTracker is a master node daemon, whereas the TaskTracker is a worker node daemon. The JobTracker monitors the status of the physical nodes in the cluster as well as the status of jobs in execution. The JobTracker accepts the MapReduce jobs and assigns the job to the respective TaskTrackers based on scheduling algorithms [30]. Some of the popular job scheduling algorithms are FIFO (First In First Out), Fair Scheduler and Capacity Scheduler. Each job consists of several map and reduce tasks. The actual tasks are executed by the TaskTracker daemon. The TaskTracker updates its information (status of the task in execution) with the JobTracker using heartbeat signals. In response to the heartbeat signals, the JobTracker sends the next task to be executed. If a TaskTracker fails to send heartbeat signals to the JobTracker or if the JobTracker finds any straggler tasks in execution, the JobTracker resubmits the respective tasks to another TaskTracker. In Hadoop, tasks are executed in fixed slots. Slots are actual resources (1GB, 1CPU, etc) which are set along with the cluster configuration. Unfortunately it has been discovered that the slot mechanism is a bottleneck in this environment [26].

Appendix A describes a walk-through of Apache Hadoop with a simple example.
2.2 Apache Yet Another Resource Negotiator

Apache Yet Another Resource Negotiator (YARN) [26] is an open source framework that provides a resource management service on Apache Hadoop clusters. There are two shortcomings in traditional Hadoop:

1. It tightly links a specific programming model (MapReduce) with the resource management infrastructure, thus forcing developers to restrict themselves to the MapReduce programming model.

2. The JobTracker [11] schedules and monitors the jobs as well as monitors the cluster resources, which affects the performance of the tasks.

The design goal of Apache YARN is to allow non-MapReduce applications to utilize the HDFS (Hadoop Distributed File System). This goal is achieved by separating resource management from the programming model and by delegating many scheduling-related functions to per-job components. In other words, YARN separates the JobTracker’s properties into two services, namely the Resource Manager, which manages the cluster’s resources and the Application Master, which is a per application master that tracks the status of a submitted job in the YARN cluster. The TaskTracker [11] has been replaced by the NodeManager, which replaces the fixed slot mechanism of the classic Hadoop. The following subsections discuss the architecture of Apache YARN.

2.2.1 Architecture of Apache YARN

Apache YARN’s architecture contains 4 important components, shown in Figure 2.3, namely the Container, the Resource Manager, the Application Master and the Node
2.2. APACHE YET ANOTHER RESOURCE NEGOTIATOR

Manager.

![Diagram of Apache YARN Architecture](image)

Figure 2.3: Apache YARN Architecture [26]

**Container**

A Container is a logical bundle of resources in a physical node of the Apache YARN cluster. The Containers are responsible for executing tasks of a job. In a classic MapReduce system, fixed slots are used to run the mapper and reducer tasks, which offers poor resource utilization. Apache YARN has changed the fixed slot mechanism to dynamic resource allocation with the help of Containers [26]. Containers can be customized at run time by requesting the amount of resources (e.g., 2GB, 1CPU) needed to execute the tasks.
2.2. APACHE YET ANOTHER RESOURCE NEGOTIATOR

Resource Manager

The Resource Manager is a per cluster service which deals with two responsibilities as follows;

1. Creates an Application Master for the job, which executes actual tasks of the job on the cluster,

2. Satisfying the resource requests from the Application Masters and obtains the node information from the Node Managers using heartbeat signals.

The Resource Manager allocates resources to the applications using a global scheduler. A resource request [26] which is received from an Application Master contains the following;

1. Number of containers (e.g., 200 containers);

2. Resources per container (e.g., 2GB RAM, 1 CPU);

3. Locality preferences (e.g., Hostname);

4. Priority of requests within the application.

The Resource Manager monitors the status of the nodes in the cluster with the help of the Node Managers. The Resource Manager is responsible for allocating resources, marking failed nodes and stopping the resource allocation to these failed nodes. It is not responsible for providing the status of an application’s metrics and an application’s task fault tolerance.
2.2. APACHE YET ANOTHER RESOURCE NEGOTIATOR

Application Master

The Application Master is a per application service that helps to execute the application in the YARN cluster. An Application Master launches in another container in the cluster and periodically sends its liveness to the Resource Manager. It sends resource requests to the Resource Manager for containers to execute the application’s tasks. When an Application Master gets the requested resources, it launches the containers in the worker nodes with the help of the Node Managers. After executing the tasks, the Application Master informs the Resource Manager and the Resource Manager deletes the container with the help of the Node Managers. The Application Master is also responsible for monitoring the status of the application and it is dealing with application’s task fault tolerance.

Node Manager

A Node Manager is a Worker daemon and a per physical node service that helps to launch and kill the containers of a physical node. All the containers in YARN, including an Application master, are deployed with the help of a “Container Launch Context (CLC)”. A CLC is a script that loads the prerequisites of the task to be executed in the container. The Node Manager is responsible for monitoring the container’s status and preventing over utilization of resources. The Node Manager also offers local services such as log aggregation.

Appendix B describes a walk-through of Apache YARN with a simple example.
2.2.2 Apache YARN Ecosystem

Apache YARN ecosystem is the set of frameworks which are able to run on Apache YARN. Apache YARN extends its usability for non-MapReduce applications, so that they can utilize the benefits of the Hadoop Distributed File System (HDFS) which is capable of storing large quantities of data with efficient fault tolerance. Since Apache YARN allows MapReduce and non-MapReduce application to make use of the benefits of the HDFS, workflows that make use of different engines can be coordinated. Figure 2.4 illustrates the other frameworks that run on Apache YARN.

![Apache YARN Ecosystem Diagram](image)

**Figure 2.4: Apache YARN Ecosystem**

Apache PIG

Apache Pig [19] is a platform to analyze large amounts of batch data. Its infrastructure layer consists of a compiler which translates sequences of jobs into MapReduce jobs. Jobs are written in a script language called Pig Latin.
Apache HIVE

Like Apache Pig, Apache Hive [23] is a platform to analyze large amounts of batch data. Hive is a schema oriented database which stores structured data in the Hadoop Distributed File System (HDFS). Jobs are expressed in a SQL representation called HiveQL and translated into MapReduce jobs.

HBase

Apache HBase [9] is an open source implementation of Google’s BigTable [5]. The primary design goal of BigTable is to provide a fault tolerant way of storing large amounts of sparse data. HBase is designed to run on HDFS to make use of its fault tolerant mechanism. HBase has been adopted by Facebook for its messaging service system [12].

Storm

Apache Storm [21] is an open source distributed real time system. It is designed to process huge amounts of real time data. It supports a wide varieties of programming languages. It is highly scalable and fault-tolerant. It has been widely adopted for real time analytics, online machine learning, ETL and more.

Spark

Apache Spark [31] is a fast in-memory processing system for streaming and batch data. Spark is designed to provide high fault tolerance with the help of Resilient Distributed Datasets. It runs on distributed resource scheduling frameworks, such as Apache YARN and Mesos [13], etc.
2.3 Workflow

A “Workflow” can be defined as a series of business activities, which are mapped to execute on a set of resources and there may or may not be an outcome in a workflow [24]. The term “Workflow” is used here as a synonym for “business process”, which is standardized by the Workflow Management Coalition (WFMC). This is an organization dedicated to developing standard terminology and standard interfaces for Workflow management system components. Figure 2.5 illustrates a simple representation of analytic workflow that processes unstructured data with multiple engines.

![Figure 2.5: Example Analytic Workflow in Graphical Representation](image)

Figure 2.5: Example Analytic Workflow in Graphical Representation
2.3. WORKFLOW

2.3.1 Workflow Pattern

Workflow patterns can be expressed as a Directed Acyclic Graph (DAG) structure, where each job has to wait for the completion of its predecessor jobs. In workflow systems, the predecessors of a job are those jobs that must run to produce the direct input to the job. Similarly, the successors of a job are those jobs whose direct input include the output of the job. Dependencies of the jobs are represented in a workflow DAG by directed edges. Jobs with no dependencies at the same level can be executed in parallel. Parallel execution of the jobs in the workflow enhances the overall completion time of the workflow [25]. In general, workflow patterns can be broadly classified into two types;

- Control flow workflow pattern, and
- Data flow workflow pattern

Control Flow Workflow Pattern

The Control Flow workflow pattern describes the execution order of the jobs in the workflow through different constructors. There are several constructors that control the execution of the jobs in the workflow [25] including;

- Sequence: All predecessors jobs must to be completed before the execution of current job.
- Choice: Decision making between various paths in the workflow.
- Parallelism: Independent jobs can execute in parallel.
- Join synchronization: Merge output of two jobs after their execution.
2.3. WORKFLOW

Control flow patterns originated from Unix shell scripts, where a series of jobs is executed with the help of a script program.

Data Flow Workflow Pattern

The Data Flow workflow pattern describes the data flow between the jobs in the workflow. It allows jobs to run in parallel when there are no data dependencies between them. Typically, workflow data are passed through applications by the way of application interfaces, allowing manipulation of the data within applications.

2.3.2 Big Data Analytic Workflow and Its Representation

Big Data Analytics Workflow, also called the “Hybrid flow Workflow” pattern, requires both the Control flow and Data flow design patterns. Most of the BigData Analytics workflows involve large amounts of data with complex workflow design patterns requiring multiple engines in a distributed environment. In general, BigData Analytics workflows can be represented as graph structures, where nodes represent the activities of the workflow and the edges between the jobs in the workflow indicate the dependencies.

Analytic workflows that uses Hadoop jobs are called Hadoop workflows. In Hadoop workflows, instead of a script, mapper and reducer classes are provided, since all the jobs in the workflow are going to execute in MapReduce paradigm on the Hadoop Distributed File System. For Example, Hadoop workflows [17] can be represented as follows;

- Workflows $W=\{w_1, w_2, \ldots, w_l\}$, where $W$ is a set of workflows and the $w_c$’s compete for resources on the Hadoop cluster.
Workflow $w_i$ contains a set of $N$ jobs - $J_i = \{J_{i1}, J_{i2}, J_{i3}, ..., J_{iN}\}$, where each job contains input dataset path to be processed, mapper and reducer classes and output result path.

- Each job $J_{ij}$ contains a set of mappers, $M_{ij} = \{m_{ij1}, m_{ij2}, ..., m_{ijK}\}$ and a set of reducers $R_{ij} = \{r_{ij1}, r_{ij2}, ..., r_{ijL}\}$.

- Each job $J_{ij}$ has a predecessor set $P_{ij}$. If job $J_{ik}$ is in $P_{ij}$ then $J_{ik}$ must complete its execution before $J_{ij}$ can start. The root of a DAG has an empty predecessor set.

Considering all these assumptions, Hadoop Workflows can be formalized as follows,

$$W_i = \{J_i, P_i\}$$

Similarly, we can convert any BigData frameworks’ Workflow to a DAG (Directed - Acyclic Graph) structure, which will be further processed with the help of Workflow design patterns to get the processed information.

2.4 Workflow Systems

A number of workflow systems exist to solve many complex workflows. In this section, some of the popular workflow Systems are discussed including Apache Oozie, Wings/Pegasus, IBM InfoSphere and Kepler.

2.4.1 Apache Oozie

Apache Oozie [15] is a workflow scheduler system for Hadoop jobs. Initially it was developed by the Yahoo! community, but later the project was migrated to the Apache
open source community. Apache Oozie allows construction a directed cyclic graph of Hadoop jobs. It submits the jobs to a Hadoop cluster based on data availability of the jobs or time deadlines. Apache Oozie provides an interactive user interface and also provides APIs to integrate with other Hadoop ecosystem frameworks like Pig, Hive and Java MapReduce.

Apache Oozie uses the HDFS as its storage system and periodically checkpoints the jobs in the workflow to the persistent storage with the help of a database. In terms of flow control operations, Oozie supports conditionals, fork and join along with start, stop, suspend, resume and rerun operations but cycles are not supported. Unlike other workflow systems, Oozie provides Software as a Service, where the user can create/manage the workflow without considering the Hadoop cluster deployment. Oozie provides a programmatic workflow invocation mechanism via the Oozie Web Services APIs or Oozie Java client APIs.

Since Apache Oozie is designed to run for Hadoop jobs, the workflow template in Oozie is designed more specifically for the Hadoop architecture, which turns out to be its limitation. It does not support non MapReduce applications and does not allow to plug in new frameworks. Since Apache Oozie runs on the top of the Hadoop cluster, it does not consider the resource allocation of the workflow jobs in the Hadoop cluster.

2.4.2 Wings/Pegasus

Wings [8] is an Intelligent workflow management system which uses artificial intelligence to help users construct and execute workflows. It also validates a workflow by compiling the components and data set required by the workflow. Wings uses W3Cs Web Ontology Language (OWL4) [2], Resource Description Framework (RDF5) [20]
and Semantic Web Rule Language (SWRL6) [14] to represent Workflows and their associated constraints. In Wings, there are three types of users, namely,

1. Software developers, who write and execute the components of the Workflow.

2. Experienced users, who develop the templates of the Workflow. They connect the components in the Workflow with different input paths and configuration.

3. Less experienced users, who create and execute instances of existing workflows with different available datasets.

Wings allows for the parallel execution of components with the help of Pegasus [7]. Pegasus is a resource management framework, which allocates resources to provide an execution platform for the workflows. Before executing the workflow, Pegasus runs a set of workflow optimization operations. In the case of large complex workflows, Pegasus can partition the workflow to a number of smaller sub-workflows and run further refinement operations on each individual sub-workflow. Wings ensures the Workflow’s components are valid with available datasets, which helps to provide a recovery system at component levels [6].

Wings provides APIs in several programming languages which allows user to develop the workflow in their own language and then integrate it with Wings. Wings works well with DAG structured workflows but does not provide support for loops and choice operations.

2.4.3 IBM InfoSphere

IBM InfoSphere [3] provides a proprietary workflow management framework that allows the user to create and manage workflows for streaming applications. It provides
interactive drag and drop features to ease the workflow creation process. IBM InfoSphere enables users to create, deploy, schedule, execute, monitor and visualize the Workflows in real time. With the help of a highly interactive user interface, it allows a less experienced user to work with workflows easily.

Workflows are represented using the Streaming Programming Language (SPL) [3], which is a high level abstract programming language for creating workflows. IBM InfoSphere deals with large amounts of real time streaming data. The data is processed in real time with an appropriate query and the resulting data is sent to the appropriate components. IBM InfoSphere provides solutions to process huge amount of historical data (i.e. stored data) with the help of other IBM systems like IBM SPSS Modeler [18] and IBM BigInsights [4].

IBM InfoSphere provides a recovery system by checkpointing the workflow at intermediate stages between jobs. Parallel execution of the workflow is possible provided if there is no data dependency between jobs. It uses IBM General Parallel File System (GPFS) instead of the Hadoop Distributed File System (HDFS) for replicating the data, so there is no single node point failure.

2.4.4 Kepler

Kepler [1] is a workflow management system that allows the user to design, create and execute the workflow interactively. It also allows the execution of workflows in heterogeneous environments such as Web services, Grid environment or local machines.

Kepler manages data flow across the jobs in the workflow, so the user does not need to worry about the intermediate dataset. However, Kepler does not run in a distributed shared file storage, it downloads and runs the dataset used by the job
2.4. WORKFLOW SYSTEMS

on the local machine. Kepler uses Modeling Markup Language (MoML) [16], which allows the workflows to manage multiple versions of the data and archive the data.

An extension to Kepler proposed by J. Wang et al. [27], which enables Kepler users to easily create and manage MapReduce-like workflows. This extension allows the users to create MapReduce jobs in the Workflow with the help of any programming language that supports the MapReduce paradigm. The template can be saved and later reused.

From these existing workflow systems, I have drawn the conclusion that none of the existing workflow systems are allowing a workflow with multiple analytic engines and are not adaptable to allow the inclusion of multiple analytic engines. MEWSE is designed to handle the above mentioned shortcomings.
Chapter 3

Multi Engine Workflow Submission and Execution

The challenges of this research are to create a framework to submit and execute the workflows on Apache YARN that have multiple engines. It should also have design with plug and play functionalities to allow the inclusion of new engines. In this Chapter, we discuss our solution for the above mentioned issues.

The chapter is organized as follows: in Section 3.1, we describe the system architecture, Section 3.3 we discuss the workflow job’s life cycle. The representation of the Workflows, submission of the Workflow and the addition of a new engine to the MEWSE framework are discussed in Sections 3.2, 3.4 and 3.5 respectively.

3.1 Proposed Architecture

Figure 3.1 illustrates the components of the proposed framework. It is divided into two sections: the Workflow Submitter (WS) and the Workflow Execution (WE).

1. The WS parses and submits the workflow’s jobs on the Apache YARN cluster.

2. The WE component executes the actual workflow’s jobs on the Apache YARN cluster.
The components WS and WE are discussed in the subsection 3.1.1 and 3.1.2 respectively.

### 3.1.1 Workflow Submitter

The Workflow Submitter (WS) translates the user’s representation of the workflow to a format understood by the system. The user represents the workflow in an Extensible Markup Language (XML) format, where the workflow will be in a Directed Acyclic Graph (DAG) structure. The representation of workflow by the user is discussed in Section 3.2. The WSE consists of three important components: the Workflow Specification, the Workflow Parser and the Workflow Submitter Queues.
The following subsections are explained with the example workflow 3.2.

**Workflow Specification Component**

The major role of the Workflow Specification component is to check the syntax of the workflow. The client submits the XML format of the workflow to the Workflow specification component which checks the correctness of the workflow and sends the XML format of the workflow to the Workflow Parser component. By default, every job in the workflow has jobName and predecessors tags. Type of the job is defined in the job attribute. Here type is known as type of the engine use by the job. It also has optional parameters such as input, output and jar type tags, which are varying with different types of jobs.
3.1. PROPOSED ARCHITECTURE

Workflow Parser Component

The Workflow Parser Component parses the workflow, which is represented in XML language. The parameters of each individual job in the Workflow are extracted and used to create a unique job object in the system. For example, if the workflow has 10 different jobs with different configurations, the Workflow Parser parses these jobs and creates 10 different job objects in the system. Each job object holds the information about the job, which is explained in the table 3.1.

Table 3.1: Properties of the job object

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job type</td>
<td>Type of the job based on engine used</td>
</tr>
<tr>
<td>Input</td>
<td>Input dataset path</td>
</tr>
<tr>
<td>Output</td>
<td>Output dataset path</td>
</tr>
<tr>
<td>Predecessors</td>
<td>Holds the predecessors job name, which is used to identify the dependencies between jobs</td>
</tr>
<tr>
<td>Jar</td>
<td>Jar path that holds the main class file</td>
</tr>
<tr>
<td>Class</td>
<td>Classname that holds the main method</td>
</tr>
<tr>
<td>Arguments</td>
<td>Required arguments for the main class</td>
</tr>
<tr>
<td>other</td>
<td>Based on job type, parameters can vary.</td>
</tr>
<tr>
<td></td>
<td>For example, MapReduce job requires Mapper and reducer classes.</td>
</tr>
</tbody>
</table>

After the Workflow Parser successfully parses the workflow, it sends all the job objects to the Workflow Submitter Queues.
3.1. PROPOSED ARCHITECTURE

**Workflow Submitter Queues**

The main goal of the Workflow Submitter Queues is to submit the jobs in the workflow to the actual execution engine (Apache YARN). The Workflow Submitter Queues are made up of 2 queues;

- The Waiting Queue, which holds the jobs whose parent jobs are yet to be completed.
- The Ready Queue, which contains the jobs ready to be submitted.

With the help of these queues, MEWSE processes the workflow, which is in a DAG like structure. An overall workflow submission is discussed in Section 3.4.

3.1.2 **Workflow Execution Manager**

The Workflow Execution Engine executes the Workflow’s jobs in the actual cluster. It consists of three different layers, the engines (e.g. MapReduce, Mahout and Spark), Apache YARN and HDFS.

From Section 2.2, it is clear that Apache YARN runs on the Hadoop Distributed File System, which is designed to process more batch data than typical MapReduce Applications. Apache YARN acts as a Resource Management operating system on top of the Hadoop Distributed File System. Several engines, such as, Hadoop MapReduce, Hive, Spark, Mahout, etc. are designed to run on Apache YARN. Since all the services provided by Apache YARN are extensible, it is easy to convert any other parallel processing framework to run on Apache YARN by defining an Application Master (AM) and the job that is to be run on the actual node. The Application Master is a per job service which coordinates application’s tasks on Apache YARN. Hence,
3.2. REPRESENTATION OF WORKFLOW

Workflow Execution Engine is responsible for coordinating the workflow’s jobs in the actual cluster.

3.2 Representation of Workflow

MEWSE uses the Extensible Markup Language (XML) to represent the workflow. In general, a job in the workflow holds the information like type, name, jar file to be processed, input and output paths and predecessors which connect the job with another jobs. The XML code 3.1 represents the textual representation of the workflow (Figure 3.2).

Listing 3.1: Representation of the example workflow in XML language

```
<workflow>
  <workflowName>WorkflowExample</workflowName>
  <job type="JavaClass">
    <jobName>RemoveLabel</jobName>
    <javaClassName>test.RemoveLabel</javaClassName>
    <arguments>
      <argument><value>NACRS_final_short4.txt</value>
      <argument><value>input.txt</value>
    </arguments>
    <predecessor/></predecessor>
  </job>
  <job type="MahoutKMeansCluster">
    <jobName>MahoutKMeansCluster</jobName>
    <input>input.txt</input>
    <output>input−kmeans</output>
    <distanceMeasure>org.apache.mahout.common.distance.CosineDistanceMeasure</distanceMeasure>
    <kPoints>3</kPoints>
    <iterations>100</iterations>
    <clusterPath>input−kmeans−cluster</clusterPath>
    <predecessor>RemoveLabel</predecessor>
  </job>
</workflow>
```
3.3. LIFE CYCLE OF A JOB

From Code 3.1, each job has job name and predecessors tags. Type of the job is mentioned in the attribute of the job. Job with dependencies are identified by using predecessors tag.

Appendix C describes the representation of jobs such as, Apache Mahout - classification and clustering and Java programs.

3.3 Life Cycle of a job

Each job in the Workflow follows different states in the proposed solution. The life cycle of a job in the Workflow is illustrated in Figure 3.3.
3.3. LIFE CYCLE OF A JOB

As soon as the client submits the workflow, all the jobs in the workflow enter into the Parse state. The Parse state is a state in which the description (job name, job type, input, output, jars, predecessors, etc) of the jobs are parsed and stored in the object. Once all the jobs have successfully been parsed, the system finds the root jobs and moves all the root jobs to the ready state. Jobs with dependencies that need to wait for the completion of others are moved to the waiting state. When all parent jobs of a job in the waiting state have successfully completed execution, then the job moves to the ready state. If a job is in the ready state, it waits for the resources to be allocated from the Apache YARN framework to execute the job. After Apache YARN allocates resources to the jobs in the ready state, the job moves to the Running state. The Running state is a state in which jobs execute on the actual Apache YARN cluster. If a job is executed successfully on the Apache YARN cluster, it moves to the exit state. At this point a job performs some cleaning up actions such as copying the local data from the temporary directory to the working directory, saving the terminal (System.out) data into a file, etc. If a job fails to execute, it terminates the program abnormally irrespective of the program exceptions. The data which is required by the job to process, will be stored on the Hadoop Distributed File System. Hence, the data is stored in the cluster and the availability of data will be taken care of Hadoop
Distributed File System.

3.4 Workflow Submission

Each state in a workflow job is defined as a Java method. The algorithm shown in Code 3.2 explains the submission phase of the workflow to the actual execution engine. The Algorithm takes as input the description of the workflow which is defined in the XML language and submits the jobs in the workflow on Apache YARN. It creates an object for each individual job in the workflow and stores the respective parameter values in it. The root jobs are identified and submitted to the ready queue. Successors of the root job are pushed into the waiting queue. If all the parent jobs have successfully executed then a job is moved from the waiting queue will be moved to the ready queue. Currently, the Submitter continuously checks for the completion of parent jobs, which may cause busy waiting issue (over utilization of CPU cycles). This issue can be overcome by using an event based mechanism. The entire process continues until the waiting queue becomes empty. If all the jobs in the workflow have been successfully executed then the algorithm returns a success status otherwise it returns a failure status along with the cause of the failure.

If a job fails at any level, Apache YARN tries to resubmit the job on the cluster for certain number of times. If a job fails even after several retries of submission, MEWSE tries to fetch the cause of the exception and display it to the client.
Listing 3.2: Algorithm that submits Workflow’s jobs on Apache YARN

---

**INPUT:** workflow definition XML path  
**OUTPUT:** Status of the submitted workflow (success or failure)  
waitQueue: Waiting jobs whose dependant jobs are not yet completed.  
rootJobs: Jobs with no dependencies.  
childJob: Successor of the current job.  
lookUpTable: It stores properties of the job such as job name, predecessors, status and jar path.  

**Def submit_workflow (workflowXmlPath):**

```python
waitQueue = []
createLookUp(workflowXmlPath)  // Parses and stores the parameters
rootJobs = findRootJobs(lookUpTable)  // push the successor of root jobs in waitQueue
for each job in rootJobs:
    Begin
        for each childJob in job.successors:
            Begin
                waitQueue.push(childJob)
            End
        End
    submitJob(job, lookUpTable)  // submit job to the YARN cluster.
    Job.status=submitted
End
for each job in waitQueue:
    Begin
        If all predecessors of the current job is completed then
            For each childJob in job.successors:
                Begin
                    waitQueue.push(childJob)
                End
        End
End
```
3.5 Adding Frameworks

The jobs in the workflow follow several states. The states of the different types of jobs are defined using Java. In the proposed framework to add a new type of workflows for execution on YARN, it is enough to define parse, submit, isJobComplete and afterJobComplete methods. These methods help to coordinate the job with other types of jobs. For example, the following snippets describe the injection of Hadoop MapReduce framework into the proposed framework.

1. Parsing - 3.3 defines the parsing method for a Hadoop MapReduce job.

```
Listing 3.3: Parse method of Hadoop MapReduce job

void parseXML(Node jobDetail) throws
ClassNotFoundException, DOMException
{
    if(jobDetail.getNodeType() == Node.ELEMENT_NODE &&
        this.jobType.equals("HadoopMR"))
    {
        /* Parse XML and get the requirement value to */
        run
    }
```

submitJob(job, lookUpTable)
Job.status=submitted
endif
End
endfor
End
endwhile
// Check the status of all jobs
If all job.status=completed:
    return success
else
    return failure
```
3.5. ADDING FRAMEWORKS

**Hadoop jobs in the YARN cluster */

```java
Element jobElement = (Element) jobDetail;
this.setJobName(new String(jobElement
  .getElementsByTagName("jobName") .item(0)
  .getTextContent()));
this.setMapperClass(new String(jobElement
  .getElementsByTagName("mapperClass") .item
  .getTextContent()));
this.setReducerClass(new String(jobElement
  .getElementsByTagName("reducerClass") .item
  .getTextContent()));
this.setInputPaths(new String(inputPathElement
  .getElementsByTagName("path") .item
  .getTextContent()));
this.setOutputPath(new String(jobElement
  .getElementsByTagName("outputPath") .item(0)
  .getTextContent()));
String predecessorSet = new String(jobElement
  .getElementsByTagName("predecessor") .item
  .getTextContent());
}
```

The above snippet parses the parameters of the Hadoop job such as job name, paths of main jar, mapper, combiner and reducer classes, input and output paths from the user defined Workflow and creates a unique job object.

2. Submit Method for Hadoop MapReduce job in Code 3.4

Listing 3.4: Submit method of Hadoop MapReduce job

```java
void submitJob() throws IOException,
  ClassNotFoundException, InterruptedException
{
  if (this.jobType.equals("HadoopMR"))
  {
    this.clientJob.setJobName(this.getJobName());
    this.clientJob.setJarByClass(this.getJobClass());
  }
```
3.5. ADDING FRAMEWORKS

```java
this.clientJob.setOutputKeyClass(this.setOutputKeyClass());
this.clientJob.setOutputValueClass(this.setOutputValueClass());
this.clientJob.setMapperClass(this.getMapperClass());
this.clientJob.setCombinerClass(this.getCombinerClass());
this.clientJob.setReducerClass(this.getReducerClass());
String inputs="";
for (int input=0; input<this.getInputPathsCount()−1; input++)
    inputs+=this.getInputPaths(input)+" , ";
inputs+=this.getInputPaths(this.getInputPathsCount()−1);
FileInputFormat.addInputPaths(this.clientJob, inputs);
FileOutputFormat.setOutputPath(this.clientJob, new Path(this.getOutputPath()));
this.setJobStatus("Submitted");
this.clientJob.submit();
System.out.println("Submitted "+this.getJobName()+" job to the YARN cluster");
}
```

The above snippet submits the Hadoop job to the Apache YARN cluster. This snippet acts as a generic main driver for the mapper and reducer classes, which creates a unique MapReduce client job object that sets the mapper, reducer classes, input and output paths and also submits the job on Apache YARN cluster.

3. isJobComplete method of Hadoop MapReduce job in Code 3.5.

Listing 3.5: isJobComplete method of Hadoop MapReduce job
boolean isJobComplete() throws Exception
{
    if (this.jobType.equals("HadoopMR") && !this.
        getJobStatus()
        .equals("Initialize") && !this.clientJob.
        isSuccessful())
    {
        return false;
    }
}

The above snippet periodically checks the status of the job. If the job is successfully completed, it will send notification to the children jobs signalling them to start.

4. afterJobCompletion method of Hadoop MapReduce job in Code 3.6. This is an optional method.

Listing 3.6: afterJobCompletion method of Hadoop MapReduce job

void afterJob() throws Exception
{
    if (this.afterJobCompletion)
    {
        if (this.jobType.equals("HadoopMR"))
        {
            String filePathComponent[] = this.
                getOutputPath().split("/");
            String localFilePath = this.HADOOPHOME + "/" + filePathComponent[filePathComponent
                .length - 1];
            FileSystem hdfs = FileSystem.get(this.
                url, new Configuration());
            Path hdfsFilePath = new Path(this.
                getOutputPath());
In this case, after the job completes, the temporary data is copied to the working directory.

All these snippets should be implemented in a java programming language and added to the actual methods of MEWSE. Hence, the proposed solution is a generic workflow submitter, which handles different types frameworks of jobs. It is designed to be pluggable to support future frameworks.

Appendix E contains the injection code for Spark.
Chapter 4

Test Case Workflow

We tested the MEWSE framework with an example workflow which uses multiple engines to process the sample dataset that run on Apache YARN. The sample workflow applies clustering and classification algorithms on a sample dataset to gain insights from it. The following chapter is organized as follows, Section 4.2 explains the Experimental setup of the proposed framework and Sections 4.1, 4.3 and 4.4 describe the sample workflow, execution of the sample workflow and summarize the evaluation of MEWSE respectively.

4.1 Example Workflow Representation

The proposed framework was tested using a sample dataset provided by Ontario South East Local Health Integration Network (SELHIN). SELHIN is an organisation, which collects the patient details from doctors and hospitals for medical analytics purpose. This dataset is supplied to us under a non disclosure agreement. This workflow is designed to test the proposed framework and hence, the focus is to create different types of jobs rather than the accuracy of the results. This workflow processed the SELHIN dataset to analyse the patient admission reasons and predict whether the
4.1. EXAMPLE WORKFLOW REPRESENTATION

patient is likely to be admitted in the future. This analysis is done by using the workflow represented in the Figure 4.1

![Figure 4.1: Example Workflow in Graphical Representation](image)

The SELHIN dataset consists of 20 attributes and 200,000 records. Attributes that include patient details, region, diagnosis type, admission status and other medical related fields. The workflow initially cleans up the dataset by removing NULL and unique values. It also converts chararray values into numerical values to apply machine learning algorithms. The Workflow applies a clustering algorithm on diagnosis type attribute, a classification algorithm on the admission status to create a model with the use of Apache Mahout on Apache YARN and also joins the demographics
records with the patient records with the help of a MapReduce job on Apache YARN. The following table 4.1 explains the operations involved in the jobs in the workflow.

Table 4.1: Operations involved in jobs in the example workflow

<table>
<thead>
<tr>
<th>Job Name</th>
<th>Description</th>
<th>Predecessors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patient records</td>
<td>Cleans the dataset by removing null and unique value records</td>
<td>Root job</td>
</tr>
<tr>
<td>Remove label</td>
<td>Removes the label field from the dataset</td>
<td>Patient records</td>
</tr>
<tr>
<td>Split data</td>
<td>Splits the dataset into training and testing dataset</td>
<td>Patient records</td>
</tr>
<tr>
<td>Join demographics with patient records</td>
<td>Joins the two different dataset with a simple MapReduce program</td>
<td>Patient records</td>
</tr>
<tr>
<td>Mahout cluster input conversion</td>
<td>Converts given dataset to machine understandable format</td>
<td>Remove label</td>
</tr>
<tr>
<td>Mahout KMeans cluster</td>
<td>Applies KMeans algorithm using Apache Mahout on the dataset</td>
<td>Mahout cluster input conversion</td>
</tr>
<tr>
<td>Mahout sequence dumper</td>
<td>Converts machine format of the dataset into user understandable format</td>
<td>Mahout KMeans cluster</td>
</tr>
<tr>
<td>Visualize the cluster output</td>
<td>Plots the graph on the target label field</td>
<td>Mahout sequence dumper</td>
</tr>
<tr>
<td>Training dataset</td>
<td>Converts the dataset into machine understandable format</td>
<td>Split data</td>
</tr>
<tr>
<td>Random forest describe</td>
<td>Describes the attributes of the dataset</td>
<td>Training dataset</td>
</tr>
<tr>
<td>Random build forest model</td>
<td>Applies the random build forest algorithm on the described dataset and creates the model</td>
<td>Random forest describe</td>
</tr>
<tr>
<td>Test dataset</td>
<td>Converts the dataset into machine understandable format</td>
<td>Split data</td>
</tr>
<tr>
<td>Test random forest build model</td>
<td>Tests the built model and emits the confusion matrix of the model as an output</td>
<td>Random build forest model and Test dataset</td>
</tr>
</tbody>
</table>

The full specification of the sample workflow is defined in Appendix D.

The workflow D.1 uses engines of several different types including Apache Hadoop MapReduce Job, Apache Mahout job and also simple jobs like Java and shell. The workflow processing begins with the input dataset, which is preprocessed in order to apply clustering and classification algorithms. For the clustering algorithm, it
removes the label from the dataset, then converts the dataset into Apache Mahout readable format and applies the Kmeans algorithm on the dataset. The cluster output can be visualized by joining the clustered output with the demographics records with the help of Hadoop MapReduce job. For the classification algorithm, the workflow splits the dataset into training and testing datasets. The Random forest algorithm is then applied to the training dataset to build the model. Once the model is built, the workflow applies the model to the testing dataset to get the accuracy of the prediction technique from the confusion matrix.

4.2 Experimental Setup

The MEWSE framework is deployed on Amazon EC2 (Elastic Cloud Computing) service on 10 virtual machines. The configuration of each machine is described in Table 4.2.

Table 4.2: Configuration of Amazon EC2 Machines

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance type</td>
<td>m3.medium</td>
</tr>
<tr>
<td>Zone</td>
<td>US East 1a</td>
</tr>
<tr>
<td>Processor</td>
<td>64 bit Intel Xeon E5-2670</td>
</tr>
<tr>
<td>Virtual CPU</td>
<td>1</td>
</tr>
<tr>
<td>Elastic Computing Unit (ECU)</td>
<td>3</td>
</tr>
<tr>
<td>RAM</td>
<td>3.75 GB</td>
</tr>
<tr>
<td>Solid State Device (SSD)</td>
<td>$1 \times 4$</td>
</tr>
<tr>
<td>Network performance type</td>
<td>Moderate</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 12.04, 64 bit OS</td>
</tr>
</tbody>
</table>
4.2. EXPERIMENTAL SETUP

Apache Yarn, Apache Mahout, JDK 1.7 and shell engines are deployed in 10 machines. Therefore, the cluster contains 1 master node and 9 worker nodes.

When configuring a YARN cluster, the values of the following parameters are provided, RAM (Amount of memory), CORES (Number of CPU cores) and DISKS (Number of DISKS). When the available memory is calculated, the reserved memory should be taken into account. For a disk size of 3.75 GB, the suggested reserved memory is 1 GB. In our case, the total memory is 3.75 GB and hence the available memory is 2.75 GB. The recommended formula for the number of containers per node is,

Number of Containers = min \( (2 \times CORES, 1.8 \times DISKS, (\text{Total available RAM}) / \text{MIN\_CONTAINER\_SIZE}) \)

where MIN\_CONTAINER\_SIZE is the minimum container size (in RAM). In the case of medium m3 instances, there is one core and disk, the available RAM is 2.75 GB and the recommended minimum container size is 256 MB. Hence the number of containers in our case is, number of containers = min \( (2 \times 1, 1.8 \times 1, (2.75 \text{ MB} / 256 \text{ MB})) \), which gives us two containers. The amount of RAM per container is

\[
\text{RAM\_PER\_CONTAINER} = \max(\text{MIN\_CONTAINER\_SIZE, (Total Available RAM) / containers})
\]

Hence, \( \text{RAM\_Per\_Container} = \max(256 \text{ MB, (2.75GB/2)}) \), which is 1.375 GB for our YARN configuration. We therefore use two containers, each having 1.375 GB. The rest of the parameters are calculated based on the above formulas as suggested by HortonWorks [29].

The screen shot in Figure 4.2 illustrates the configuration of Hadoop Distributed File System (HDFS).
4.3. SUBMIT AND MONITOR THE WORKFLOW

All the engines required to process the data are deployed on all the worker nodes and MEWSE is deployed on the master node, since it is a submitter framework on Apache YARN.

4.3 Submit and monitor the Workflow

Figure 4.3 illustrates the screenshot of executing the workflow example which starts with the input datasets NACRS_final_version1_patient.txt and NACRS_final_version2_demo.txt respectively. The dataset was loaded on the Hadoop Distributed File System.

Once the Workflow is submitted on the MEWSE framework, MEWSE submits all the jobs in the workflow on the Apache YARN cluster. The workflow’s submit and monitor processes are shown in Figure 4.4.

Since the workflow is designed to test MEWSE, which submits the workflow with different varieties of jobs on Apache YARN, we did not consider the accuracy of the workflow’s result. From Figure 4.4, it is clear that the proposed system is capable of executing workflows involve jobs using different analytic engines.

4.4 Discussion

The correctness of the workflow which has been submitted on MEWSE framework can be verified by using logs by Apache YARN. Since MEWSE framework runs on Apache YARN, the status of each individual analytic engines will be maintained by the Apache YARN log system. MEWSE is a submitter framework that submits all jobs in the workflow which use multiple analytic engines. So any change in the dataset or the configuration of analytic engines on Apache YARN will not affect
the performance of MEWSE. Since MEWSE is highly dependant on the execution layer (Apache YARN), the performance of MEWSE is directly proportional to the performance of analytic engines on Apache YARN. MEWSE may fail to perform when there is a logical error in designing a workflow job or when unstable engines are used on Apache YARN. MEWSE is responsible for submitting and monitoring jobs in the workflow but it is not responsible for delivering the results to end users. The execution layer (Apache YARN) stores the result in the Hadoop Distributed File System (HDFS) and the end users can fetch the result from the HDFS. The reason MEWSE is not compared to any other existing systems is that the existing workflow systems do not support multiple analytic engines. Some systems may support few engines but an entire workflow cannot be executed on the system.
Figure 4.2: The configurations of Hadoop Distributed File system
4.4. DISCUSSION

Figure 4.3: Before executing the sample workflow
Figure 4.4: Execution of the sample workflow
Chapter 5

Conclusions and Future Work

As the size of the data grows and analytic processes on that data become more complex analysis, users are faced with an increasing variety of available analytic engines. Hence, the formulation of workflows and the workflows themselves are becoming more complex. The workflows make use of different engines to analyse huge amount of data. MEWSE is designed to coordinate the different types of jobs in the workflow on the Hadoop Distributed File System with the help of Apache YARN.

The framework makes use of the benefits of Apache YARN, which runs non-MapReduce applications on the Hadoop Distributed File System and coordinates different types of jobs in the workflow with minimal human intervention. Workflow systems like Apache Oozie [15] are designed to run Hadoop jobs and the representation of the workflow is designed to be specific to the Hadoop cluster. MEWSE is more generic and is not restricted to only Hadoop’s MapReduce jobs. Major contributions of this thesis are, to provide a workflow submission framework on Apache YARN and to design a framework so that it should be adaptable to allow the inclusion of multiple analytics engines.

MEWSE has been developed using the Java programming language. The states
of the jobs are defined by the Java programming language. Thus, the representation of the workflow is customizable by developers who can add new frameworks to MEWSE. Hence, developers can create an user interface, which collects the value of the parameters of the job from the users and generates XML code for the given values. MEWSE was successfully tested on Amazon Elastic Cloud Computing service with the sample workflow which is explained in the Section 4.1 and the datasets provided by the SELHIN community.

MEWSE is missing some workflow processing concepts. Control flow features like time frequency (deadline or repetitive task), choice (based on the decisions) and sub workflows concepts have been left for future work. Since MESOS [13] and Apache YARN share the same properties, I believe MEWSE is expected to run on MESOS with minimal modifications.

MEWSE acts as an interface between the workflow and the distributed processing framework. Since it is acting as a submitter on the distributed processing framework, it does not take control over resource allocation. Instead, the resource allocation is done by the distributed processing system. In future, the resource allocation mechanism of the workflow can be integrated into MEWSE analyzing the overall needs of the workflow. Another possible optimization is to reuse the containers which are used by the parent jobs. Since a workflow is designed in a DAG structure, it is necessary to transfer the data from the parent jobs to the children jobs. By reusing the containers, which are used by the parent jobs, data locality of the current job would be enhanced. Also the overall processing performance of the application would be improved.
References


REFERENCES


Appendix A

Apache Hadoop

This appendix describes a working example of a MapReduce job in Apache Hadoop. The process is illustrated in the Figure A.1. At the highest level, there are four entities, including

- The Client, which submits the MapReduce job.
- The JobTracker, which coordinates the job run.
- The TaskTrackers, which run the individual tasks of the job.
- The distributed filesystem (HDFS), which is used for sharing job files between the other entities.
When the job is submitted to the JobTracker by the JobClient, the JobTracker responds to the JobClient with the JobID. The JobClient is responsible for copying the job resources to the HDFS. After the completion of these steps, the JobTracker initializes the job and computes the number of splits (chunks of data) to be processed. The TaskTracker sends the heartbeat signals to the JobTracker and in return it gets the task to be processed. Finally, tasks (Map and Reduce) are processed by the JVM.
Appendix B

Apache YARN

Appendix B describes the walk-through of Apache YARN with a simple example. Figure B.1 illustrates the working example of multiple engines on Apache YARN.

Figure B.1: Apache YARN walk-through with an example

In this example, two clients submit a MapReduce Job and a HBase job on Apache
YARN, respectively. Apache YARN has 1 master node which runs Resource Manager service and 8 worker nodes, which run Node Manager service. When a client submits an application, say MapReduce job (MR $AM_1$), to the Resource Manager, the Resource Manager allocates a container for the Application Master in one of the Node Managers. As soon as the Application master (per job master service, which coordinates tasks execution) is launched, it sends resource requests to the Resource Manager. Based on the resource availability and other conditions (data locality, number of containers, so on), the Resource Manager will respond to the Application Master requests. Once the Application Master receives enough containers to start the task, the Application Master will launch all the containers with the help of the Node Managers. Once all the tasks have been executed successfully, all the containers which are used to run tasks will be terminated with the help of the Node Managers. The Node Managers are responsible for notifying the resource status to the Resource Manager.

Since Apache YARN is designed to support non-MapReduce applications on the Hadoop Distributed File System, the clients can submit multiple types of jobs (say, HBASE On YARN (HOYA)) along with MapReduce applications.
Appendix C

Representation of Jobs

Appendix C gives the representations for example jobs using different engines. In this Appendix, we have included the representation of Mahout clustering and classification jobs which is used in the example 4.1 and a simple Java job. Code C.1 defines the representation of a KMeans clustering job for Apache Mahout.

Listing C.1: Representation of Mahout KMeans Clustering Job

```xml
<job type="MahoutKMeansCluster">
  <jobName>MahoutKMeansCluster</jobName>
  <input>/user/kiran/MahoutKmeansSynthetic/convertedInput</input>
  <output>/user/kiran/MahoutKmeansSynthetic/input−kmeans</output>
  <distanceMeasure>org.apache.mahout.common.distance.CosineDistanceMeasure</distanceMeasure>
  <kPoints>3</kPoints>
  <iterations>10</iterations>
  <clusterPath>/user/kiran/MahoutKmeansSyntehtic/input−kmeans−cluster</clusterPath>
  <predecessor/></predecessor>
</job>
```
The job is defined using parameters such as job name, input and output paths, distance measure class path (Euclidean, Cosine distance, etc.), KPoints value to the define number of clusters, iterations, clusterPath to store the processed clustered points.

Code C.2 represents the Mahout Build Forest job as defined by parameters such as job name, a CSV/arff file indicating the training dataset, the output (a trained model), describePath defines the attributes of the dataset, selectedAttributes and numTrees value indicates the number of trees to be generated.

Listing C.2: Representation of Mahout Build Forest Job

```xml
<job type="MahoutBuildForest">
  <jobName>MahoutBuildForest</jobName>
  <input>/user/kiran/Mahout/KDDTrain+.arff</input>
  <output>/user/kiran/Mahout/nsl-forest</output>
  <describePath>/user/kiran/Mahout/KDDTrain+.info</describePath>
  <numTrees>100</numTrees>
  <selectedAttribute>5</selectedAttribute>
  <predecessor></predecessor>
</job>
```

Code C.3 represents the Mahout Test Forest job as indicated by the jobName parameter. It takes as input a test dataset and classifies the test dataset with the use of trained model. The parameter describePath defines the dataset to be used for processing.

Listing C.3: Representation of Mahout Test Forest Job

```xml
<job type="MahoutTestForest">
  <jobName>MahoutTestForest</jobName>
  <input>/user/kiran/Mahout/KDDTest+.arff</input>
  <output>/user/kiran/Mahout/predictions</output>
</job>
```
<describePath>/user/kiran/Mahout/KDDTrain+.info</describePath>
<modelPath>/user/kiran/Mahout/nsl-forest</modelPath>
<predecessor></predecessor>
</job>

Code C.4 represents the simple Java job. Parameter job name represents the name of the job, java class path, which consists a main method as a input parameter and arguments of the main method can be provided in the arguments parameter.

Listing C.4: Representation of Java job

<job type="JavaClass">
  <jobName>RemoveLabel</jobName>
  <javaClassName>test.RemoveLabel</javaClassName>
  <arguments>
    <argument><value>hdfs://localhost:9000/user/kiran/Testing/NACRS_final_short4.txt</value></argument>
    <argument><value>hdfs://localhost:9000/user/kiran/Testing/ClusterInput/input.txt</value></argument>
  </arguments>
  <predecessor></predecessor>
</job>
Appendix D

Test Case Workflow Representation

The representation of the test case workflow in XML format D.1 is shown in this Appendix.

Listing D.1: Representation of the test case workflow

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<workflow>
  <workflowName>SELHINSampleWorkflow</workflowName>

  <job type="JavaClass">
    <jobName>RemoveLabel</jobName>
    <javaClassName>test.RemoveLabel</javaClassName>
    <arguments>
      <argument>
        <value>NACRS_final.txt</value>
      </argument>
      <argument>
        <value>input.txt</value>
      </argument>
    </arguments>
    <predecessor></predecessor>
  </job>

  <job type="JavaClass">
    <jobName>SplitData</jobName>
    <javaClassName>test.DataSplit</javaClassName>
    <arguments>
      <argument>
        <value>NACRS_final.txt</value>
      </argument>
    </arguments>
  </job>
</workflow>
```
<job type="HadoopMR">
<jobName>JoinDemographicsWithPatient</jobName>
<jobClass>test.join</jobClass>
<mapperClass>test.join$joinMapper</mapperClass>
<reducerClass>test.join$joinReducer</reducerClass>
<outputKeyClass>org.apache.hadoop.io.Text</outputKeyClass>
<outputValueClass>org.apache.hadoop.io.Text</outputValueClass>
<inputPaths>
<input>
<path>NACRS_final.txt</path>
</input>
<input>
<path>NACRS_final_dem.txt</path>
</input>
</inputPaths>
<outputPath>output</outputPath>
<predecessor></predecessor>
</job>
<job type="MahoutClusterInputConversion">
<jobName>MahoutClusterInputConversion</jobName>
<input>input.txt</input>
<output>convertedInput</output>
<predecessor>RemoveLabel</predecessor>
</job>
<job type="MahoutKMeansCluster">
<jobName>MahoutKMeansCluster</jobName>
<input>convertedInput</input>
<output>input−kmeans</output>
<distanceMeasure>org.apache.mahout.common.
  distance.CosineDistanceMeasure</distanceMeasure>
<kPoints>3</kPoints>
<iterations>100</iterations>
<clusterPath>input−kmeans−cluster</clusterPath>
<predecessor>MahoutClusterInputConversion</predecessor>
</job>

<job type="MahoutSeqDumper">
  <jobName>MahoutSequenceDumper</jobName>
  <input>input−kmeans/clusteredPoints/part−m−00000</input>
  <output>input−kmeans/cluster−points.txt</output>
  <predecessor>MahoutKMeansCluster</predecessor>
</job>

<job type="MahoutRandomForestDescribe">
  <jobName>MahoutRandomForestDescribe</jobName>
  <input>NACRS_train.txt</input>
  <output>NACRS_train.info</output>
  <describe>18 N L N</describe>
  <predecessor>SplitData</predecessor>
</job>

<job type="MahoutBuildForest">
  <jobName>MahoutBuildForest</jobName>
  <input>NACRS_train.txt</input>
  <output>nsl−forest</output>
  <describePath>NACRS_train.info</describePath>
  <numTrees>100</numTrees>
  <selectedAttribute>5</selectedAttribute>
  <predecessor>MahoutRandomForestDescribe</predecessor>
</job>

<job type="MahoutTestForest">
  <jobName>MahoutTestForest</jobName>
  <input>NACRS_test.txt</input>
  <output>predictions</output>
  <describePath>NACRS_train.info</describePath>
</job>
<modelPath>nsl-forest1</modelPath>
<predecessor>MahoutBuildForest, SplitData</predecessor>
</job>
</workflow>
Appendix E

Injecting New Frameworks

In this Appendix, the injection of a Spark job into the proposed framework is discussed. In order to inject the new framework into the proposed system, the developer must define all the following functionalities:

- Define the parameters of the job (XML representation),
- Define the parse method,
- Define the submit method,
- Define the isJobComplete method and
- Define the afterJobCompletion method (optional).

These tasks are illustrated in the following code snippets. Code E.1 shows the
representation of Spark job. A Spark job requires the following parameters: job name, java class name which contains the main method and the jar file path which contains the java class.

Listing E.1: Representation of Spark job in XML

```xml
<job type="sparkJar">
  <jobName>XXXXYYYZZZ</jobName>
  <sparkJar>AAAAABBBB</sparkJar>
  <javaClassName>QQQQQEEEE</javaClassName>
  <arguments>
    <argument><value>hdfs://localhost:9000/xyz</value></argument>
  </arguments>
  <predecessor></predecessor>
</job>
```

Code E.2 defines the parse method of a Spark job. It parses all the parameters from the Spark job representation file (XML format) and stores the value of the parameters in the job object.

Listing E.2: Parse method of Spark job

```java
void parseXML(Node jobDetail) throws ClassNotFoundException, DOMException {
  if (jobDetail.getNodeType() == Node.ELEMENT_NODE && this.jobType.equals("sparkJar")) {
    Element jobElement=(Element) jobDetail;
    this.jobName=new String(jobElement.getElementsByTagName("jobName").item(0).getTextContent());
    String predecessorSet = new String(jobElement.getElementsByTagName("predecessor").item(0).getTextContent());
    predecessorSet=predecessorSet.trim();
  }
}
```
if (predecessorSet != null && !predecessorSet.isEmpty())
{
    String predecessors[] = predecessorSet.split(",");
    for (int predecessor = 0; predecessor < predecessors.length;
         predecessor++)
    this.setPredecessor(predecessors[predecessor]);
}
this.javaClassName = new String(jobElement.getElementsByTagName("javaClassName").item(0).
    getTextContent());
this.sparkJarName = new String(jobElement.getElementsByTagName("sparkJar").item(0).
    getTextContent());
NodeList argumentsList = jobElement.getElementsByTagName("arguments");
Node argumentsListDetail = argumentsList.item(0);
Element argumentsListElement = (Element) argumentsListDetail;
NodeList argumentList = argumentsListElement.getElementsByTagName("argument");
int count = 0;
arguments = new String[argumentList.getLength()];
for (int argument = 0; argument < argumentList.getLength();
    argument++)
{
    Node argumentvalueDetail = argumentList.item(argument);
    Element argumentvalueElement = (Element) argumentvalueDetail;
    arguments[count++] = new String(argumentvalueElement.
        getElementsByTagName("value").item(0).
        getTextContent());
}
}

Code E.3 explains the submission method of a Spark job. It submits the spark job on a YARN cluster with the use of parsed parameters.
Listing E.3: Submit method of Spark job

```java
void submitJob() throws IOException, ClassNotFoundException,
        InterruptedException {
    if (this.jobType.equals("sparkJar")) {
        this.setJobStatus("Submitted");
        this.process = Runtime.getRuntime().exec(this.SPARK_HOME
            \+"/bin/spark-submit-master-yarn-client-class"
            \+this.javaClassName+" \+this.sparkJarName");
        System.out.println("Submitted "+this.getJobName()+" job 
            to the YARN cluster");
    }
}
```

Code E.4 defines the isJobComplete method of spark. It is used to check the complete status of the spark job periodically.

Listing E.4: isJobComplete method of Spark Job

```java
boolean isJobComplete() throws Exception {
    if (this.jobType.equals("sparkJar") && !this.getJobStatus()
        .equals("Initialize") && this.isRunning(this.process)) {
        return false;
    }
}
```

Once the above snippets are injected into the proposed framework, Spark jobs can be included in the representation of workflows along with other engines of jobs. Now, the proposed system executes the workflows, which have Spark jobs.