Policy-Driven YARN Launcher

VASILEIOS GIANNOKOSTAS
Policy-Driven YARN Launcher

Vasileios Giannokostas

Master of Science Thesis

Software Engineering of Distributed Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

September 2016

Examiner: Associate Prof. Jim Dowling
Supervisor: Professor Seif Haridi

TRITA-ICT-EX-2016:101
Abstract

In recent years, there has been a rising demand for IT solutions that are capable to handle vast amount of data. Hadoop became the de facto software framework for distributed storage and distributed processing of huge datasets with a high pace. YARN is the resource management layer for Hadoop ecosystem which decouples the programming model from the resource management mechanism. Although Hadoop and YARN create a powerful ecosystem which provides scalability and flexibility, launching applications with YARN currently requires intimate knowledge of YARN’s inner workings. This thesis focuses on designing and developing support for a human-friendly YARN application launching environment where the system takes responsibility for allocating resources to applications. This novel idea will simplify the launching process of an application and it will give the opportunity to inexperienced users to run applications over Hadoop.
Sammanfattning

De senaste åren har haft en ökad efterfrågan på IT-lösningar som är kapabla att hantera stora mängd data. Hadoop är ett av de mest använda ramverken för att lagra och behandla stora datamängder distribuerat och i ett högt tempo. YARN är ett resurshanteringslager för Hadoop som skiljer programmeringsmodellen från resurshanteringsmekanismen. Även fast Hadoop och YARN skapar ett kraftfullt system som ger flexibilitet och skalbarhet så krävs det avancerade kunskaper om YARN för att göra detta. Detta examensarbete fokuserar på design och utveckling av en människovänlig YARN applikationsstartmiljö där systemet tar ansvar för tilldelning av resurser till program. Denna nya idé förenklar starten av program och ger oerfarna användare möjligheten att köra program över Hadoop.
Acknowledgements

I would like to thank my supervisor Associate Professor Jim Dowling for giving me the opportunity to conduct my thesis at SICS and get involved with an interesting topic. Jim gave me invaluable insights throughout my thesis, he motivated me to work hard and he supported me when needed. Also, I would like to acknowledge the efforts and support of all participants in this projects.
To my parents for their sacrifices and unconditional love.
Contents

1 Introduction .................................................. 3
   1.1 Problem description ........................................ 3
   1.2 Purpose .................................................... 4
   1.3 Goals ....................................................... 4
   1.4 Risks, Consequences and Ethics ............................ 4
   1.5 Thesis outline ............................................. 5

2 Background .................................................. 7
   2.1 Big Data ................................................... 7
   2.2 Distributed System ........................................ 8
   2.3 Hadoop ..................................................... 9
   2.4 Distributed File Systems (DFS) ............................ 10
      2.4.1 Google Distributed File System (GFS) ............... 10
      2.4.1.1 GFS Architecture .................................. 10
      2.4.2 Hadoop Distributed File System (HDFS) ............. 11
      2.4.2.1 HDFS Architecture .................................. 11
   2.5 Resource Management and Data Processing ............... 12
      2.5.1 MapReduce ............................................. 12
      2.5.2 HDFS 2 ............................................... 15
      2.5.3 YARN - Yet Another Resource Negotiator .......... 15
         2.5.3.1 YARN Architecture and Components .......... 16
         2.5.3.2 Resource Manager (RM) ......................... 16
         2.5.3.3 Application Master (AM) ....................... 17
         2.5.3.4 Node Manager (NM) ............................. 17
         2.5.3.5 Containers ....................................... 17
         2.5.3.6 YARN Application Startup .................... 18
      2.5.4 Apache Spark ........................................ 18
         2.5.4.1 Resilient Distributed Dataset (RDD) ........ 19
         2.5.4.2 Spark Applications over YARN ............... 19
         2.5.4.3 Dynamic Allocation on Spark .................. 20
         2.5.4.4 Monitoring Spark applications ............... 21
2.6 Dr. Elephant ................................................. 21
2.7 Hops ....................................................... 22

3 Research Method and Methodology ........................................... 25
3.1 Research Method ............................................... 25
3.2 Research Strategy ............................................... 25
3.3 Data Collection and Analysis ......................................... 26
3.4 Proof of Concept ............................................... 26

4 Implementation ......................................................... 27
4.1 Dr. Elephant Integration ........................................... 27
4.2 Job History ...................................................... 30
  4.2.1 Model ..................................................... 30
  4.2.2 Relational Model .......................................... 30
  4.2.3 Job History webpage ...................................... 32
4.3 Policy-Driven Execution ........................................... 33
  4.3.1 Auto-configuration panel ................................... 39

5 Evaluation .......................................................... 41
5.1 Use cases ....................................................... 41
  5.1.1 Pi Estimation ............................................... 41
  5.1.2 Word Count ............................................... 43

6 Conclusion and Future Work ................................................... 45
6.1 Conclusion ...................................................... 45
  6.1.1 Goals Achieved ............................................. 45
6.2 Future Work ..................................................... 46

Bibliography ........................................................... 47
List of Figures

2.2 Hadoop Distributed File System Architecture .................. 12
2.3 MapReduce example: Shape count problem .................... 13
2.4 MapReduce execution overview [2] ............................. 14
2.5 YARN Architecture [3] ........................................ 16
2.6 YARN Application start-up .................................... 18
2.7 Iterative operations on MapReduce ............................. 19
2.8 Interactive Operations on Spark ................................. 19
2.9 Apache Spark over YARN ..................................... 20
2.10 Heuristic example on Dr.Elephant ............................. 23

4.1 Flowchart - Job Runtime Heuristic ............................. 29
4.2 Job History .................................................. 31
4.3 Job History - Relational Model ............................... 31
4.4 Job History page ............................................. 34
4.5 Job History Details .......................................... 35
4.6 Policy-Driven mechanism architecture ........................ 37
4.7 UI - Auto-configuration panel ................................. 39

5.1 Pi estimation - execution ................................... 42
5.2 Word count example - execution ............................. 43
# List of Tables

2.1 Differences between Hadoop v1 and Hadoop v2 .......................... 15
4.1 Heuristic Severity for Dr. Elephant .......................................... 28
4.2 Job Runtime Heuristic - Thresholds and evaluation .................. 28
4.3 Job Degree of similarity ....................................................... 38
# List of Acronyms

This document requires readers to be familiar with a list of Acronyms that will be used throughout this report.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AM</td>
<td>Application Master</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>DS</td>
<td>Data Set</td>
</tr>
<tr>
<td>FS</td>
<td>File System</td>
</tr>
<tr>
<td>HOPS</td>
<td>Hadoop Open Platform-as-a-Service</td>
</tr>
<tr>
<td>GFS</td>
<td>Google’s File System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>NDB</td>
<td>Networked Database Service</td>
</tr>
<tr>
<td>NM</td>
<td>Node Manager</td>
</tr>
<tr>
<td>RDD</td>
<td>Resilient Distributed Dataset</td>
</tr>
<tr>
<td>REST</td>
<td>REpresentational State Transfer</td>
</tr>
<tr>
<td>RM</td>
<td>Resource Manager</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>YARN</td>
<td>Yet Another Resource Manager</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In recent years, the rapid development of science and technology as well as the vast expansion of the Internet have led to huge amount of data. In the science domain, such as DNA profiling, applications which store and identify human characteristics require a lot of storage and computational power [4]. Moreover, in the enterprise domain, like social networks and search engines, this need is driven by applications which store the user behavior, analyses their activities and gives valuable insights to the companies. Big Data[5] is a term which describes both structured and unstructured data that flood science and industry in usual daily tasks.

The importance of Big Data is not only related to the volume of the datasets but also improves the analytical and statistical models which organizations use to extract business value using data from various sources such as databases, social networks, blogs and sensors.

Hadoop[6] is the state-of-the-art ecosystem for storing and processing large datasets on a network of commodity hardware. Moreover, Hadoop uses its own distributed file system which is designed to store large blocks of data in a distributed manner called HDFS[6]. Nowadays, many companies seem to adopt the Hadoop ecosystem in order to deal with Big Data. However, using Hadoop for launching applications requires intimate knowledge by the users. This thesis focuses on providing a friendly environment to inexperienced users in order to run jobs on Hadoop ecosystem.

1.1 Problem description

YARN[7] is the resource management layer for Hadoop ecosystem which decouples the programming model from the resource management mechanism providing a solid platform for management across Hadoop clusters. Moreover, it extends the power of Hadoop to run new technologies and provides a scalable and multi-
tenant environment. An advantage of YARN is that it enhances the cluster utilization because of the dynamic allocation of cluster resources and increases the utilization in comparison with the early versions of Hadoop which used only static MapReduce methods [2]. However, launching applications with Yarn currently requires intimate knowledge of YARN’s inner workings. User should be capable to specify how much memory and CPUs to dedicate to the ApplicationMaster(AM) which is the heart of every application as well as how much memory, CPUs and workers to dedicate to the application itself. Users should adapt these resource requests to the application’s expected resource requirements, the required application completion time and the current load on the cluster.

1.2 Purpose

This project will design and develop support for a human-friendly YARN application launching environment, where the system takes responsibility for allocating resources to applications. More precisely, this thesis aims to enhance the performance of an existing Hadoop platform with is called Hops[8]. Hops is a Hadoop Open Platform-as-a-Service and is a new distribution of Hadoop. Improving the launching process and decoupling it from the need to know the internal functionality, we will give the opportunity to common users to use Hops and manage their datasets.

1.3 Goals

The solutions should be efficient in its use of resources, be fair to applications and take into consideration application Quality of Service(QoS) requirements(the application priority, latest possible completion time, available quota at the user/project).

1.4 Risks, Consequences and Ethics

The goal for the project is to come up with a Policy-Driven YARN Launcher. The major risk is how to create a functional design for this purpose. A nonfunctional implementation will cause serious problems such as unfairness distribution of resources among the users and it will degrade the performance of the existing system. In the worst scenario, an inappropriate solution will cause system malfunction. Moreover overprovision of resources will cause higher power consumption which is an inefficient way of handling available resources.
As the content of the thesis is about software engineering, the research should comply with the \textit{Software Engineering Code of Ethics} as it proposed by Don Gotterbarn et. al \cite{9}. A software engineer should act for the public interest. The topic of this thesis is about Hadoop ecosystem which is a very hot topic and attracts interest from software developers and companies. Moreover, the products that will be designed should meet the highest standard quality possible and maintain the existing public interest. Last but not least, working with other people in a company, a software engineer should act fairly to his/her colleagues and be supportive.

1.5 Thesis outline

Chapter 1 introduces the reader to the general field of the problem as well as the problem description and the goals of the thesis. Chapter 2 presents the necessary theoretical background that a reader should have in order to fully comprehend the problem being investigated. Chapter 3 describes the model and the important decisions taken for the implementation. The details of the implementation are showed in Chapter 4 where important components are presented thoroughly. Chapter 5 demonstrates the evaluation process followed and results of this phase. Conclusions and future work are explained in the 6\textsuperscript{th} Chapter.
Chapter 2

Background

This chapter encompasses all the necessary knowledge that a reader should have in order to fully comprehend the content and the objectives of the thesis. It starts with general concepts and ends up with particular methodologies that have been used throughout the thesis implementation.

2.1 Big Data

Big Data[5] is a term which describes blocks of data that are too big to be stored and processed using traditional methods. Moreover, it is related to different aspects of human activity and is widely used nowadays. A good example to understand the importance of Big Data is in e-Science. For instance, the Large Hadron Collider (LHC) accelerator at CERN produces 30 petabytes of data every year [10] while the physicists examine that data in order to determine any valuable information.

We can imagine that the analysis of the data could not be possible without a distributed way for storing and processing. In order to understand better the major principles of Big Data we will describe the 5 fundamental dimensions a.k.a. 5 V’s of Big Data, as described by Demchenko et al. [5].

- **Volume** This is the major feature of Big Data. It represents data in terms of size, scale, dimension and amount. It is a vital attribute if we take into account that recent surveys have shown that the data production will be 44 times greater in 2020 than it was in 2009 [11].

- **Velocity** Despite the big size, Big Data are generated with high rate and require to be processed almost in real-time. Organizations abandoned traditional ways of periodic batch processing and move towards real-time processing of data.
8 Chapter 2. Background

- **Variety** This term is related to the complexity of data and the semantic information behind them. Data could be structured or unstructured and will increase their complexity the next years [5].

- **Value** The value that is added to the system by the data is very important. Different kinds of value can be added and enhance the existing system, such as stochastic, probabilistic, regular or random value.

- **Veracity** Data veracity can be split in two major aspects. Data consistency which describes the uniformity of structured and unstructured data and to data trustworthiness which encompasses data origin, the processing methods and a trustworthy infrastructure where data are transmitted.

The description of these five dimensions of Big Data give to the reader insights about the importance and the requirements of Big Data to supporting Big Data infrastructures.

2.2 Distributed System

Most of today’s IT infrastructures are Distributed Systems (DS) like telecommunication systems, the Internet, cloud computing systems and IoT. At the core of these systems you find services which run on multiple computers. Hence, DS is a collection of computers which operate together in order to achieve a common goal. They communicate by sending and receiving messages.

A distributed operating system gives a higher performance than a single machine, more resources can be added easily by importing new nodes to the cluster and it is fault tolerant, which means that if a node crashes the other nodes “replace” the failed node and operate instead of it. Although DSs have these advantages, we should care about some issues that cause problems. A big problem is about security due to communicating, where some messages can be lost in transit and the overloading should be prevented in order to keep the cluster active and functional.

This thesis focuses on commodity computing which refers to an organization’s use of low-cost hardware components in order to get more computing power. Companies use a number of lower-cost computers (conventional PCs) instead of using supercomputers that are expensive and difficult to maintain. This way, companies can acquire more computational power by spending less money.
2.3 Hadoop

Hadoop[6] was created by Doug Cutting and it was the main component of Apache Nutch[12], a web search engine which started in 2002. In reality, the Apache Nutch engine operated as a web crawler browsing thousands of websites and accumulating huge amounts of data. However, the designers realized that their architecture was not powerful preventing the system to scale up and crawl millions of webpages.

The inspiration to improve their architecture came from a paper published by Google in 2003 that described the architecture of Google’s Distributed File System (GFS)[1], we will discuss about it later on (see Section 2.4.1) . One year later, in 2004, Google announced another pioneer technology which is called MapReduce[2] (we discuss MapReduce further at Section 2.5.1) . Both GFS and MapReduce became the basic ingredients of Hadoop ecosystem which adopted by Nutch but became very famous when Yahoo! announced that the Yahoo! Search Webmap, which is a Hadoop application, uses a 10000-core Linux cluster that is able to index 1 trillion links and produces 300 TB output [13].

The main characteristic of Hadoop is that it is an open-source software framework for storing data and launching applications on clusters with commodity hardware. Hadoop satisfies every dimension of Big Data which described above. More precisely, Hadoop framework breaks big data into blocks, stores them on a cluster and processes huge amounts of data by using low-cost computers for quick results.

The benefits of Hadoop are:

- **Processing power**: Hadoop’s distributed model processes big data quickly. The power of your cluster depends on the number of nodes you have. The more nodes you have, the more computing power you have.

- **Scalability**: It is simple to extend Hadoop’s processing power by adding more nodes easily.

- **Flexibility**: Hadoop provides a flexible way to store and retrieve data. The user can store unstructured data like files, images and videos without refining that data. This does not happen using traditional ways, like using a relational database where you have to store every information explicitly.

- **Fault Tolerance**: A key advantage of using Hadoop is that it is resilient to failures. Hadoop replicates the data to several nodes. So, even if a node crashes there is at least another copy available on the cluster.
• **Cost effective:** The framework is open source and uses commodity hardware to store the data. Also, traditional relational databases are extremely expensive to scale up because they are designed to run on a single server in order to preserve the integrity of its contents and avoid problems of distributed computing [14].

### 2.4 Distributed File Systems (DFS)

A Distributed File System (DFS) is a common model of a file system distributed across multiple machines. DFS is implemented using distributed computing as described above and provides location transparency for the files as well as storage scalability for the entire system. A DFS that paved the way for the development of Hadoop Distributed File System (HDFS), and consequently Hadoop, is the Google File System (GFS) that released in 2003 [1].

#### 2.4.1 Google Distributed File System (GFS)

Google File System (GFS)[1] is a scalable DFS for big distributed data-intensive applications that was developed to meet the requirements of Google’s data processing needs in 2003. The design of GFS was based on existing DFS and it was driven by the current and predicted application work-loads of Google. GFS extends the traditional DFS and imports radical key points in the world of Distributed File Systems. More precisely, GFS is more fault tolerant as it consist of thousands of storage machines built from low-cost commodity hardware. Moreover, it can support multi-GB files and concurrently appends to the same file. In addition, GFS offers high throughput and low latency making the applications faster.

##### 2.4.1.1 GFS Architecture

GFS consists of three main entities, the *master*, the *chunkservers* and the *clients*. Chunkservers store files and replications of the files, while the master acts like the arbitrator of the system as it has a global knowledge of the system, monitors the files and assigns chunks to the chunkservers. Files are divided in 64MB blocks (chunks), chunks are stored among the chunkservers and are replicated three times. A client can store or retrieve a file from the file system following two steps. The first step is to contact the master which knows which chunkservers keep a file or which are available to store a file. Then the client communicates with the chunkservers directly without the mediator master to transfer the data. Figure 2.1 depicts the GFS architecture with its main entities and operations.
2.4. Distributed File Systems (DFS)

2.4.2 Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS) is an open source DFS written in Java and it is based on GFS. HDFS is the file system component of Hadoop and stores file system data and application data separately. Initially, HDFS was designed to support a reliable and a fast distributed file system for the analysis of very large data sets using the MapReduce [2] paradigm.

2.4.2.1 HDFS Architecture

The basic entities of the HDFS are the DataNodes, NameNodes, and the HDFS Client. Secondary components are the Client, Image and Journal, CheckpointNode and BackupNode. Figure 2.2 illustrates the architecture of HDFS and some of its basic operations. A brief explanation of the basic entities, NameNode and DataNode, are given below for a better understanding of the HDFS Architecture:

**NameNode:** NameNode is the master of HDFS as it manages all the blocks that are stored on the DataNodes. HDFS cluster consists of a single NameNode which is a high-availability server that manages the file system namespace and controls the permissions to files by the clients.

**DataNode:** HDFS cluster consists of a number of DataNodes. DataNodes are the slaves nodes in HDFS and are the physical nodes where the blocks are stored. As we mentioned before, a Client communicates with the NameNode to find the location of a file and afterwards communicates directly with the DataNode to store or retrieve data. A NameNode should be aware of the existing blocks in every DataNode and the state of every DataNode (active, failed). For this reason every DataNode sends heartbeats to the NameNode as well as a list of the block that it
stores.

Figure 2.2: Hadoop Distributed File System Architecture

2.5 Resource Management and Data Processing

Instead of a Distributed File System another need arose. As we know, the rapid expand of Internet in the last decades created large amounts of data. Moreover, the advent of multimedia in the web-page contents instead of just plain-text changed the form of the Web. Web crawling became a task where a lot of storage and computational power was required. The need to deal with this problem led Google to develop a programming model for processing and generating large datasets. This model named MapReduce [2] and it initially developed in 2004. So, instead of a distributed file system we should have a distributed data-processing framework that process the data in a distributed manner.

2.5.1 MapReduce

As the name indicates, MapReduce consists of two functions, the Map function and the Reduce function and it based on many functional languages and especially Lisp. The map function takes a set of key/value pairs and generates the
intermediate key/value pairs. Then, the reduce function merges all the intermediate results associated with the same key. The benefits of MapReduce it that parallelizes the map and reduce functions and executes these tasks in parallel using a large cluster of commodity hardware, so MapReduce jobs have a high performance running on large clusters. Figure 2.3 depicts a simple MapReduce shape counter example. In this case, the input is sets of shapes, the map function emits the input producing (key,value) pairs (the intermediate pairs). Keys are the shapes and value is the amount of shapes. The next step is the reduce function where values of the same key are aggregated. The final results consists of the keys (shapes) and the total number of the appearances of this particular shape.

MapReduce Architecture follows the pattern of a single master and many workers. The master works as the arbitrator which coordinates task executions and resource management and the workers which run the tasks. Figure 2.4 illustrates the execution of MapReduce jobs.

Since many node failures occur during an execution, the system should be able to handle these failures properly. The master is a single point of failure and for that reason the system keeps snapshots with the master’s state periodically.
In case of a master’s failure the system restores the state of the master using the latest snapshot. Master is aware of the state of the cluster (healthy or dead workers) because it receives heartbeats from every worker. If a worker crashes, the master assigns the task (map or reduce) to another alive worker in the cluster and informs other workers (that depend on the dead worker) to redirect their internal requests to the newly assigned worker. This method allows the system to be resilient to massive failures of workers.

![MapReduce execution overview](image)

Figure 2.4: MapReduce execution overview [2]

Hadoop v1 supported only MapReduce jobs. In this case MapReduce was working as a cluster resource management and data processing tool. MapReduce was intensive in I/O and was constrained in interactive analysis and graphics support. This paved the way for further development of Hadoop v1 to Hadoop v2. We will describe the major improvements of Hadoop 2 as well as we will present basic differences of these two versions. Table 2.1 presents the major changes.

The two biggest contributions of Hadoop v2 in Hadoop ecosystem were changes in HDFS (HDFS 2) and the second was the improvements on the cluster resource management (YARN). We will briefly explain the basic characteristics of HDFS 2 and YARN resource scheduler.
2.5. Resource Management and Data Processing

<table>
<thead>
<tr>
<th>No.</th>
<th>Hadoop v1</th>
<th>Hadoop v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Poor scalability. Limited to 4000 nodes per cluster.</td>
<td>Improved scalability. Supported up to 10000 nodes per cluster.</td>
</tr>
<tr>
<td>2</td>
<td>Only has one namespace for managing HDFS.</td>
<td>Supports multiple namespaces for managing HDFS.</td>
</tr>
<tr>
<td>3</td>
<td>A single NameNode is a single point of failure. Hadoop v1 requires only manual intervention for recovery.</td>
<td>Hadoop v2 supports automatic recovery upon NameNode failure.</td>
</tr>
<tr>
<td>4</td>
<td>Supports MapReduce processing models only.</td>
<td>Allows to run MapReduce jobs as well as other models like Giraph, Spark, Flink, HBase, Storm, OpenMPI.</td>
</tr>
<tr>
<td>5</td>
<td>MapReduce operates both like a processing data and resource management tool.</td>
<td>Processing data is separated with resource management. YARN does cluster resource management. Processing data is done using different processing tools.</td>
</tr>
</tbody>
</table>

Table 2.1: Differences between Hadoop v1 and Hadoop v2

2.5.2 HDFS 2

As we know HDFS consists of two main components. The first is the namespace service which manages operations on files and directories such as creating, deleting and editing a file or directory. The second major component is the block storage service which is responsible to handle data node cluster management like data block operations and data replication. The previous Hadoop version only supported a single namenode to manage the entire namespace where Hadoop v2 supports multiple NameNodes to manage multiple namespaces. The new introduction of multiple namespaces leads to horizontal scaling as well as improves the total performance of the Hadoop cluster.

2.5.3 YARN - Yet Another Resource Negotiator

As we mentioned before, Hadoop v1 was designed to run massive MapReduce jobs (see 2.5.1). The biggest disadvantage of MapReduce was that was working both as a data processing model and resource manager of the cluster. In addition, MapReduce started becoming insufficient due to issues regarding scalability and fault tolerance. YARN [7] was designed to solve the aforementioned issues.

The new architecture decouples the programming model from the resource management. In this new version, MapReduce is just one of the applications running on top of YARN. Other programming models that run on top of YARN are Spark, Storm, Hive and Pig. As the thesis focused on the designing of a Policy-
Driven YARN Launcher we skip the details of other programming models and we provide information about the YARN scheduler and the Spark framework.

As we can see from table 2.1, the advantages of YARN compared to MapReduce are plenty. First of all, it is more scalable in comparison with MapReduce. In addition, YARN allows multiple data processing model to use Hadoop as the common standard for batch. Moreover, YARN enables Hadoop to share resources dynamically among all the processing models and allows a more reasonable and finer-grained resource configuration for efficient cluster utilization. It would be interesting to take deeper look into YARN components in order to understand it better.

### 2.5.3.1 YARN Architecture and Components

In this section, we will briefly describe the YARN architecture and its major components. Figure 2.5 depicts the architecture of YARN.

![YARN Architecture](image)

**Figure 2.5: YARN Architecture [3]**

#### 2.5.3.2 Resource Manager (RM)

Resource Manager (RM) is the master that keeps track of the resources in a cluster and schedules applications. It is the arbitrator of the entire system and works
together with the per-node Node Managers and the per-application Application Master. RM has a pluggable component which allocates resources to applications and it is called YarnScheduler. There are different types of schedulers like FIFO, Capacity, Fair and Reservation-Based Schedulers.

2.5.3.3 Application Master (AM)

A newly introduced concept in YARN is the Application Master (AM). The AM is spawned by the scheduler and is executed in its own container. Every submitted application has its own AM which is responsible for monitoring and managing application’s execution. Also, AM communicates with the NM and negotiates needed resources from the RM. Hence, AM requests containers if needed or releases idle containers.

The integration of ApplicationMaster allows YARN to perform the following characteristics:

- **Scalability**: AM works like the traditional RMs. Shifting the execution of an application to the AM makes the system more scalable.

- **Open**: Moving all application code into the AM makes the system more open because it can support multiple frameworks such as MapReduce and Graph processing tools simultaneously.

2.5.3.4 Node Manager (NM)

Node Manager (NM) has similar functionality like workers in MapReduce. NM is a per-node agents which monitors the total status of its node. More precisely, it authenticates container leases, it monitors executions of an application and provides additional services to the containers. NM communicates with the RM and the AM and sends status of running containers and available resources of the node. Like launching a container, NM can also tear down a container upon a request from RM or AM. In addition, NM checks the node’s health and informs the RM sending it heartbeats periodically.

2.5.3.5 Containers

As we mentioned, an application requests resources via its AM from the RM. The Scheduler responds to a resource request by assigning a Container which satisfies the resource requirements. Essentially, a Container is a collection of physical resources (memory, CPU etc) which resides inside a node. A Container is supervised by the NM and is scheduled by the RM. Also every AM is the first assigned container of an applications and always called Container[0].
2.5.3.6 YARN Application Startup

In conclusion, we can summarize the start-up process of a YARN application. Figure 2.6 depicts that process in a more detailed description. The first action (1) is the submission of an application from the client to the RM. In the second step (2) the RM allocates a container for the Resource Scheduler. Afterwards, the RM contacts a NM (3) and the NM launches a new container for the AM (4).

![YARN Application Startup Diagram](image)

Figure 2.6: YARN Application start-up

2.5.4 Apache Spark

Another data processing model is the Apache Spark[15] which is a framework designed for fast computation. Originally, it was developed at AMPLab - UC Berkeley* in 2009 and it became an open-source Apache project in 2010. It is an in-memory(see section 2.5.4.1) data processing framework which allows data scientists to execute streaming, machine learning and graph computation that require fast iterative access to datasets.

Benefits of Apache Spark:

- **Speed**: Faster batch processing than MapReduce. Apache Spark executes data-processing jobs approximately up to 100 times quicker by utilizing in-memory storage.

- **Easy to use**: Apache Spark provides intuitive APIs for Java, Scala, Python and SparkR† with more than 100 operations for transforming and managing data.

---


† SparkR is an R package which allows data scientists to analyze big datasets and run jobs on them using the R shell - [https://spark.apache.org/docs/latest/sparkr.html](https://spark.apache.org/docs/latest/sparkr.html), Accessed: 7 July 2016.
2.5. Resource Management and Data Processing

- **A unified engine:** High level packages for SQL queries, stream data and graph processing are integrated into Spark packages.

### 2.5.4.1 Resilient Distributed Dataset (RDD)

Resilient Distributed Datasets (RDD) [16] is the fundamental data abstraction in Apache Spark which implements in-memory data processing. On the other hand, MapReduce uses HDFS to write the data. This method downgrades the performance of MapReduce especially for iterative and interactive operations where intermediate results must be used by other tasks (Map or Reduce). Matei Zaharia et al.[16] showed that Spark outperforms Hadoop by up to 20 times in graph processing and iterative machine learning applications because Spark avoids disk I/O and stores data in memory as objects.

We are presenting two figures which illustrate a Mapreduce iterative job using HDFS (Figure 2.7) and an interactive Spark application (Figure 2.8) using memory to store intermediate data.

#### Figure 2.7: Iterative operations on MapReduce

#### Figure 2.8: Interactive Operations on Spark

### 2.5.4.2 Spark Applications over YARN

This section gives a brief overview of how Spark applications run over YARN. It is important to understand that Spark applications run independently on a cluster. The *Driver* (also called Spark Context) program coordinates the Spark application...
and resides in a Driver node of the cluster. In order to run on a cluster, the SparkContext can connect to different types of cluster managers like YARN[7] or Mesos[17] but in our case we focus of YARN resource manager.

**Driver** is the component where SparkContext is created. Moreover, Driver translates RDD into execution graphs and splits the graph into stages. Another responsibility of Driver is to schedule the tasks among the executors and to monitor their execution state. On the other hand, executors store the data in cache in the JVM, perform data editing and execute all the data processing. Figure 2.9 illustrates the a Spark execution over YARN.

![Figure 2.9: Apache Spark over YARN](image)

### 2.5.4.3 Dynamic Allocation on Spark

Spark can dynamically scale up and down the number of executors allocated to an application[18]. The number of executor is decided by the workload of the cluster. For instance, when an application becomes idle, its resources can be released to the resource pool and be acquired by other applications or request them again when there is demand. We can enable/disable the dynamic allocation as well as to define lower and upper bounds of executors, executor idle timeout and timeout for
pending tasks. Dynamic allocation offers an efficient utilization of the resources of the cluster it is useful when multiple applications share resources in your Spark cluster.

2.5.4.4 Monitoring Spark applications

For the thesis purposes, we should have a monitoring system for Spark applications. Generally, there are several ways to monitor Spark applications like web UIs, metrics and external instrumentation[19]. Every SparkContext launches a web UI which displays information about the application. Since that information is only available while the application is running we should have a mechanism that reconstructs the UI even if the application is finished. Spark’s history server gives this opportunity through the application’s event logs. Hence, Spark history server displays completed Spark applications (succeed or failed) and gives useful information about task and scheduler stages, environmental variables and information about the running executors. Although spark history server exhibits valuable application information, a performance monitoring tools is needed in order to analyze better the applications. A new tools that introduced recently is Dr. Elephant which is presented in the next section.

2.6 Dr. Elephant

Dr. Elephant[20] is a performance monitoring tool for Spark applications that run over Hadoop ecosystem. It was developed at LinkedIn and became very popular because of its simplicity. LinkedIn uses Dr. Elephant for different tasks, including monitoring flows running on the cluster and reporting problematic executions.

Dr. Elephant automatically gathers all the metrics of an application, runs analysis on them and presents the results in a simple way. Its major goal is to increase the cluster efficiency by giving insights on how to properly tune a Spark job. Dr. Elephant communicates with the YARN RM to get a list of succeeded and failed applications and talks with the Spark history server to gather all the metadata for these applications. Once it has all the metadata, it executes a batch of heuristics and generates diagnosis reports. After the analysis, it categorizes applications with different level of severity. This grouping helps developers to decide how to tune their applications.

Dr. Elephant performs heuristics for:

- **Configuration best practices:** Examine the Spark configuration properties that the user defines before the execution of an application.
• **Event log limit:** Checks if the log file can be parsed. If the log file is too big then Dr. Elephant returns a severe diagnosis.

• **Job and Stage runtime:** Evaluates the execution time and the failure rate of an application.

• **Memory limit:** Calculates the memory utilization rate taking into account parameters like the AM memory, executor’s memory, total memory allocated for storage and the memory that used at peak.

• **Executor load balance:** Checks the load of executors (one executor overloaded and an other idle), it detects unused executors and gives proposals for a better optimization of the cluster.

Figure 2.10 presents one of the aforementioned heuristics that Dr. Elephant uses. The diagram describes the *Memory Limit* heuristic. In this example Dr. Elephant calculates the fraction of the used memory at the peak and the total allocated memory. After that calculation, the system categorizes the severity of the results based on thresholds that are defined by Dr. Elephant.

### 2.7 Hops

This section describes the platform that this thesis focuses on. Hops (Hadoop-as-a-Service) [8] is a new distribution of Apache Hadoop that supports project-based multi-tenancy, a secure way of sharing DataSets and projects across different users and an intuitive UI which is called HopsWorks. The resource management tool of Hops is YARN and the data processing frameworks currently supported are Apache Flink[21], Apache Spark[15] and Adam[22]. Concepts and ideas of this thesis (stated in Chapter 1) will be integrated into HopsWorks enriching its functionality.
Figure 2.10: Heuristic example on Dr.Elephant
Chapter 3

Research Method and Methodology

This chapter focuses on method and methodology that were used throughout the thesis and especially for conducting this research. Research methods and methodologies are very important when conducting research and they should be applied carefully before the starting of the research. As Anne Håkansson[23] has mentioned to her research paper many factors should be considered in order to conclude on suitable methods and methodologies for the thesis. Choosing methods and methodologies requires a literature study which gives the needed background knowledge in the research area and a good planning in order to reach the desired goals.

3.1 Research Method

For the thesis purposes, the Qualitative Research method applied during the entire research activity because the thesis focuses on developing artifacts for a Policy-Driven YARN Launcher. This research method suits to the thesis because we have only to prove our initial assumptions by examining the utility of that artifact. Moreover, because of the lack of massive data, we can not apply Quantitative Research method. Hopefully, a Quantitative Research method will be more suitable when the new feature will be accessible for many users on the production version of HopsWorks, where aggregated data from user behavior could be utilized.

3.2 Research Strategy

The research strategy guides the researcher for carrying out the research. A strategy helps the researcher to properly plan, design and conduct his/her research. The research strategy that was followed was the Case Study. Some use cases
like the inability of an inexperienced user to launch a job on YARN motivated this thesis. Moreover, the goal of the thesis can be achieved by evaluating the performance of the system in different cases, like the aforementioned one. The Case Study strategy was also suitable because it is based on an empirical observation and investigation of this particular problem and combines quantitative evidences like collected experimental data and qualitative proof through qualitative data.

3.3 Data Collection and Analysis

Data collection play a critical role in impact evaluation by providing evidence of processes followed and the final result. The method used in this thesis is the Case Study, where we can acquire data by analyzing a small number of cases in depth. Analytic Induction was the data analysis method that was applied for the thesis purposes because our idea was to examine different cases and adjust our system until none of the cases deviates from the hypothesis and the thesis objectives.

3.4 Proof of Concept

The Proof of Concept (POC) is a method that helps researchers to realize their ideas, demonstrate the feasibility of their theories and verify their concepts. POC becomes more and more critical in production as companies present a project and can prove that the core ideas are workable and applicable before going further. The use of POC drives companies and gives valuable feedback from the commercial usage. In our case, POC can be assumed as a suitable method as the thesis focuses on new concepts that might or might not be feasible. The verification of our ideas will be done by examining different case studies in order to confirm that the hypothesis and the thesis goal are functional and viable.
Chapter 4

Implementation

In this chapter the work done throughout this project will be presented thoroughly. Section 4.1 presents the integration of Dr. Elephant in the existing ecosystem and the necessary modifications applied for this purpose. Section 4.2 introduces a new feature of HopsWorks that is used for keeping track of applications which run in the system. The new feature is called Jobs History and presents details and heuristic results for an application. Finally, section 4.3 demonstrates the logic of the policy-driven mechanism that was designed for the thesis.

4.1 Dr. Elephant Integration

As described in the background, Dr. Elephant is a performance analysis tool for Spark and MapReduce applications that run on Hadoop. One of the thesis objectives was the integration of Dr. Elephant into HopsWorks platform.

Dr. Elephant runs a batch of Heuristics and evaluates executed jobs giving to user valuable feedback about proper ways to tune a job. The heuristics corresponds to different aspects of the application attributes such as memory, stage runtime, job runtime and executor load balance. After the completion of the evaluation, Dr. Elephant defines the severity of every heuristic that ran. Using the “Severity” property Dr.Elephant classifies properly tuned jobs from jobs with low performance because of a bad configuration. Dr. Elephant assigns five different types of severity that are showed at table 4.1. Different colors have assigned to different types of severity to be more visible through the UI.

The total severity of an application is based on the highest severity of its particular heuristics. For example if 4 out of 5 heuristics have LOW severity but the fifth has CRITICAL severity then the application is characterized as CRITICAL and needs some configuration improvements.

It is worth to mention that the different types of severity are based on
<table>
<thead>
<tr>
<th>Severity</th>
<th>Color</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>Blue</td>
<td>The job is safe. No tuning necessary.</td>
</tr>
<tr>
<td>LOW</td>
<td>Green</td>
<td>There is scope for few minor improvements.</td>
</tr>
<tr>
<td>MODERATE</td>
<td>Light Blue</td>
<td>There is scope for further improvement.</td>
</tr>
<tr>
<td>SEVERE</td>
<td>Orange</td>
<td>There is scope for improvement.</td>
</tr>
<tr>
<td>CRITICAL</td>
<td>Red</td>
<td>The job is in critical state and must be tuned.</td>
</tr>
</tbody>
</table>

Table 4.1: Heuristic Severity for Dr. Elephant

<table>
<thead>
<tr>
<th>Job Failure Limits</th>
<th>Severity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0.1</td>
<td>LOW</td>
<td>Low job failure rate, no improvement is needed.</td>
</tr>
<tr>
<td>&lt; 0.3</td>
<td>MODERATE</td>
<td>Medium job failure rate, there is scope for further improvement.</td>
</tr>
<tr>
<td>&lt; 0.4</td>
<td>SEVERE</td>
<td>High job failure rate. There is scope for improvement.</td>
</tr>
<tr>
<td>&gt;= 0.4</td>
<td>CRITICAL</td>
<td>Very high job failure rate. The job is in critical state and must be tuned.</td>
</tr>
</tbody>
</table>

Table 4.2: Job Runtime Heuristic - Thresholds and evaluation

thresholds that have been set by the designer and can be changed easily. Figure 4.1 illustrates the process of a heuristic, in this case the "Job Runtime Heuristic" heuristic which counts the number of failed and succeeded jobs and calculates a total percentage (the job failure rate). For that particular example we have set the job failure limits (thresholds) and the classification of the job severity is based on them (table 4.2 describes the process).

Dr. Elephant makes a few suggestions for every heuristic that runs in order to give the right insights to the user for a better configuration of an application. For instance, in our example a high Job Failure Rate can have multiple causes:

- Unstable implementations.
- Unbalanced work load.
- Limited allocated memory.
- Using more than two cores per executor in YARN.

* A Spark application can be split into multiple jobs.
After the completion of the analysis, Dr. Elephant stores every job detail (including job configuration and heuristic results) in a database. Hops and HopsWorks both use a MySql cluster [24] to store data. We configured Dr. Elephant to use the same MySql cluster for a smoother and more unified way of keeping all the data of our ecosystem.
4.2 Job History

Jobs History is a new feature of HopsWorks. Through Jobs History a user is able to see his/her successfully run Spark applications as well as the heuristic results that are coming from Dr. Elephant analysis. For every starting job several attributes are stored in the databases and records are updated upon specific events. For example, after the completion of an application the corresponded records are updated with the final execution status (Succeeded or Failed). Only jobs related to a particular project (and consequently to a specific user) are visible via the Jobs History tab. This means that only the owner of a job can see the history details of that job.

4.2.1 Model

Figure 4.2 illustrates the model that is used for the Job History feature. Only major components are depicted in the figure and minor ones are omitted for the sake of simplicity. Moreover, picture 4.2 depicts the general process flow for the Job History feature and numbers are used to describe the important actions of the entire system:

1. Run a Spark job over YARN.
2. Dr. Elephant asks YARN RM for executed jobs.
3. YARN RM responses with a list of executed jobs.
4. Dr. Elephant ask Spark History Server for details of a job.
5. Spark History Server sends the details.
6. Provide analysis of the jobs to HopsWorks.

HopsWorks fetches analytic data from Dr. Elephant by retrieving information from the database and by doing GET requests to Dr. Elephant. In our implementation, both ways are used in order to fetch the data and present them through the UI.

4.2.2 Relational Model

As it mentioned before, Dr. Elephant and Jobs History store data in the database and retrieve them upon a request. It will be useful to briefly present the relational model which is stored in the MySql cluster that Hops and HopsWorks use. Existing tables from Dr. Elephant framework were integrated with a new
imported table. Foreign keys (FK) are demonstrated in this figure and depict the relationships among the tables. A naming convention is used to allow the reader to direct easily tables with YARN attributes and understand the relational model as a whole. In figure 4.3 tables with the yellow color represent the relational model of Dr. Elephant where the green colored table depicts the newcomer jobs_history table that was created for the Job History. This new table is not connected with a FK but it makes primary key (PK) operations. In the jobs_history table the PK is the combination of the execution_id and the app_id and the app_id refers to PK of the yarn_app_result table.
Below, the tables functionality of the relational model used (Figure 4.3) are explained:

- **jobs_history**: Contains information about already successfully run or failed jobs that had been run through HopsWorks platform.

- **yarn_app_result**: This table presents general information about an application such as the total severity after the running of heuristics, the name of heuristic and execution duration of the application.

- **yarn_app_heuristic_results**: Contains results for every heuristic that it was run on an application, in our case five different heuristics apply (REFERENCE - BACKGROUND DR. E.). Actually, the general details of table yarn_app_result consists of a group of records in the yarn_app_heuristic_result and that is the reason for the many-to-one relationship between these two tables.

- **yarn_app_heuristic_results_details**: Rows of this table store data for each one heuristic that run on Dr. Elephant. The table contains details in depth that give a thorough description about the configuration of an application.

### 4.2.3 Job History webpage

Through the Jobs History tab a user can see all the necessary details of a run job. More specifically, basic job information are showed in the page such as:

- The application ID of a job
- The name of a job
- The owner/creator of a job
- The total duration of a job (the aggregate of both the queuing time and the running time)
- Job Type
- Job Severity (it derives from Dr. Elephant analysis)
- The configuration of a Job
  - Job Details
    - Main class name
    - Selected jar file
4.3 Policy-Driven Execution

As we already know, Hadoop is designed to let users tune their jobs but this has a few challenges:

- A user cannot optimize a job if he/she does not understand the internals of the framework.

- Critical information is scattered. This happens because many Hadoop components such as RM, NM, DN etc keep valuable information. Hence, a proper tuning should contain data gathered from different sources.

- Hadoop has a huge set of parameters that are correlated. Tuning some parameters may impact some other.

The aforementioned challenges prevent an inexperienced user to run a job over Hadoop. The thesis goal is to enable that kind of users to run applications on Hops and provide them a friendly environment where previous knowledge of Hadoop is not needed. Currently, running a job through HopsWorks requires a lot of parameters to be defined. For a Spark job, attributes such as the jar file which contains the executable Spark code, class name which represent the main class of the Spark job and the attributes of this class must be defined during the creation of the job. Moreover, a user must explicitly set attributes for the configuration of
Figure 4.4: Job History page
### 4.3. Policy-Driven Execution

#### Figure 4.5: Job History Details

<table>
<thead>
<tr>
<th>Job Details</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Application ID</td>
<td>application_147105580794_0002</td>
</tr>
<tr>
<td>Application Name</td>
<td>SparkPi-500</td>
</tr>
<tr>
<td>Application Severity</td>
<td>default</td>
</tr>
</tbody>
</table>

#### Configuration

<table>
<thead>
<tr>
<th>Configuration Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main class name</td>
<td>org.apache.spark.examples.SparkPi</td>
</tr>
<tr>
<td>Jar File</td>
<td>spark-examples-1.5.2-hadoop2.6.0.jar</td>
</tr>
<tr>
<td>Job arguments</td>
<td>1000</td>
</tr>
</tbody>
</table>

#### Heuristic Results

<table>
<thead>
<tr>
<th>Heuristic Name</th>
<th>Severity</th>
<th>Score</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark Configuration Best Practice</td>
<td>NONE</td>
<td>0</td>
<td>Drive Memory</td>
<td>512m</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Executor Cores</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Serializer</td>
<td>org.apache.spark.serializer.KryoSerializer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Stale Manager</td>
<td>Not presented. Using default</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Spark Memory Limit</th>
<th>NONE</th>
<th>0</th>
<th>Memory utilization rate</th>
<th>0.000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total driver memory allocated</td>
<td>512 MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total executor memory allocated</td>
<td>1.0 GB (1 GB x 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total memory allocated for storage</td>
<td>627.76 MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total memory used at peak</td>
<td>0.0</td>
</tr>
</tbody>
</table>
a job.

The **configuration attributes** are:

1. Application Master memory in MB.
2. Application Master virtual cores.
3. Number of Executors.
4. Number of Executor cores.
5. Executor memory in MB.

The aforementioned attributes are difficult to be decided by an inexperienced user and a mechanism should be found to ease this problem. In addition, users would have the chance to choose different run policies of their applications. Policies describe different ways of tuning and running an application on Hadoop and they mainly depend on the initial configuration of a job (AM memory, number of AM Vcores, number of executors, executor memory etc.).

Two different policies have been designed for this purpose:

1. **Minimal Configuration:** A minimal configuration proposes the minimum amount of resources needed for launching a specific application. Resources refer to memory and Vcores of the AM and to the number of executors requested for the job. The advantage of the minimal configuration is that the user does not waste resources and for this reason the cost of launching an application is low. The disadvantage is that less resources give less performance to a job. For instance, decreasing the number of executors increases the total running time of an application.

2. **Fast Configuration:** A fast configuration suggests optimal amount of resources for a quick execution of an application. Running time of an application is the most important attribute for making a good proposal. This policy refers to users who want to run an application as fast as possible and it is suitable in cases where the execution duration of a job is critical.

At this point, an attentive reader should have a rational question: "**How does the system know which is a suitable minimal and/or fast configuration of a specific application?**"

The answer in this question is straightforward and simple. The **policy-driven mechanism** utilizes the **Job History** feature which introduced before. This
Policy-driven mechanism means that the policy-driven mechanism has a pool of executed applications, it examines these applications and concludes to a minimal and a fast configuration proposals. We can imagine that an empty pool of applications does not give data for evaluation, in that case the policy-driven mechanism does not propose anything. On the other hand, a big pool gives more accurate proposals since more data are under evaluation. Initially, the pool is empty because the system has not run any application. For this reason, the system should be "taught" by running different types of applications with several configurations. Figure 4.6 depicts the policy-driven mechanism architecture and its basic components. As we can see from the picture, the policy-driven mechanism consists of two sub components. The first one is the search engine which tries to find relative jobs and the second one is the analysis component which receives a set of jobs and calculates job configuration proposals which satisfy our policies.

Policy-driven mechanism selects run applications from the history and analyzes them. The pool of jobs that is used for the selection phase can contain different kinds of jobs. For example, Spark jobs can be RDD Api\(^\ast\) jobs, DataFrame API\(^\dagger\) jobs or Machine Learning API\(^\ddagger\) jobs. The search engine component of the policy-driven mechanism selects relevant jobs. Relevant jobs are jobs with similar attributes (same Spark class name, same Spark jar file etc.) with the job which is under creation. Hence, the search engine component takes all the job attributes and looks for similar jobs with the same attributes. The found jobs have a special characteristic which represents how similar are two applications, this characteristic is called degree of similarity. Table 4.3 presents the different

\[^{\ast}\] Spark operates using the concept of a resilient distributed dataset (RDD)  
\[^{\dagger}\] DataFrame is a Dataset organized into columns and can be constructed by various data such as structured data files, existing RDDs and external databases.  
\[^{\ddagger}\] Spark’s Machine Learning (ML) library, supports many distributed ML algorithms
degrees of similarity and how they have been designed.

After the search for relevant jobs, the analysis component starts the evaluation process in order to end up to possible configurations for the job. The analysis process tries to satisfy the two policies (minimal and fast) that described above. It is possible to not have a proposed configuration because of several reasons. A reason could be zero relevant results from the search engine or inability to propose a fast configuration (for example if we have only one relevant job, in this case only the minimal configuration is proposed). It is important to highlight that only jobs with "LOW" severity (from the analysis of Dr. Elephant) are taken into account for two reasons:

- **Avoid failed jobs:** We want to exclude failed jobs. A failed job has a "bad" configuration which leads to inappropriate execution. Excluding these executions we know that the proposed configuration attributes are suitable for the job and the job will be executed successfully.

- **Find an optimal tuning:** Dr. Elephant heuristics measure the performance of a job using several heuristics to define the severity of a job. There are many scenarios where a job is executed successfully but the configuration is not optimal. For instance, if there are unused executors of an application, then there is waste of resources and this application is characterized as "Critical". This application should be excluded from the analysis because it fails to comply with the server optimization standards that have been set in this thesis.
4.3. POLICY-DRIVEN EXECUTION

4.3.1 Auto-configuration panel

The policy-driver mechanism is integrated into HopsWorks enabling users to select different job configurations according to their desires. The initial “job configuration” panel of HopsWorks remains the same for users who prefer to tune their applications manually. In addition, a new panel named “Pre-Configuration” has been added one step before the “Manual Configuration” panel. The new inserted panel implements the logic of the policy-driven mechanism. Hence, using this panel a user receives at most two different configuration choices (a minimal and a fast configuration). The described panel is depicted in figure 4.7. As we can see, the new feature provides an intuitive and friendly configuration environment where every user can tune an application easily.

In this particular example (figure 4.7) the policy-driven mechanism found three similar results with a “VERY HIGH” similarity and returns two configuration options to the user. As we can see the fast configuration is slightly faster than the minimal configuration. The difference is not so high because this application has naturally a low execution duration. The difference will be higher for intensive applications which require much resources and much time to be completed. We will thoroughly examine this case in to the next chapter which describes the evaluation phase.
Chapter 5

Evaluation

This chapter presents the evaluation process that followed in order to prove our initial assumptions. As described in Chapter 3, the Qualitative Research method applied throughout the research. The research strategy that followed was the Case Study and our proof of concept will be based on the analysis of a few Use Cases in depth.

5.1 Use cases

For the evaluation purposes, two use cases will be examined. Both cases belong to RDD API Examples. The first example deals with the \( \pi \) (pi) estimation and the second use case is the Word Count example.

5.1.1 Pi Estimation

This application is a very common Spark job which is used widely for testing, it consists of compute-intensive tasks and it is written in Java. This job estimates the value of \( \pi \) by ”throwing darts” at a circle. The application picks random points in the unit square (0,0) to (1,1) and detects how many fall in the unit circle. The estimated fraction should be \( \pi/4 \), so we have the approximate value for \( \pi \). The main Java class takes one argument which is the number of random points that the program picks in order to calculate \( \pi \). The more points we have the more accurate value will be estimated.

The evaluation will be based on monitoring the performance of the system and if the policy-driven mechanism successfully picks the best configurations that satisfy the minimal and fast policies. Figure 5.1 depicts the performance of the system for different pi executions. In our example we have chosen pi executions with 1000, 2500 and 5000 random points. The y axis represents the execution duration in sec. The execution duration is the time needed from the submission of
Figure 5.1: Pi estimation - execution

an application until the successful or failed completion of that application. Also, the queuing time is supposed to be zero because we submit applications if and only if the server is "idle" and no application is running at that point. The x axis depicts the number of executors that assigned for this particular application. Each executor has one Vcore and 1024 MB of memory. The purpose of this experiment was to observe the behavior of the pi application when the number of executors varies from 1 to 6.

As we can see, the application fails when the executor memory is less than 512 MB. Consequently the minimum amount of the executor memory that should be used is 512 MB, this is the _minimal configuration_ and is highlighted with the green colored point in the plot. Moreover, the _fast configuration_ is highlighted with the pink colored point and presents the amount of executors that should be used in order to have the minimum execution duration. An interesting behavior that should me mention is that the execution duration increases at some point even if the number of executors increases. This happens because the application has already the needed executors for the execution and the additional executors takes some time to be launched. In other words we have redundant executors that downgrade the performance of the execution.

Having all these records in the _history_, the _auto-configuration_ mechanism successfully detects the _minimal_ and the _fast_ configurations for this particular job.
5.1.2 Word Count

This Spark application is a transformed version of WordCount MapReduce example. In this version, the aim is count words in distributed fashion for a given passage. The application reads a text document, counts the number of times each word appears and outputs a list with the results. For input text, we used three files which contain “dummy” content. These three files are a small, a medium and a big file with sizes 151.8 MB, 397.2 MB and 1.2 GB respectively. The logic that followed for this use case is exactly the same as the previous example. We try to monitor the performance of the Word Count execution by changing the number of the executors the application uses. In this example every executor has one Vcore and 512 MB of memory and the application runs using one to twelve executors. Figure 5.2 illustrates the execution behavior of the Word Count example. As previously, the x axis represents the number of executors used from the application and the y axis depicts the execution duration of the application. The minimal and the fast configurations are depicted with the yellow and the pink points respectively.

![Figure 5.2: Word count example - execution](image)

An interesting thing that derives from the Figure 5.2 is that the execution fails if the application uses one only executor with memory less than 512 MB. Moreover, we have to highlight that the application runs if we assign 1-12 executors with 512 MB but some of these configurations are not optimal for the application. The same behavior with redundant executors is also visible in this example. The execution duration increases even if the number of executor
increases and this is caused by the fact that the application has already the executors it needs and the system takes more time to launch many executors.

As we can see from the plot, the red line indicates a bad configuration (with critical severity) that derives from Dr. Elephant’s analysis. In this particular example two are the possible reasons that cause an non optimal execution. The first reason is that there are unused executors, with zero assigned tasks for example. The second reason could be an unbalanced workload of every executor. Some executors are getting much more tasks than others.

In this case, the policy-driven mechanism will choose the minimal and the fast configuration that are highlighted in the figure and represent configurations with low severity. But even when that is not happening, the policy-driven mechanism excludes configurations with critical severity from its analysis. This means that it is guaranteed that every configuration proposed by the policy-driven mechanism successfully runs and is executed optimally.

For these two Use Cases that we examined, the policy-driven mechanism successfully selected the most suitable configuration attributes for each application. The same logic applies in every Spark application that runs on HopsWorks. At the moment, the analysis of two Use Cases was enough for our evaluation but a more complete evaluation method could be examining the policy-driven mechanism in a real system. This will happen after the commercial release of HopsWorks when common users will run applications and will interact with HopsWorks. In this case the bulk of data will be huge and the evaluation stage will be based on this data, testing if the system work in various cases.
Chapter 6

Conclusion and Future Work

This chapter presents the epilogue of this thesis. Section 6.1 contains our conclusions about the work done and Section 6.2 presents the next possible steps in continuing this work and enhancing the current system.

6.1 Conclusion

This report discusses the design and implementation of a Policy-Driven YARN Launcher. The first chapters of the report present the problems that arise during the definition of a new application which runs in HopsWorks. The thesis focuses especially in inexperienced users of HopsWorks who desire an easy way of running an application. After the demonstration of the necessary background and the model used, the next chapter presents the implementation ideas for the policy-driven mechanism. Policies for different executions have been designed and the policy-driven mechanism implements these policies. The mechanism analyzes existing applications and provides suitable configuration choices to the user. Having this new feature a user does not struggle with tuning an application. This enables HopsWorks to be a platform for normal people, not just data engineers.

6.1.1 Goals Achieved

This work achieves the goals that stated in Chapter 1. The purpose of this thesis was to develop a human-friendly YARN application launching environment where the system takes responsibility for allocating the necessary resources for running an application, this is applied by designing policies and the policy-driven mechanism which is responsible for resource allocation. Moreover, the solutions should be efficient and fair to applications. The evaluation of the system verifies our initial hypothesis and presents how the policy-driven mechanism reacts with
the existing system. The results show that the policy-driven mechanism satisfies our initial ideas and enhances the current system.

6.2 Future Work

This work showed a way for the implementation of a Policy-Driven YARN Launcher. The proposed solutions are applicable and are integrated properly in the existing system. However, the continuously development of IT systems requires future steps that could improve further our existing solutions. Future work includes mainly the following:

- Extension to other data-processing frameworks: The thesis ideas are applied to Spark data-processing framework. Both the Jobs History feature and the policy-driven mechanism get involved with Spark applications only. A further improvement could be the implementation of the same logic in other data-processing frameworks such as Flink and Adam.

- Failure Prediction: As mentioned, a user has two choices for configuring a new job. The first is a auto-configuration choice which is implemented by the policy-driven mechanism and the second is the manual-configuration choice. Using the auto-configuration choice, a user knows that the application will be executed successfully, but this does not happen if the user tunes the application manually. A great improvement of the system could be an execution ”prediction” of an application when the user configures a job manually. This new idea will be useful especially when an application tends to fail.

- Evaluation: In the evaluation chapter, we examined two Use Cases and we found that our mechanism is applicable and works as expected. The lack of vast data led us to examine only two Use Cases. A better evaluation process would be to gather massive data from the users. After the commercial release of HopsWorks, it will be possible to collect users data and examine again the behavior of the policy-driven mechanism.
Bibliography


