UNIVERSIDAD DE GRANADA

Bachelor's Thesis
Computer Science Degree

BIG DATA SERVICES ON CLOUD COMPUTING

Author

Rshad Zhran

Directors

José Manuel Benítez Sánchez

Manuel Jesús Parra Royón

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicaciones
ETSIIT

Granada, September of 2018

BIG DATA SERVICES ON CLOUD COMPUTING

Rshad Zhran

Key Words: OpenStack, Spark, HDFS, Ansible, Processing, Storage

Abstract

This bachelor's thesis is focused on the design and deployment of Big Data services over cloud computing platforms.Inspired on the OCCML[1] platform for cloud services the frequent services for Big Data computation has been analyzed and formalized through a standardized catalogue. The proposal has been implemented over a cluster running under OpenStack IaaS software, with Hadoop and Spark for Big Data services.

[1] OCCML is an abbreviation of Open Cloud Computing Machine Learning project. This project is a property of the University of Granada, directed by the professor *Jose Manuel Benitez*. [30]

Me, *Rshad Zhran*, student of Bachelor's Degree of Computer Science Engineering in *The Higher Technical School of Information Technology and Telecommunications Engineering of the University of Granada*, with NIE Y2944240R, authorize the existence of this copy of my bachelor's thesis in the library of the center of the institute, to be accessible for any interested person.

Signed: *Rshad Zhran*

Granada, 7 of September of 2018

Acknowledgement

My sincere thanks go to my advisors, *José Manuel Benitez* and *Manuel Jesús Parra Royón* for the continuous support, for their patience, motivation, enthusiasm, and immense knowledge.

Also I would like to thank my family and friends, for the huge support during all those beautiful last 4 years, making me able to complete my career with all the difficulties I passed through.

Finally, I must not forget to say, many thanks for all the professors I knew in our institute, the officers, servants and all who formed a beautiful part of those 4 years in our university.

As we express our gratitude, we must never forget that the highest appreciation is not to utter words, but to live by them. – ***John F. Kennedy***

D. José Manuel Benitez Sanchez, Professor in the area *Soft Computing and Intelligent Information Systems* of the *Department of Artificial Intelligence and Computational Science* of the *University of Granada*. Also he is the Lab Director of the investigation group, *Distributed Computational Intelligence and Time Series Lab 'DiCITS lab'.*

D. Manuel J. Parra Royón, PhD student in *The Higher Technical School of Information Technology and Telecommunications Engineering of the University of Granada.* Also he is BigData Architect and Engineer and a member of the investigation group, *Distributed Computational Intelligence and Time Series Lab 'DiCITS lab'.*

advice:

This present bachelor thesis, titled as *BIG DATA SERVICES ON CLOUD COMPUTING*, has been developed under their supervision by *Rshad Zhran* and they authorize the defense of the mentioned thesis under the evaluation of the correspondent court members.

And to be recognized, they sign the present form in *Granada, 7 of September of 2018*.

Directors:

José Manuel Benitez Sanchez *Manuel J. Parra Royón*

Table of Contents

List of Figures

Chapter 1

1. Introduction

You can’t read a technology journal or blog or even your local newspaper without coming upon a reference to cloud computing. While there’s been a lot of debate about what cloud computing is and where it’s headed, no one has doubts that it is real. It will change the way we deploy technology and how we think about the economics of computing.

Cloud computing is more than a service sitting in some remote data center. It’s a set of approaches that can help organizations quickly, effectively add and subtract resources in almost real time. the cloud is as much about the business model as it is about technology.

Cloud computing isn’t a quick fix. It requires a lot of thought: Which approach is most appropriate for your company? For example, companies have to decide if they want to use public (external) cloud services or if they want to have private clouds behind their firewalls. [1]

Also when we talk about the cloud, we talk about data, and to be more specific, it’s about Big Data.

Data is a commodity, but without ways to process it, its value is questionable. Data science is a multidisciplinary field whose goal is to extract value from data in all its forms. This article explores the field of data science through data and its structure as well as the high-level process that you can use to transform data into value.

Data science is a process. That's not to say it's mechanical and void of creativity. But, when you dig into the stages of processing data, from munging data sources and data cleansing to machine learning and eventually visualization, you see that unique steps are involved in transforming raw data into insight.

Thanks to the promise of new insights for innovation and competitiveness from Big Data, data science has gone mainstream. Executives are spending billions of dollars collecting and storing data, and they are demanding return on their investment. Simply getting the data faster is of limited value, so they are seeking to use the data to enrich their day-to-day operations and get better visibility into the future.[2]

1.1 Big Data

Big data is a term that describes the large volume of data – both structured and unstructured – that inundates a business on a day-to-day basis. But it's not the amount of data that's important. It's what organizations do with the data that matters. Big data can be analyzed for insights that lead to better decisions and strategic business moves. [3] [4]

While the term "big data" is relatively new, the act of gathering and storing large amounts of information for eventual analysis is ages old. The concept gained momentum in the early 2000s when industry analyst Doug Laney articulated the now-mainstream definition of big data as the three Vs:

- **Volume.** Organizations collect data from a variety of sources, including business transactions, social media and information from sensor or machine-to-machine data. In the past, storing it would've been a problem – but new technologies (such as Hadoop) have eased the burden.
- **Velocity.** Data streams in at an unprecedented speed and must be dealt with in a timely manner. RFID tags, sensors and smart metering are driving the need to deal with torrents of data in near-real time.
- **Variety.** Data comes in all types of formats – from structured, numeric data in traditional databases to unstructured text documents, email, video, audio, stock ticker data and financial transactions.

According to SAS analytics company [3], two more Vs. are considered:

- **Variability.** In addition to the increasing velocities and varieties of data, data flows can be highly inconsistent with periodic peaks. Is something trending in social media? Daily, seasonal and event-triggered peak data loads can be challenging to manage. Even more so with unstructured data.
- **Complexity.** Today's data comes from multiple sources, which makes it difficult to link, match, cleanse and transform data across systems. However, it's necessary to connect and correlate relationships, hierarchies and multiple data linkages or your data can quickly spiral out of control.

1.2 BigData and Cloud Computing

1.2.1 Cloud Computing

it is a term that refers to on-demand computer resources and systems that can provide a number of integrated computer services without being bound by local resources to facilitate user access. These resources include data storage, backup and self-synchronization, as well as software processing and scheduling tasks [19]. Cloud computing is a shared resource system that can offer a variety of online services such as virtual server storage, and applications and licensing for desktop applications. By leveraging common resources, cloud computing is able to achieve expansion and provide volume. [5]

1.2.2 What Is Big Data's Relationship To The Cloud?

How does the cloud computing environment correspond to big data? The answer to this question reflects the relationship between them. This is done through the cloud computing features to handle big data, the resources provided by cloud computing, the resource service to provide service to many users where the various physical and virtual resources are automatically set and reset upon request. Cloud computing has access from anywhere to data resources that are spread all over the world by using a (public) cloud to allow those sources faster access to storage. The nature of big data is generated by technologies and locations worldwide, so the cloud resource service provides and helps in the collection and storage of big amounts of data resulting from the use of technologies. [5]

The cloud computing structure can expand the solid equipment to accommodate small and big data volumes. The cloud can expand to handle big amounts of data by dividing the data into parts, automatically done in IAAS [1]. Expanding the environment is a big data requirement. Cloud computing has the advantage of helping to reduce costs by paying for the value of the resources used, which helps to develop big data. Flexibility is also regarded a requirement for big data. When we need more storage for data the cloud platform can dynamically expand to meet proper storage needs when we would like to handle a large number of virtual machines in a single time period. For error tolerance, the cloud helps to handle big data in the extraction and storage process. Error tolerance helps SLAs, as well as QOS levels. Service level agreements specify different rules for regulating availability of cloud service. [5]

Chapter 2

Objectives

The objectives set to be accomplished through this project are:

- ❏ Define a standardized catalogue for Big Data services on cloud platforms
- ❏ Develop an implementation for the Big Data service catalogue
- ❏ Deploy the services over a platform with OpenStack, Hadoop and Spark

Chapter 3

Cloud Architecture and Cloud Operating System OpenStack

In this chapter, we will explain the cloud architecture for both cases, generally and for our project, reasoning about each made decision for the architecture design and implementation. We'll also get to know OpenStack and how we adapted it to our needs.

Finally, We'll get into the analysis, design and implementation of our portal script, using OpenStack API in Python.

3.1 Introduction to Cloud architecture

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models. [6]

3.1.2 Essential Characteristics

a. On-demand self-service.

A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider. [6]

b. Broad network access.

Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations). [6]

c. Resource pooling.

The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth. [6]

d. Rapid elasticity.

Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time. [6]

e. Measured service.

Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service [6]

3.1.3 Service Models

Software as a Service (SaaS).

The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings. [6]

Platform as a Service (PaaS).

The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider.3 The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.[6]

Infrastructure as a Service (IaaS).

The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).[6]

3.1.4 Deployment Models

Private cloud.

The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.[6]

Community cloud.

The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.[6]

Public cloud.

The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.[6]

Hybrid cloud.

The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds). [6]

3.2 Project's Cloud Architecture

In our case, to go into the cloud architecture implementation we had to configure our hardware base, by applying the following steps on a server of n machines:

1. Network Configuration between the nodes "machines"
2. Installing CentOS on each node.
3. Installing IaaS layer representative operating system.

Once we had the hardware base prepared, we can proceed with IaaS service model implementation.

3.2.1 OpenStack choice as IaaS service model

For Iaas implementation, the market offers a lot of options, with many different characteristics, advantages and disadvantages. Options such as AWS, AZURE Stack, On-Premises Alternatives, Stratoscale Symphony, Kubernetes and Containers OpenStack, etc ...

Between all of the alternatives, we chose OpenStack to be our IaaS layer operating system, for many reasons, we mention on following:

1. **Open source**

Some say that OpenStack is the largest open source project ever. It combines computing, networking, and storage subsystems in layers via APIs within a single platform. The open source environment allows you to create a truly software defined data center. For instance, you can modify the APIs to enhance integration and subsystem usage as well as extend resource orchestration beyond your OpenStack environment to the public cloud. Some vendors, such as RedHat, have developed and continue to support their own distributions of OpenStack. And with a community of over 200 active enterprise participants, you should have no trouble finding tailored OpenStack solutions to fit your needs along with the necessary support. [7]

The challenge: *As an open source comprehensive project, influencers such as Randy Bias claim that OpenStack is lacking real leadership. Obviously this is a significant disadvantage, considering the fact that CIOs need to plan deployments years ahead and ensure that they eliminate future risks by choosing the right solution and vendors.*

2. **Hybrid cloud**

Many enterprises see the public cloud as an extension of their data centers. A recent survey from Peer 1 Hosting concluded, with 28% participant approval, that hybrid cloud adoption is likely to triple over the course of the next three years.
Following this hybrid cloud adoption trend, leading public cloud vendors such as Google have been embracing the concept of a hybrid cloud via OpenStack. Another particularly good example is Eucalyptus, which was acquired by HP and has taken hybrid cloud accessibility one step further by enhancing OpenStack's compatibility with AWS. [7]

The challenge: *IT organizations that decide to look for a hybrid cloud solution will find that OpenStack APIs don't support out-of-the-box integration with various leading public clouds. While there is great potential, integration requires hefty investments to fit the solution to specific needs.*

3. Orchestration

Without orchestration, Infrastructure-as-a-Service (IaaS) platforms remain stuck in confined environments. Orchestration is the comprehensive automated provisioning of infrastructure based on specific application requirements such as support for multi-tier, distributed applications that can utilize sporadic resources for particular independent subsets (services).

"OpenStack is a platform that frees users to run proven technologies like VMs as well as new technologies like containers," noted OpenStack COO, Mark Collier in Google's recent announcement to sponsor OpenStack. "With Google committing unequaled container and container management engineering expertise to our community, the deployment of containers via proven orchestration engines like Kubernetes will accelerate rapidly."
Containers only require a thin OS to run applications, providing the ability to increase workload density in an OpenStack environment without having to purchase more hardware. The combination consequently results in a higher level of utilization and efficiency. With containers, moving workloads is nearly seamless between OpenStack, VMware and public clouds, like AWS and Google Cloud Platform (GCP), that support containers, creating infrastructural efficiency and scalability. [7]

The challenge*: OpenStack still lacks the out-of-the-box robustness that enterprises desire for their data centers, including IT management features such as availability, scalability and security. Data center operations teams still can't completely guarantee that workloads running in OpenStack will be able to maintain preferred performance levels without extraneous measures to ensure optimal levels of infrastructure utilization.*

3.3.1 OpenStack Global Architecture

Always, when we talk about an IaaS service model, we have to mention 3 main components, ***Compute***, ***Networking*** and ***Storage***. Additionally, we can add 2 more elements, a ***Dashboard*** and ***APIs***.

OpenStack implements these components, offering many alternatives for each one. On following, we describe the different alternatives for each component of an Iaas service model. [8]

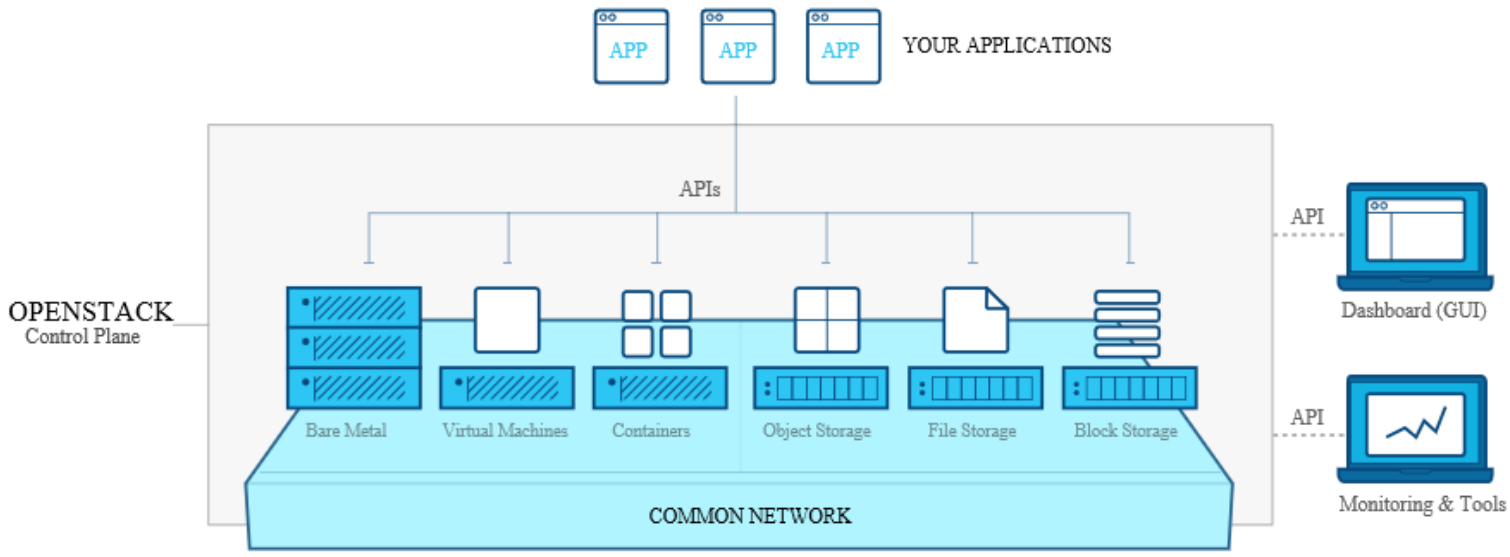

Figure 1. openstack global architecture [5]

IaaS layer Components in OpenStack

To talk about the alternatives, we will study component by components of each one mentioned recently. As we have mentioned before, the main components are:

- Compute
- Networking
- Storage

Complementary elements:

- Dashboard
- API

Compute

The OpenStack Compute service allows you to control an Infrastructure-as-a-Service (IaaS) cloud computing platform. It gives you control over instances and networks, and allows you to manage access to the cloud through users and projects.

Compute does not include virtualization software. Instead, it defines drivers that interact with underlying virtualization mechanisms that run on your host operating system, and exposes functionality over a web-based API. [9]

In our case we talk about ***OpenStack Compute*** (nova). **Nova** is the OpenStack project that provides a way to provision compute instances (aka virtual servers). Nova supports creating virtual machines, bare metal servers (through the use of ironic), and has limited support for system containers. Nova runs as a set of daemons on top of existing Linux servers to provide that service. [8]

It requires the following additional OpenStack services for basic function:

- **Keystone**: This provides identity and authentication for all OpenStack services.
- **Glance**: This provides the compute image repository. All compute instances launch from glance images.
- **Neutron**: This is responsible for provisioning the virtual or physical networks that compute instances connect to on boot.

It can also integrate with other services to include: persistent block storage, encrypted disks, and ***baremetal*** compute instances.

To check Nova version on our installation, we should run:

```
[CC_rashad@atcstack ~]$ nova-manage --version
>> 14.0.9
```

In this case, we use the version 14.0.9 .

Keystone

Keystone is an OpenStack service that provides API client authentication, service discovery, and distributed multi-tenant authorization by implementing OpenStack's Identity API.

This documentation is useful for contributors looking to get involved in our community, developers writing applications on top of OpenStack, and operators administering their own OpenStack deployments.

This documentation is generated by the Sphinx toolkit and lives in the source tree. Also see the Getting Involved page for other ways to interact with the community. [10]

Keystone provides a script, to make the access available from *OpenStack cli*. In our case this script looks like the following script.

```
#!/usr/bin/env bash
# To use an OpenStack cloud you need to authenticate against the Identity
# service named keystone, which returns a **Token** and **Service Catalog**.
# The catalog contains the endpoints for all services the user/tenant has
# access to - such as Compute, Image Service, Identity, Object Storage, Block
# Storage, and Networking (code-named nova, glance, keystone, swift,
# cinder, and neutron).
#
# *NOTE*: Using the 3 *Identity API* does not necessarily mean any other
# OpenStack API is version 3. For example, your cloud provider may implement
# Image API v1.1, Block Storage API v2, and Compute API v2.0. OS_AUTH_URL is
# only for the Identity API served through keystone.

export OS_AUTH_URL=http://atcstack.ugr.es:5000/v3/

# With the addition of Keystone we have standardized on the term **project**
# as the entity that owns the resources.
export OS_PROJECT_ID=66ecf87069c3480aab08172d3a581ab8
export OS_PROJECT_NAME="CCprojectRashad"
export OS_USER_DOMAIN_NAME="Default"
if [ -z "$OS_USER_DOMAIN_NAME" ]; then unset OS_USER_DOMAIN_NAME; fi

# unset v2.0 items in case set
unset OS_TENANT_ID
unset OS_TENANT_NAME

# In addition to the owning entity (tenant), OpenStack stores the entity
# performing the action as the **user**.

export OS_USERNAME="CC_rashad"

# With Keystone you pass the keystone password.
echo "enter your OpenStack Password, project $OS_PROJECT_NAME as user
$OS_USERNAME:"

read -sr OS_PASSWORD_INPUT
export OS_PASSWORD=$OS_PASSWORD_INPUT

# If your configuration has multiple regions, we set that information here.
# OS_REGION_NAME is optional and only valid in certain environments.
export OS_REGION_NAME="RegionOne"

# Don't leave a blank variable, unset it if it was empty
if [ -z "$OS_REGION_NAME" ]; then unset OS_REGION_NAME; fi
```

```
export OS_INTERFACE=public
export OS_IDENTITY_API_VERSION=3
```

In this case, we only have to run the following command, which require our user's password.

```
[CC_rashad@atcstack ~]$ source <script name>.sh
Please enter your OpenStack Password for project .. as user .. :
[CC_rashad@atcstack ~]$
```

Once we are authenticated, then we car run all the related commands to manage OpenStack.

¿What about the available images, and how can we manage them?

Glance

OpenStack offers Glance. A service which provides the compute image repository. All compute instances launch from glance images. Glance image services include discovering, registering, and retrieving virtual machine (VM) images. Glance has a RESTful API that allows querying of VM image metadata as well as retrieval of the actual image. [11]

Glance, as with all OpenStack projects, is written with the following design guidelines in mind:

- **Component based architecture**: Quickly add new behaviors
- **Highly available**: Scale to very serious workloads
- **Fault tolerant**: Isolated processes avoid cascading failures
- **Recoverable**: Failures should be easy to diagnose, debug, and rectify
- **Open standards**: Be a reference implementation for a community-driven api

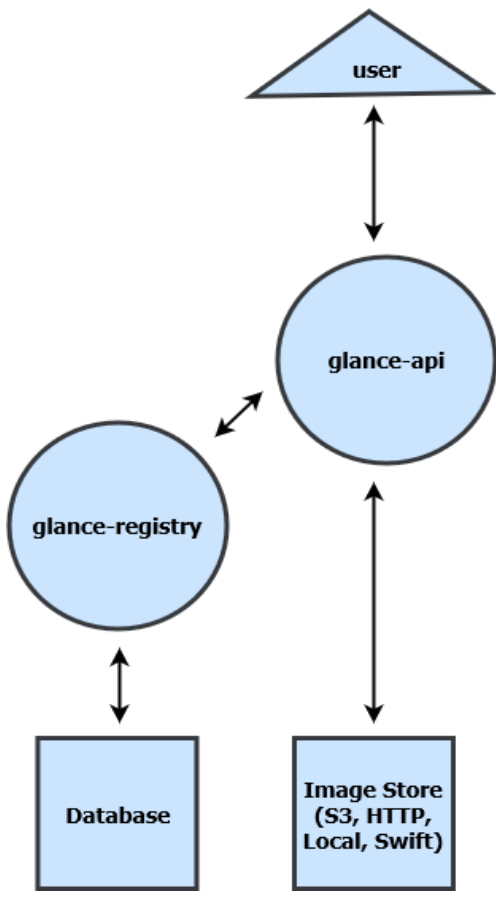

Figure 2. Glance Logical Architecture

¿ And How is Networking handled in OpenStack ?

Neutron

Neutron is an OpenStack project to provide “network connectivity as a service” between interface devices (e.g., vNICs) managed by other OpenStack services (e.g., nova). It implements the Neutron API.

This documentation is generated by the Sphinx toolkit and lives in the source tree. Additional documentation on Neutron and other components of OpenStack can be found on the OpenStack wiki and the Neutron section of the wiki. The Neutron Development wiki is also a good resource for new contributors.

OpenStack Networking (neutron) manages all networking facets for the Virtual Networking Infrastructure (VNI) and the access layer aspects of the Physical Networking Infrastructure (PNI) in your OpenStack environment. OpenStack Networking enables projects to create advanced virtual network topologies which may include services such as a firewall, a load balancer, and a virtual private network (VPN).

Networking provides networks, subnets, and routers as object abstractions. Each abstraction has functionality that mimics its physical counterpart: networks contain subnets, and routers route traffic between different subnets and networks.

Any given Networking set up has at least one external network. Unlike the other networks, the external network is not merely a virtually defined network. Instead, it represents a view into a slice of the physical, external network accessible outside the OpenStack installation. IP addresses on the external network are accessible by anybody physically on the outside network. [12]

3.4 OpenStack Compute API

The nova project has a RESTful HTTP service called the OpenStack Compute API. Through this API, the service provides massively scalable, on demand, self-service access to compute resources. Depending on the deployment those compute resources might be Virtual Machines, Physical Machines or Containers. [13]

The compute high-level interface is available through the **compute** member of a **Connection** object. The **compute** member will only be added if the service is detected.

Purposes of Making Use of OpenStack API

We use OpenStack API to make it easier to deal with OpenStack instances management. In our project we have different functions to implement with OpenStack API, such as:

- Create an instance
- Delete an instance

We use OpenStack Compute API in order to implement the required functionalities.

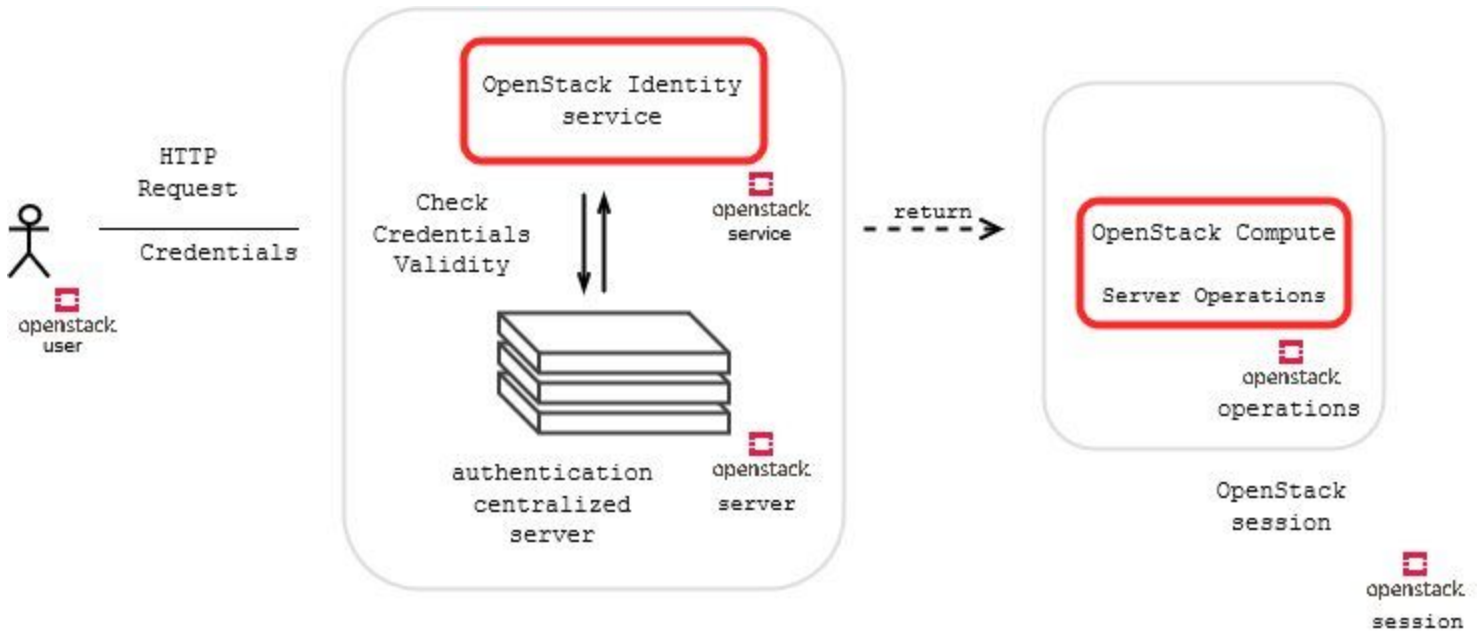

Figure 3. Compute API authentication

In *Figure 3* we can see the flow of the authentication process to connect to OpenStack Compute API.

At a minimum, the :class:`~openstack.connection.Connection` class needs to be created with a config or the parameters to build one.

While the overall system is very flexible, there are four main use cases
for different ways to create a **:class:`~openstack.connection.Connection`**.

- Using config settings and keyword arguments as described later in
  - *:ref:`openstack-config`*
- Using only keyword arguments passed to the constructor ignoring config files and environment variables.

- Using an existing authenticated `keystoneauth1.session.Session`, such as might exist inside of an OpenStack service operational context.
- Using an existing :class:`~openstack.config.cloud_region.CloudRegion`.

In our case, we use ***config settings and keyword arguments,*** which consists on:

If the application wants to avoid loading any settings from ``clouds.yaml`` or environment variables, use the :class:`~openstack.connection.Connection` constructor directly. As long as the ``cloud`` argument is omitted or ``None``, the :class:`~openstack.connection.Connection` constructor will not load settings from files or the environment.

So to establish a connection by the Compute API, we simply do the following:

```
from openstack import connection # used to establish a connection to openstack $
conn = connection.Connection(**auth_args)
```

auth_args is json file which contains the following data:

```
{
     "auth_url": "***",
     "project_name": "***",
     "project_id": "***",
     "username": "***",
     "password": "***",
     "user_domain_id": "***"
}
```

Once we get the connection instance, in this case it's conn, then we can get access to the different Compute API operations. To do so, we need to work with compute object of the connection instance.

```
compute_ = conn.compute
```

Now compute_ offers access to the server operations such as **`update_server`**(*server, **attrs*), **`delete_server`**(*server, ignore_missing=True, force=False*), `get_server`(*server*), ... etc.

Chapter 4

Spark- YARN - HDFS Cluster Deployment Automation

In this chapter, we'll describe the process of automation of a SPARK - YARN - HADOOP cluster on a given number of virtual machines. Also we will describe the architecture of a cluster to be deployed and make a simple implementation of Cluster Autoscaling concept.

Main Topics

- ❏ Study the different automation tools, advantages, disadvantages and reason about the one we chose for our project. We chose Ansible.
- ❏ Study the architecture of Ansible.
- ❏ Study the architecture of the cluster to be deployed and make an introduction for each of its components.
- ❏ Study different manners to design the Scale function in a deployed cluster.

4.1 Automation tools

One of our main goals in this project, is the automation of the deployment process of a cluster. But before choosing the right automation tool, we should ask ourselves some questions, like:

- ❏ Which environments do I need to support? What is my blend of servers and network devices?
- ❏ Who are my users? Is this for hardcore sysadmin types, the infosec team, developers?
- ❏ How much custom development am I willing to do?

My deep study will be on Ansible, so that we chose it to be our candidate in this project. So all the comparisons made in this chapter about the different automation tools, will be lead to Ansible.

Ansible

Ansible was the brainchild of Michael DeHaan, developed to automate tedious server-administration tasks across large environments. Michael was at RedHat's emerging technology group where he founded other projects like Cobbler, then went on to found Ansible after leaving RedHat (although, Ansible is now owned by RedHat). From Michael's blog on the foundations of Ansible, it is purpose is clear; [14]

> *"We wanted to create another very-democratic open source project at Red Hat, one that could have a wide variety of contributors and solve new problems. We thought back to busrpc. This project existed because it filled in gaps between Cobbler and Puppet. Cobbler could provision a system, and Puppet could lay down configuration files, but because Puppet was too declarative you couldn't use it to do things like reboot servers or do all the "ad hoc" tasks in between"*

Those ad-hoc tasks evolved into Ansible playbooks, the Ansible module ecosystem was born and boomed quickly.

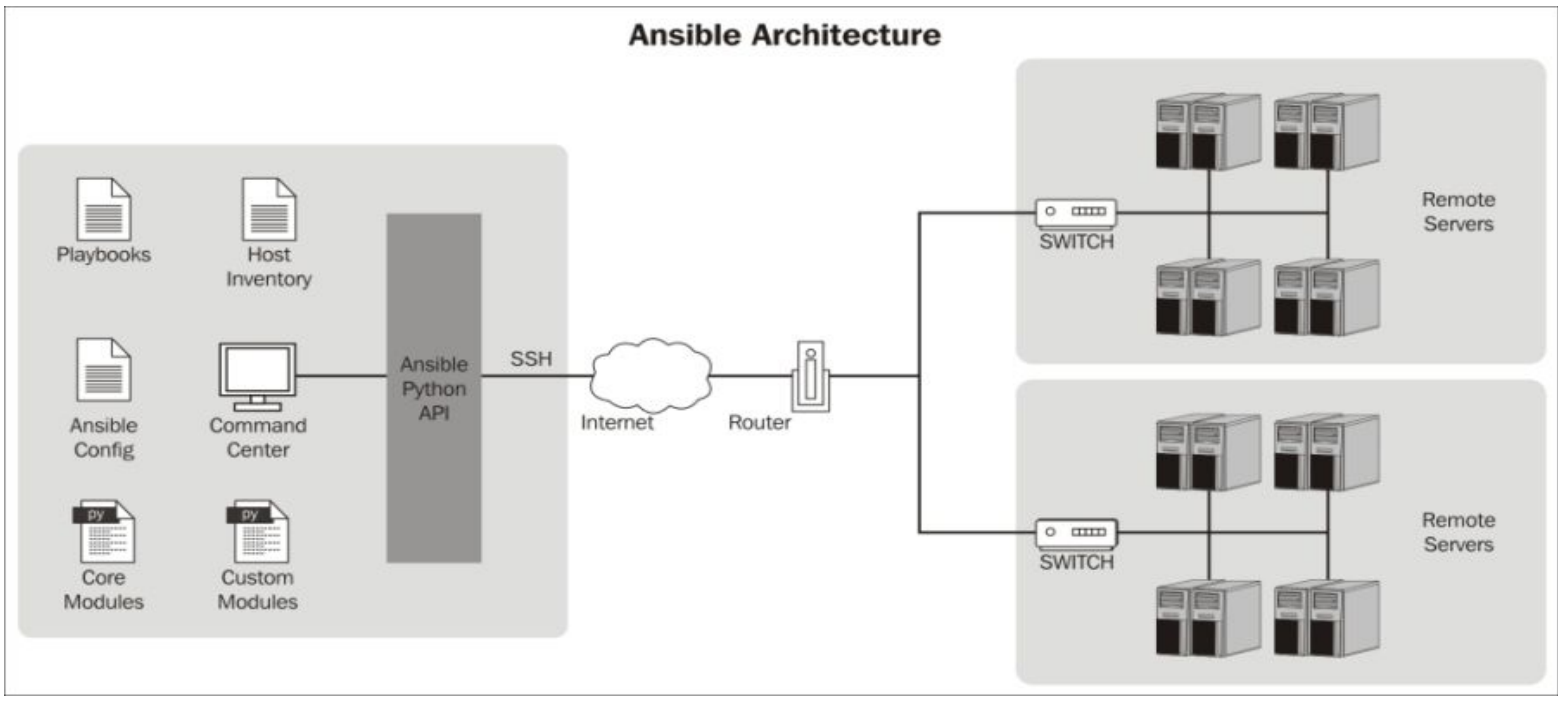

Figure 4. Ansible Architecture

4.1.1 Design

Ansible is simple, which is a major strength (and will become clear when looking at the other 2). There are no daemons, no databases, very minimal installation requirements. You just install Ansible on a Linux machine and off you go. Define the target servers either in a static file, grouped into meaningful sections, or use a dynamic host-discovery module like Amazon EC2 or OpenStack to find VMs based on an API call. Once you have an inventory you can build out host or group-specific variables which can be leveraged by your playbooks. These again are kept in static text files. [15]

Ansible will then connect to the host or group you choose and execute a playbook. The playbook is a sequence of Ansible modules that you want to execute on the remote hosts written in YAML.

When it connects to a remote host It's a bit like a *well planned military exercise*, get in, do the job and get out.
Ansible works by connecting to a server using SSH (or WS-Man/WinRM for Windows), copies the Python code over, executes it and then removes itself. [15]

4.1.2 Architecture

Ansible's architecture is straightforward, you have the application that runs on your machine and you have the tasks that run on the remote host, communicated to via SSH and files transferred by SCP/SFTP. Ansible doesn't have a "server-client" architecture like the other 2 products, so you parallelise task execution on your machine, but not scale across multiple servers (unless you use Tower).

When Ansible manages remote machines, it does not leave software installed or running on them, so there's no real question about how to upgrade Ansible when moving to a new version.

4.1.3 Extensibility

Ansible modules are *really* easy to develop, read the style guide if you later decide to try and merge your solution into the product's open-source repository instead of refactoring it again.

```
#!/usr/bin/python

import datetime
import json

date = str(datetime.datetime.now())
print json.dumps({
```

```
"time" : date
})
```

```
ansible/hacking/test-module -m ./timetest.py
```

You should see output that looks something like this:

```
{"time": "2018-08-14 22:13:48.539183"}
```

In modules you can define your own "**gather**" stage code to establish "**facts**" about the remote host which can be used by your or other modules. This could be something like looking at the files installed or configuration to determine how a service is setup.

Reasons to choose Ansible

Community Support

I've seen networking vendors specifically target and develop modules for **Ansible**, where as for the other platforms (with the exception of Brocade for StackStorm) they have been community contributed. Ansible certainly has the broadest breadth of support for networking platforms. Although, with the introduction of NAPALM and NSO into both StackStorm and Salt, this changes things as both support Arista, JunOS (Juniper), Cisco APIC-EM, NXOS et al.

Time to get started

Ansible's strength is the minimal amount of configuration to get going (basically none). It's popularity in the Networking space can be due to the simplicity and familiarity for network admins used to using something like a CLI to manage remote devices without needing to deploy any additional servers to run the software. If you have a lot of small, isolated sites (e.g. commercial branches) then you should consider whether your architecture would fall apart. My employer manages networks for some large supermarket chains and I would hesitate at having a centralised master when stores in rural areas can have unreliable connectivity.

Training

Unless you want to be the sole maintainer of this platform, you're going to need to educate some colleagues. Salt and Ansible both have detailed books published, StackStorm does not. Salt and (RedHat) Ansible offer training solutions, almost exclusively in the US, StackStorm **does not** (*yet*). Salt and Ansible have courses on PluralSight but they are really basic.

Skills

Ansible has, anecdotally (although I pay very close attention to this), got a good mind-share for network admins and DevOps engineers across the globe. You'd certainly find hiring

Ansible engineers a lot easier than Salt or StackStorm. But DevOps engineers are still as rare as hen's teeth so you're going to be paying top-dollar regardless of the platform.

4.2 Automation Scripting with Ansible

In this section we will describe the architecture of Ansible, used in the process of automation. More specifically we'll describe the main folders's structure used in our automation script.

Ansible components

Ansible uses the following components to work. Are the following:

A. Inventory file(s).
B. Group vars
C. Playbooks
D. Roles

If we look at our ansible folder structure, we can find the following tree.

```
[fedora@master ansible]$ cd /etc/ansible/
[fedora@master ansible]$ tree .

|-- ansible.cfg
|
|-- group_vars
|   `-- all
|    `-- var_file.yml
|
|-- hosts
|
|-- keys
|   |-- cluster-key.pem
|   `-- key-one.pem
|-- playbooks
|   |-- configure-Java-Hadoop-HDFS-Spark.yml
|
`-- roles
     |-- add-ssh-keys
     |   |-- tasks
     |   |   `-- main.yml
     |   `-- vars
     |    `-- main.yml
     |-- ansible-add-slave
     |   `-- tasks
     |    `-- main.yml
     |-- ansible-authorized-keys
     |   |-- tasks
     |   |   `-- main.yml
     |   `-- vars
     |    `-- main.yml
     |-- ansible-clean-HADOOP-processes
     |   |-- scripts
     |   |   |-- clear-hdfs-masters.sh
```

```
|   |   `-- clear-hdfs-slaves.sh
|   `-- tasks
|    `-- main.yml
|-- ansible-configure-firewall
|   |-- handlers
|   |   `-- main.yml
|   `-- tasks
|    `-- main.yml
|-- ansible-configure-hosts-file
|   `-- tasks
|    `-- main.yml
|-- ansible-configure-install-Hadoop
|   |-- tasks
|   |   |-- main.yml
|   |   `-- main.yml.copy
|   |-- templates
|   |   |-- template-hdfs-site-masters-xml.j2
|   |   |-- template-hdfs-site-slaves-xml.j2
|   |   |-- template-mapred-site-xml.j2
|   |   `-- template-yarn-site-xml.j2
|   `-- vars
|    `-- main.yml
|-- ansible-disable-selinux
|   |-- tasks
|   |   `-- main.yml
|   `-- vars
|    `-- main.yml
|-- ansible-install-Scala
|   |-- tasks
|   |   `-- main.yml
|   `-- vars
|    `-- main.yml
|-- ansible-install-Spark
|   |-- tasks
|   |   `-- main.yml
|   `-- vars
|    `-- main.yml
|-- ansible-install-java
    |-- tasks
    |   `-- main.yml
    `-- vars
        `-- main.yml

42 directories, 42 files
```

A. Inventory file(s)

Ansible works against multiple systems in your infrastructure at the same time. It does this by selecting portions of systems listed in Ansible's inventory file, which defaults to being saved in the location `/etc/ansible/hosts`. You can specify a different inventory file using the `-i <path>` option on the command line.

Not only is this inventory configurable, but you can also use multiple inventory files at the same time (explained below) and also pull inventory from dynamic or cloud sources, as described in Dynamic Inventory. Later we will make an experiment with a Dynamic Inventory file.

Our Inventory file, looks like follows.

```
[all:vars]
ansible_ssh_private_key_file=/etc/ansible/keys/key-one.pem
ansible_python_interpreter=/usr/bin/python3

[webservers]
192.168.0.94
192.168.0.74
192.168.0.84
[masters]
192.168.0.94

[slaves]
192.168.0.74
192.168.0.84

[storage]
192.168.0.74
192.168.0.84

[processing]
192.168.0.74
192.168.0.84

[ssh-authority]
192.168.0.94
192.168.0.74
192.168.0.84
localhost
```

We can see different sections in the inventory file. Those sections are called **Groups**.

- **`[all:vars]`**
  - Contains all the variables which have the same value for all groups, so we have not to repeat their definition in each group. These variables are configuration variables, so they are used as
    - `ansible_ssh_private_key_file=/etc/ansible/keys/key-one.pem`
    - `ansible_python_interpreter=/usr/bin/python3`
- **`[webservers]`**
  - Normally it's called on the group which contains all the hosts of the cluster. In our experiment, the cluster is formed by 3 hosts.
    - `192.168.0.94`
    - `192.168.0.74`
    - `192.168.0.84`
- **`[masters]`**
  - Contains the masters hosts.
    - `192.168.0.94`
- **`[slaves]`**
  - Contains the slaves hosts

- 192.168.0.74
- 192.168.0.84

- **[storage]**[1]
  - Contains the hosts used for the storage. Run HDFS.

    - 192.168.0.94
    - 192.168.0.74
    - 192.168.0.84
- **[processing]**[1]
  - Contains the hosts used for processing. Run Spark.

    - 192.168.0.94
    - 192.168.0.74
    - 192.168.0.84
- **[ssh-authority]**
  - Contains the hosts to be authenticated between each others using ssh.
    - 192.168.0.94
    - 192.168.0.74
    - 192.168.0.84
    - localhost

B. Groups vars

Group variables as the name suggests, applies to all of the hosts which reside inside a group. There is also an **"all"** group variables file which specifies variables for all groups. To configure group variables simply create a YAML file which matches the name of the group.

```
|-- group_vars
|   `-- all
|    `-- var_file.yml
```

This is another way to define variables for the declared groups of the inventory file, described in the section "inventory file(s)". These variables are application-level variables. That means, they're used in the the own execution of each task of the script. For example we can declare a list of the hosts to iterate over, just like follows.

```
hosts:
  - 192.168.0.94 master
  - 192.168.0.74 slave-1
  - 192.168.0.84 slave-2
```

[1] In later sections we will discuss the processing-storage architecture.

Our var_file.yml contained in group_vars/all “corresponds to all the groups”, contains:

```
hosts:
  - 192.168.0.94 master
  - 192.168.0.74 slave-1
  - 192.168.0.84 slave-2

masters:
  - 192.168.0.94 master

storage-nodes:
  - 192.168.0.74 slave-1
  - 192.168.0.84 slave-2
```

```
processing-nodes:
  - 192.168.0.74 slave-1
  - 192.168.0.84 slave-2
app_user: fedora
key_file_path: /etc/ansible/keys/key-one.pem
```

C. Playbooks

Playbooks are Ansible’s configuration, deployment, and orchestration language. They can describe a policy you want your remote systems to enforce, or a set of steps in a general IT process.

If Ansible modules are the tools in your workshop, playbooks are your instruction manuals, and your inventory of hosts are your raw material.

At a basic level, playbooks can be used to manage configurations of and deployments to remote machines. At a more advanced level, they can sequence multi-tier rollouts involving rolling updates, and can delegate actions to other hosts, interacting with monitoring servers and load balancers along the way.

Playbooks are designed to be human-readable and are developed in a basic text language. There are multiple ways to organize playbooks and the files they include, and we’ll offer up some suggestions on that and making the most out of Ansible.

```
|-- playbooks
|   |-- configure-Java-Hadoop-HDFS-Spark.yml
```

Our playbook looks like.

```
---
- hosts: ssh-authority
  become: yes
  roles:
     - ansible-authorized-keys

- hosts: webservers
  become: yes
  roles:
     - ansible-configure-hosts-file
     - ansible-configure-firewall
     - ansible-disable-selinux
     - ansible-install-java

- hosts: masters:slaves:storage
  become: yes
  roles:
     - ansible-configure-install-Hadoop
     - ansible-clean-HADOOP-processes

- hosts: masters:processing
  become: yes
  roles:
     - ansible-install-Spark
     - ansible-install-Scala
```

On each entry we have the following elements:

- ❏ hosts: The hosts groups on which the following roles will be executed.
- ❏ “become: yes” : The user to run the task, is root.
- ❏ roles: each role contains a list of tasks.

D. Roles

Roles are ways of automatically loading certain vars_files, tasks, and handlers based on a known file structure. Grouping content by roles also allows easy sharing of roles with other users.

Roles are just automation around ‘include’ directives as described above, and really don’t contain much additional magic beyond some improvements to search path handling for referenced files. However, that can be a big thing!

```
<role-folder-name>/
|-- defaults
|-- handlers
|-- meta
|-- tasks
|-- vars
`-- templates
```

❏ *defaults*

```
<role-folder-name>/
|-- defaults
```

Role Default Variables

Role default variables allow you to set default variables for included or dependent roles (see below). To create defaults, simply add a defaults/main.yml file in your role directory. These variables will have the lowest priority of any variables available, and can be easily overridden by any other variable, including inventory variables

❏ *Handlers*

Handlers are similar to tasks except that a handler will be executed only when it is called by an event. For example, a handler that will start the httpd service after a task installed httpd. The handler is called by the [notify] directive.

Important: the name of the notify directive and the handler must be the same.

```
|-- ansible-configure-firewall
|   |-- handlers
|   |    `-- main.yml
|    `-- tasks
|        `-- main.yml
```

tasks/main.yml:

```
- name: Reloading firewalld to apply the changes ...
  command: firewall-cmd --reload
  notify:
   -  restart firewalld
  become: yes
```

handlers/main.yml

```
- name: restart firewalld
  service: name=firewalld state=restarted
```

❏ *meta*

Role Dependencies

Role dependencies allow you to automatically pull in other roles when using a role. Role dependencies are stored in the meta/main.yml file contained within the role directory. This file should contain a list of roles and parameters to insert before the specified role, such as the following in an example roles/<role-name>/meta/main.yml:

```
---
dependencies:
  - { role: common, some_parameter: 3 }
  - { role: apache, apache_port: 80 }
  - { role: postgres, dbname: blarg, other_parameter: 12 }
```

❑ *tasks*

Tasks are actions carried out by playbooks. One example of deleting Hadoop folder is:

```
- name: deleting old versions of /opt/hadoop
  file:
     state: absent
     path: /opt/hadoop
```

❑ *vars*

As the name suggests, you can include custom-made variables in your playbooks. Variables defined under /roles/<role-name>/vars/main.yml represent the same variables described in **Groups vars.** These variables are application-level variables, but in this case they're only associated to the role where they're defined.

```
|-- ansible-install-Spark
|   |-- tasks
|   |   `-- main.yml
|   `-- vars
|    `-- main.yml
```

../vars/main.yml

```
download_folder: /usr/local
spark_package_file: /opt/required-packages/spark-2.3.0-bin-hadoop2.7.tgz
spark_folder_path_name_old: "{{ download_folder }}/spark-2.3.0-bin-hadoop2.7"
spark_folder_name_new: spark
```

❑ *templates*

Templates files are based on Python's Jinja2 template engine and have a .j2 extension. You can, if you need, place contents of your index.html file into a template file. But the real power of these files comes when you use variables. You can use Ansible's [facts] and even call custom variables in these template files.

```
|-- ansible-configure-install-Hadoop
|   |-- tasks
|   |   |-- main.yml
|   |   `-- main.yml.copy
|   |-- templates
|   |   |-- template-hdfs-site-masters-xml.j2
|   |   |-- template-hdfs-site-slaves-xml.j2
|   |   |-- template-mapred-site-xml.j2
|   |   `-- template-yarn-site-xml.j2
```

template-hdfs-site-masters-xml.j2

```
<configuration>
<property>
  <name>{{ name_property_1 }}</name>
  <value>{{ value_property_1 }}</value>
</property>
<property>
```

```
  <name>{{ name_property_2 }}</name>
  <value>{{ value_property_2 }}</value>
</property>
<property>
   <name>{{ name_property_3 }}</name>
   <value>{{ value_property_3 }}</value>
</property>
</configuration>
```

variables for /opt/hadoop/etc/hadoop/hdfs-site.xml

```
name_property_1: dfs.replication
value_property_1: 3
name_property_2: dfs.permissions
value_property_2: false
name_property_3: dfs.datanode.data.dir
value_property_3: "/home/{{ app_user }}/datanode"
name_property_4: dfs.namenode.data.dir
value_property_4: "/home/{{ app_user }}/namenode"
```

Chapter 5

Distributed Processing Architecture

This chapter, describes the architecture we chose for the distributed data processing approach. This architecture is formed by two fundamentals elements, storage and processing. We implement these two concepts, using SPARK and HADOOP analytics and storage engines.

5.1 The Elements of the Project Architecture

Our main approach for this project was to focus on the implementation of a distributed data processing architecture. We should consider three main concepts, we need to implement the mention architecture [17] :

- ❏ Storage.
- ❏ Resources and Applications Management.
- ❏ and Processing.

5.1.1 Storage: HDFS

For the Storage, we decided to use Hadoop Distributed File System “HDFS” which is a distributed file system designed to run on commodity hardware. It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data. HDFS was originally built as infrastructure for the Apache Nutch web search engine project. HDFS is now an Apache Hadoop subproject. [18]

a. NameNode and DataNodes

HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system’s clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode. [18]

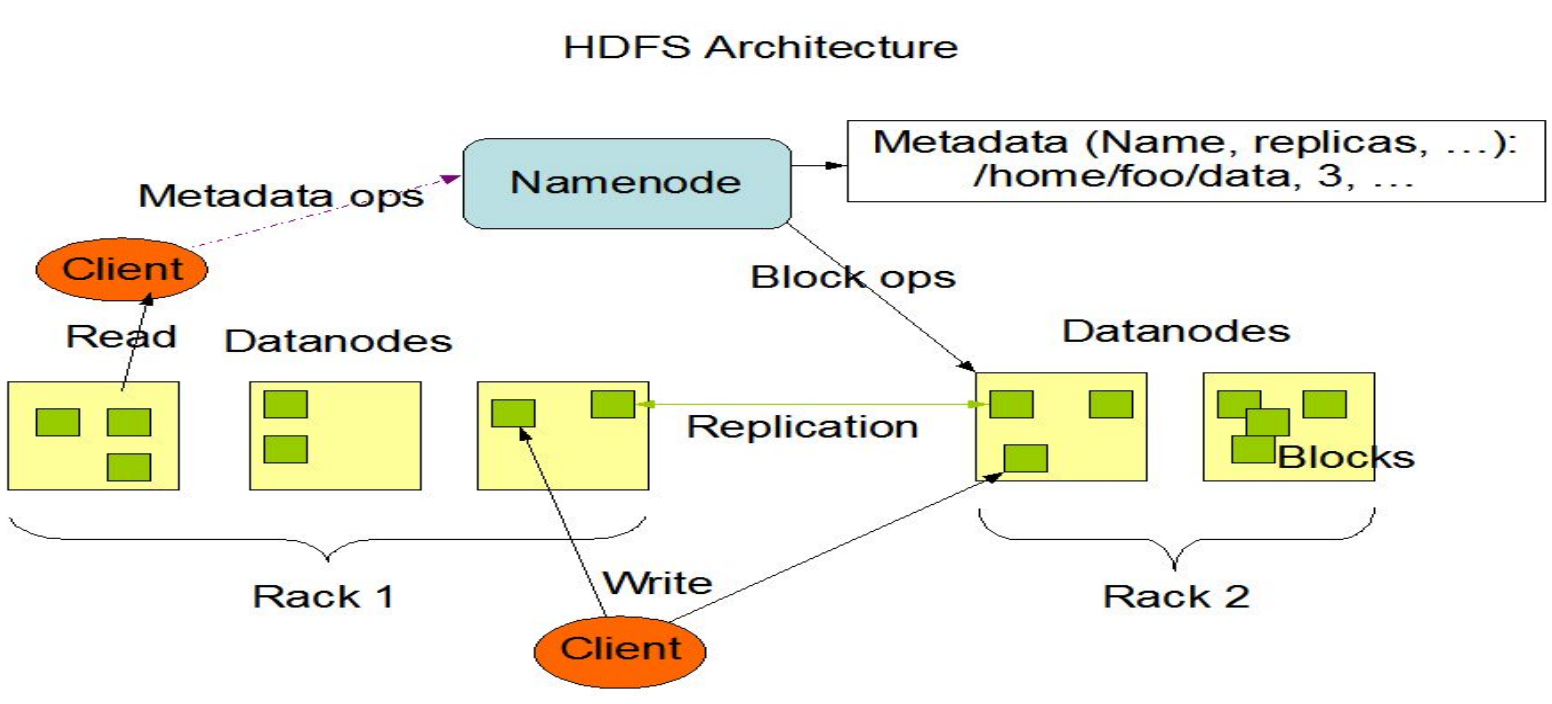

Figure 5. HDFS Architecture

The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS). HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines. A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

The existence of a single NameNode in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata. The system is designed in such a way that user data never flows through the NameNode. [18]

b. The File System Namespace

HDFS supports a traditional hierarchical file organization. A user or an application can create directories and store files inside these directories. The file system namespace hierarchy is similar to most other existing file systems; one can create and remove files, move a file from one directory to another, or rename a file. HDFS does not yet implement user quotas. HDFS does not support hard links or soft links. However, the HDFS architecture does not preclude implementing these features. [18]

The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the NameNode. [18]

c. Data Replication

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode. [18]

d. Replica Selection

To minimize global bandwidth consumption and read latency, HDFS tries to satisfy a read request from a replica that is closest to the reader. If there exists a replica on the same rack

as the reader node, then that replica is preferred to satisfy the read request. If angg/ HDFS cluster spans multiple data centers, then a replica that is resident in the local data center is preferred over any remote replica. [18]

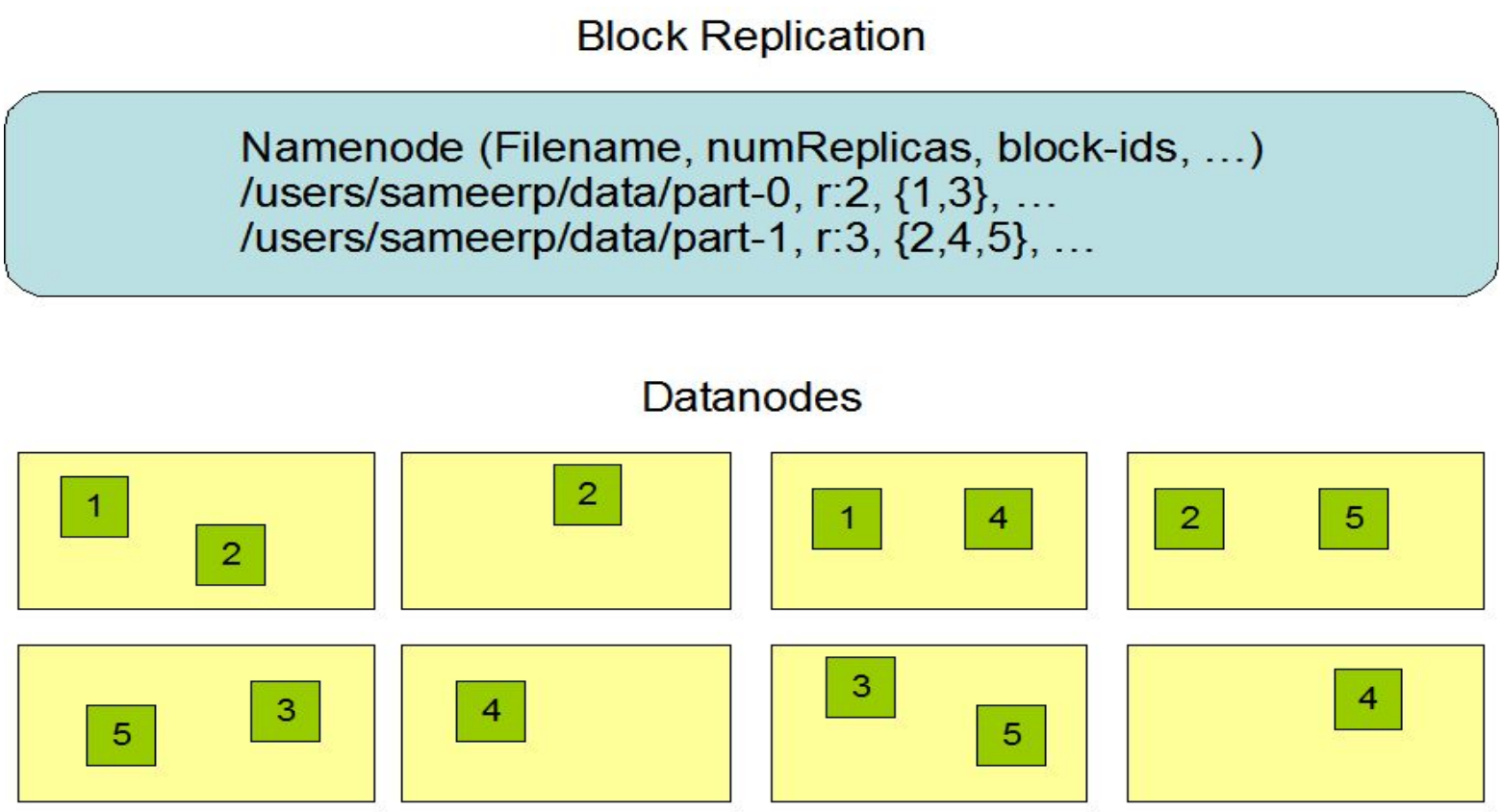

Figure 6. Block Replication in HDFS

e. Safemode

On startup, the NameNode enters a special state called Safemode. Replication of data blocks does not occur when the NameNode is in the Safemode state. The NameNode receives Heartbeat and Blockreport messages from the DataNodes. A Blockreport contains the list of data blocks that a DataNode is hosting. Each block has a specified minimum number of replicas. A block is considered safely replicated when the minimum number of replicas of that data block has checked in with the NameNode. After a configurable percentage of safely replicated data blocks checks in with the NameNode (plus an additional 30 seconds), the NameNode exits the Safemode state. It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas. The NameNode then replicates these blocks to other DataNodes. [18]

5.1.2 Resource Management: YARN

Hadoop YARN is a cluster management technology that is part of the Hadoop 2.0. YARN full form is Yet Another Resource Negotiator. It was originally used as a redesigned resource manager but now YARN is part of a large scale distributed operating system that is used for the big data applications. [19]

The fundamental idea of YARN is to split up the functionalities of resource management and job scheduling/monitoring into separate daemons. The idea is to have a global ResourceManager (*RM*) and per-application ApplicationMaster (*AM*). An application is either a single job or a DAG of jobs.

The ResourceManager and the NodeManager form the data-computation framework. The ResourceManager is the ultimate authority that arbitrates resources among all the

applications in the system. The NodeManager is the per-machine framework agent who is responsible for containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager/Scheduler.

The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks. [19]

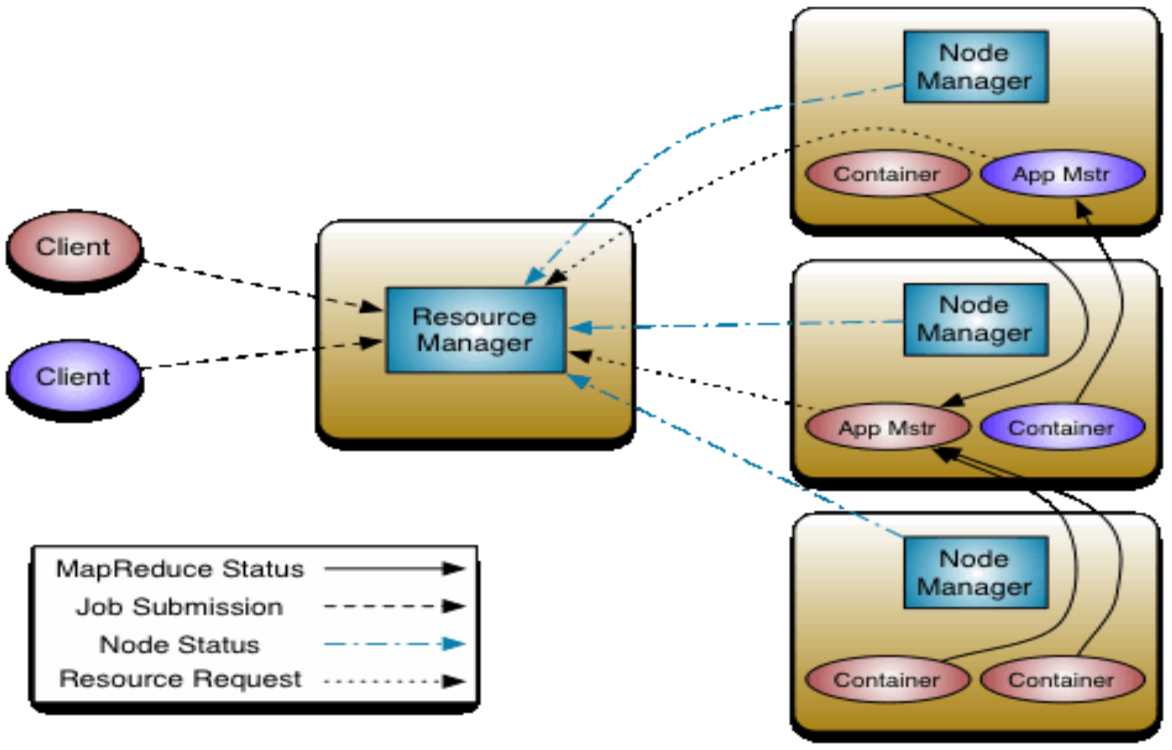

Figure 7. YARN Architecture

YARN is part of the Hadoop architectural setup and is a core Hadoop project. The architecture of YARN lets you process data using multiple processing engines using real-time streaming, interactive SQL, batch processing, handling of data stored in single platform and working with analytics in a completely different manner. YARN can be called as the basis of the next generation of the Hadoop ecosystem that is ensuring the forward-thinking organizations are realizing the modern data architecture. [19]

The `ResourceManager` has two main components:

- Scheduler
- ApplicationsManager.

The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is pure scheduler in the sense that it performs no monitoring or tracking of status for the application. Also, it offers no guarantees about restarting failed tasks either due to application failure or hardware failures. The Scheduler performs its scheduling function based on the resource requirements of the applications; it does so based on the abstract notion of a resource *Container* which incorporates elements such as memory, cpu, disk, network etc. [19]

The Scheduler has a pluggable policy which is responsible for partitioning the cluster resources among the various queues, applications etc. The current schedulers such as the CapacityScheduler and the FairScheduler would be some examples of plug-ins. [19]

The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster and provides the service for restarting the ApplicationMaster container on failure. The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress. [19]

5.1.3 Processing: APACHE SPARK

Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. [20]

Apache Spark, also is a fast, in-memory data processing engine with elegant and expressive development APIs to allow data workers to efficiently execute streaming, machine learning or SQL workloads that require fast iterative access to datasets. With Spark running on Apache Hadoop YARN, developers everywhere can now create applications to exploit Spark's power, derive insights, and enrich their data science workloads within a single, shared dataset in Hadoop. [20]

The Hadoop YARN-based architecture provides the foundation that enables Spark and other applications to share a common cluster and dataset while ensuring consistent levels of service and response.

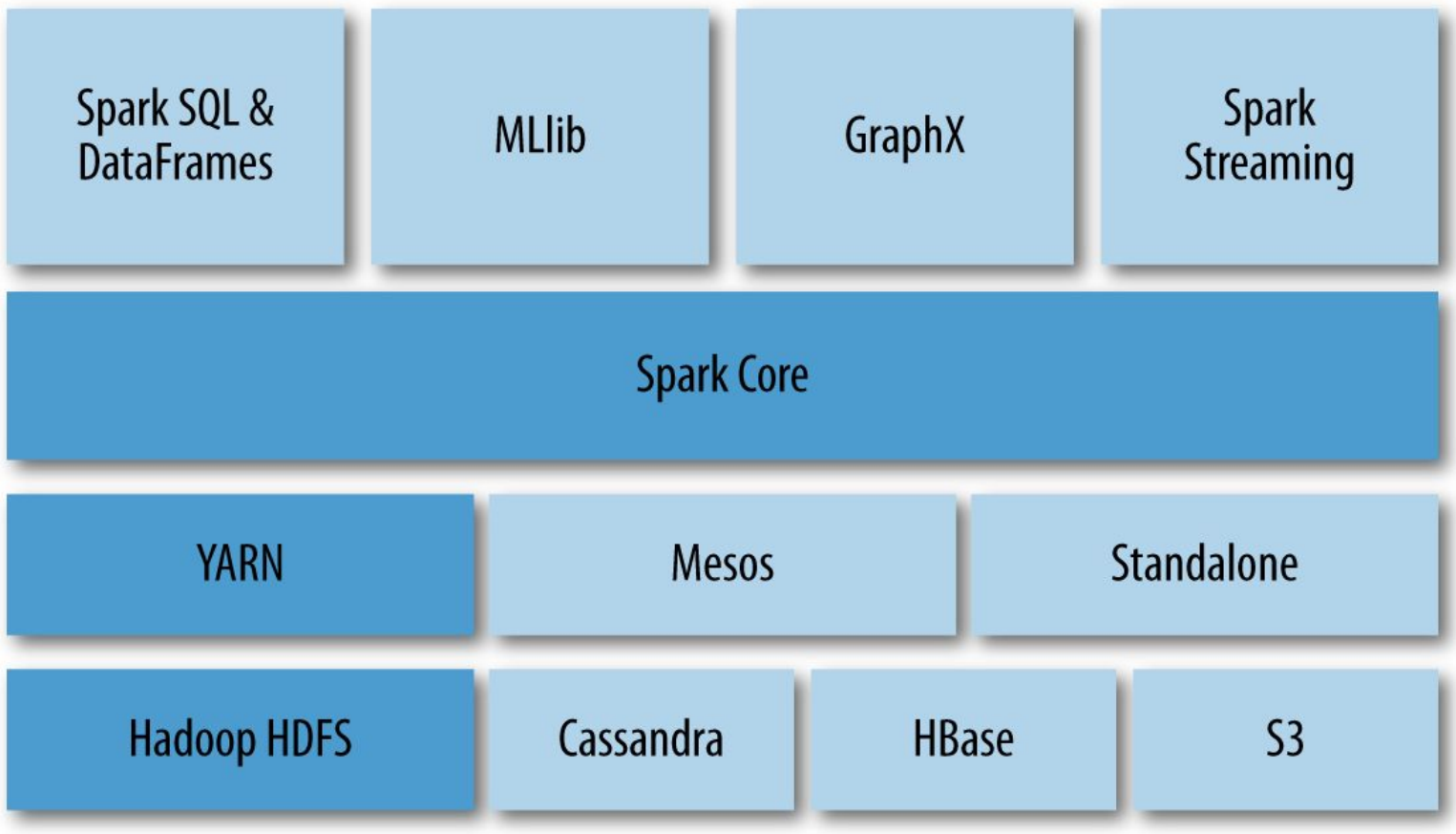

Figure 8. SPARK - YARN - HDFS Architecture

5.1.4 Apache Spark Resource Management and YARN App Models

The most popular Apache YARN application after MapReduce itself is Apache Spark. At Cloudera, we have worked hard to stabilize Spark-on-YARN (SPARK-1101), and CDH 5.0.0 added support for Spark on YARN clusters.

In this post, you'll learn about the differences between the Spark and MapReduce architectures, why you should care, and how they run on the YARN cluster ResourceManager. [20]

5.2 Applications

In MapReduce, the highest-level unit of computation is a job. The system loads the data, applies a map function, shuffles it, applies a reduce function, and writes it back out to persistent storage. Spark has a similar job concept (although a job can consist of more stages than just a single map and reduce), but it also has a higher-level construct called an "application," which can run multiple jobs, in sequence or in parallel. [20]

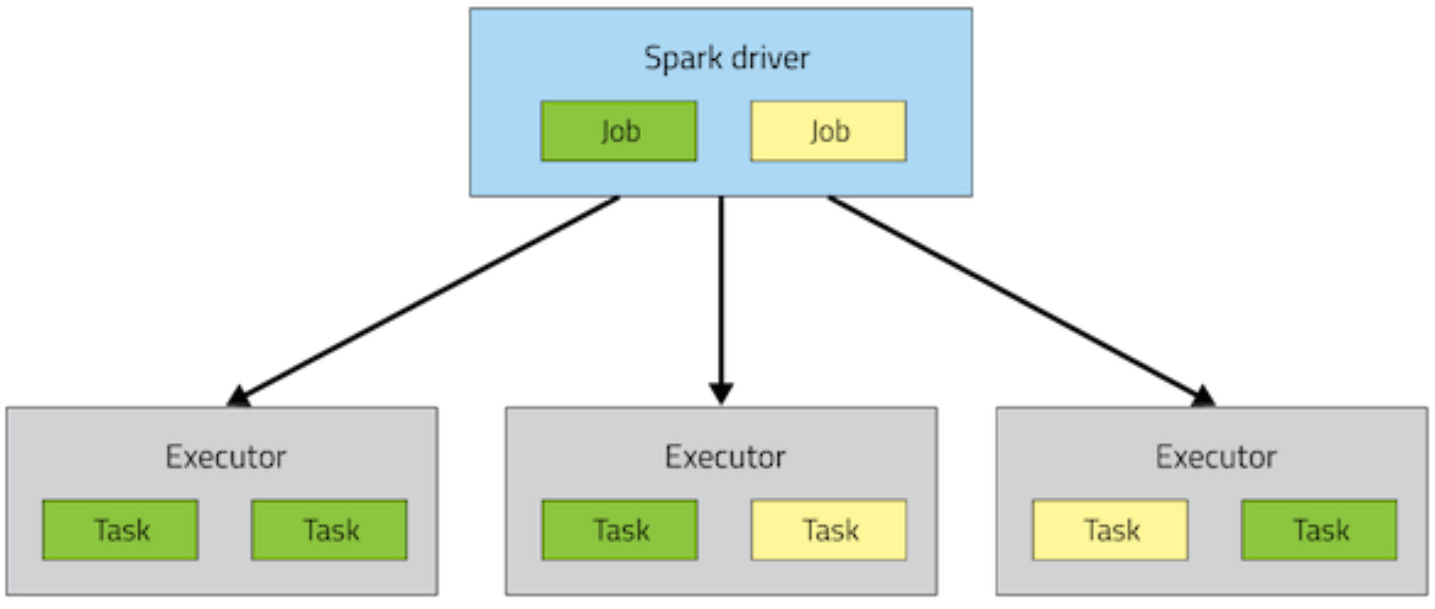

Figure 9. Spark application architecture

Spark applications consist of a driver process and a set of executor processes. The driver process runs your main() function, sits on a node in the cluster, and is responsible for three things: maintaining information about the Spark Application; responding to a user's program or input; and analyzing, distributing, and scheduling work across the executors (defined momentarily). The driver process is absolutely essential – it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

The executors are responsible for actually executing the work that the driver assigns them. This means, each executor is responsible for only two things: executing code assigned to it by the driver and reporting the state of the computation, on that executor, back to the driver node. [20]

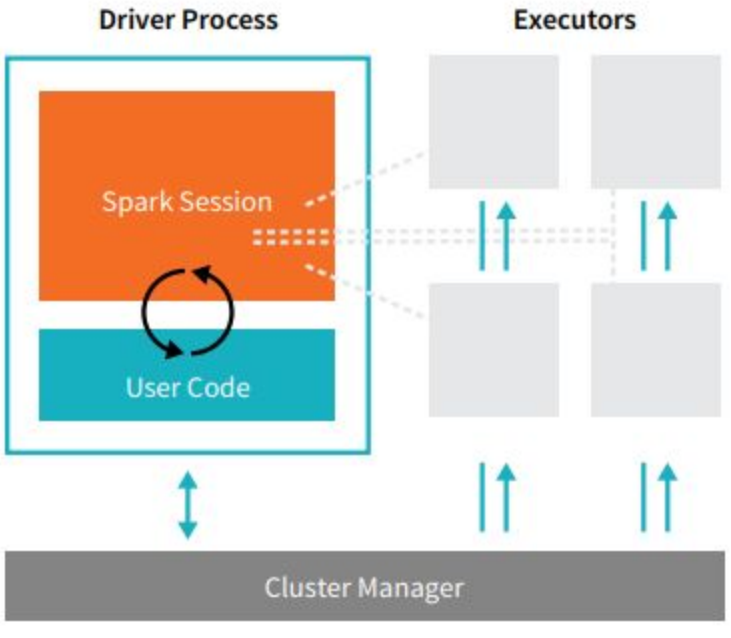

Figure 10. Spark application internal structure

The cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of several core cluster managers: Spark's standalone cluster manager, YARN, or Mesos. This means that there can be multiple Spark Applications running on a cluster at the same time. We will talk more in depth about cluster managers in Part IV: Production Applications of this book. [21]

In the previous illustration we see on the left, our driver and on the right the four executors on the right. In this diagram, we removed the concept of cluster nodes. The user can specify how many executors should fall on each node through configurations. [21]

Why Run on YARN?

Using YARN as Spark's cluster manager confers a few benefits over Spark standalone and Mesos:

- YARN allows you to dynamically share and centrally configure the same pool of cluster resources between all frameworks that run on YARN. You can throw your entire cluster at a MapReduce job, then use some of it on an Impala query and the rest on Spark application, without any changes in configuration.
- You can take advantage of all the features of YARN schedulers for categorizing, isolating, and prioritizing workloads.
- Spark standalone mode requires each application to run an executor on every node in the cluster, whereas with YARN, you choose the number of executors to use.
- Finally, YARN is the only cluster manager for Spark that supports security. With YARN, Spark can run against Kerberized Hadoop clusters and uses secure authentication between its processes.

Running on YARN

When running Spark on YARN, each Spark executor runs as a YARN container. Where MapReduce schedules a container and fires up a JVM for each task, Spark hosts multiple tasks within the same container. This approach enables several orders of magnitude faster task startup time.

Spark supports two modes for running on YARN, “yarn-cluster” mode and “yarn-client” mode. Broadly, yarn-cluster mode makes sense for production jobs, while yarn-client mode makes sense for interactive and debugging uses where you want to see your application’s output immediately.

Understanding the difference requires an understanding of YARN’s *Application Master* concept. In YARN, each application instance has an Application Master process, which is the first container started for that application. The application is responsible for requesting resources from the ResourceManager, and, when allocated them, telling NodeManagers to start containers on its behalf. Application Masters obviate the need for an active client — the process starting the application can go away and coordination continues from a process managed by YARN running on the cluster.

In yarn-cluster mode, the driver runs in the Application Master. This means that the same process is responsible for both driving the application and requesting resources from YARN, and this process runs inside a YARN container. The client that starts the app doesn’t need to stick around for its entire lifetime. [22]

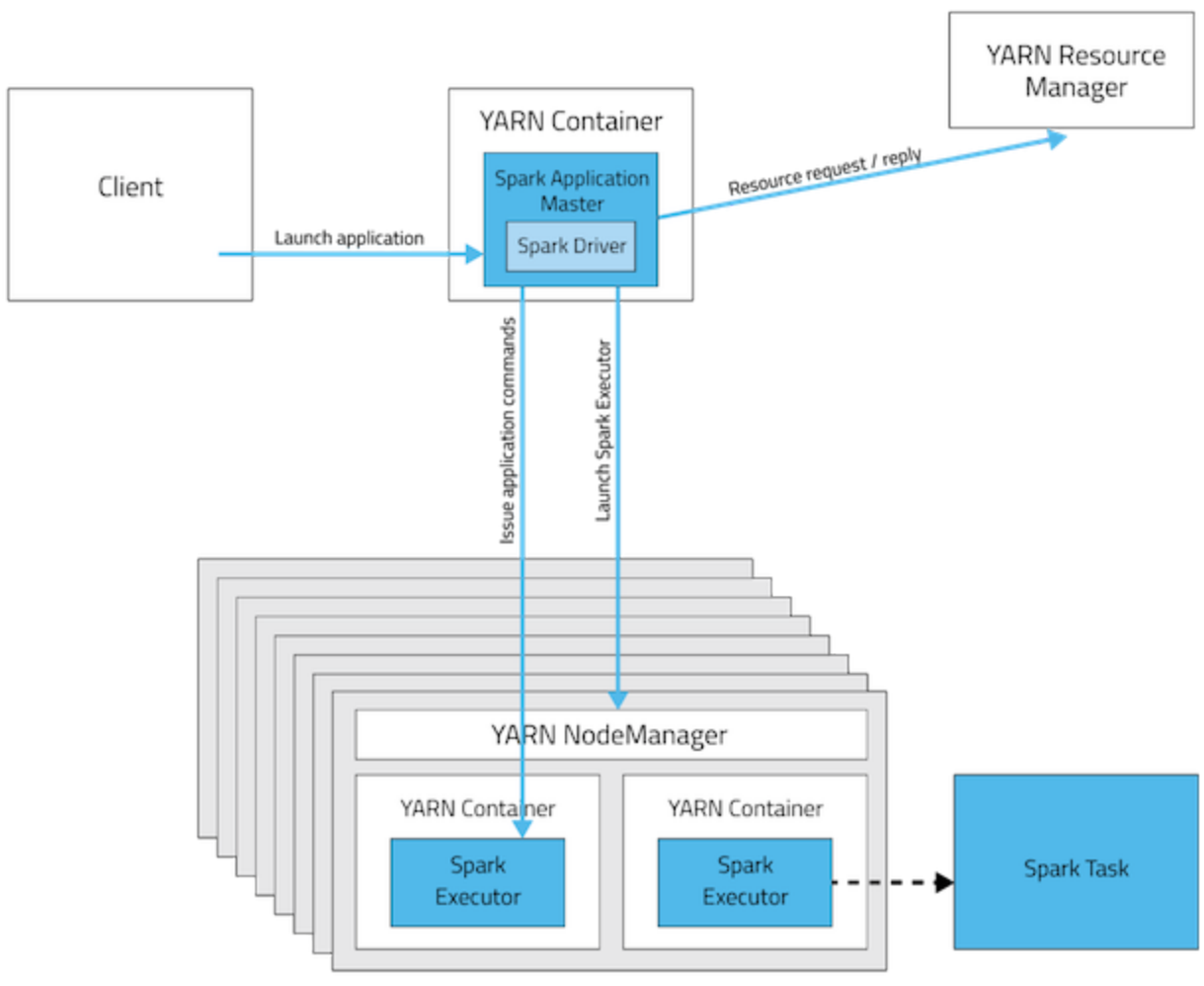

Figure 11. Yarn-cluster mode

The yarn-cluster mode, however, is not well suited to using Spark interactively. Spark applications that require user input, like spark-shell and PySpark, need the Spark driver to run inside the client process that initiates the Spark application. In yarn-client mode, the Application Master is merely present to request executor containers from YARN. The client communicates with those containers to schedule work after they start [22]:

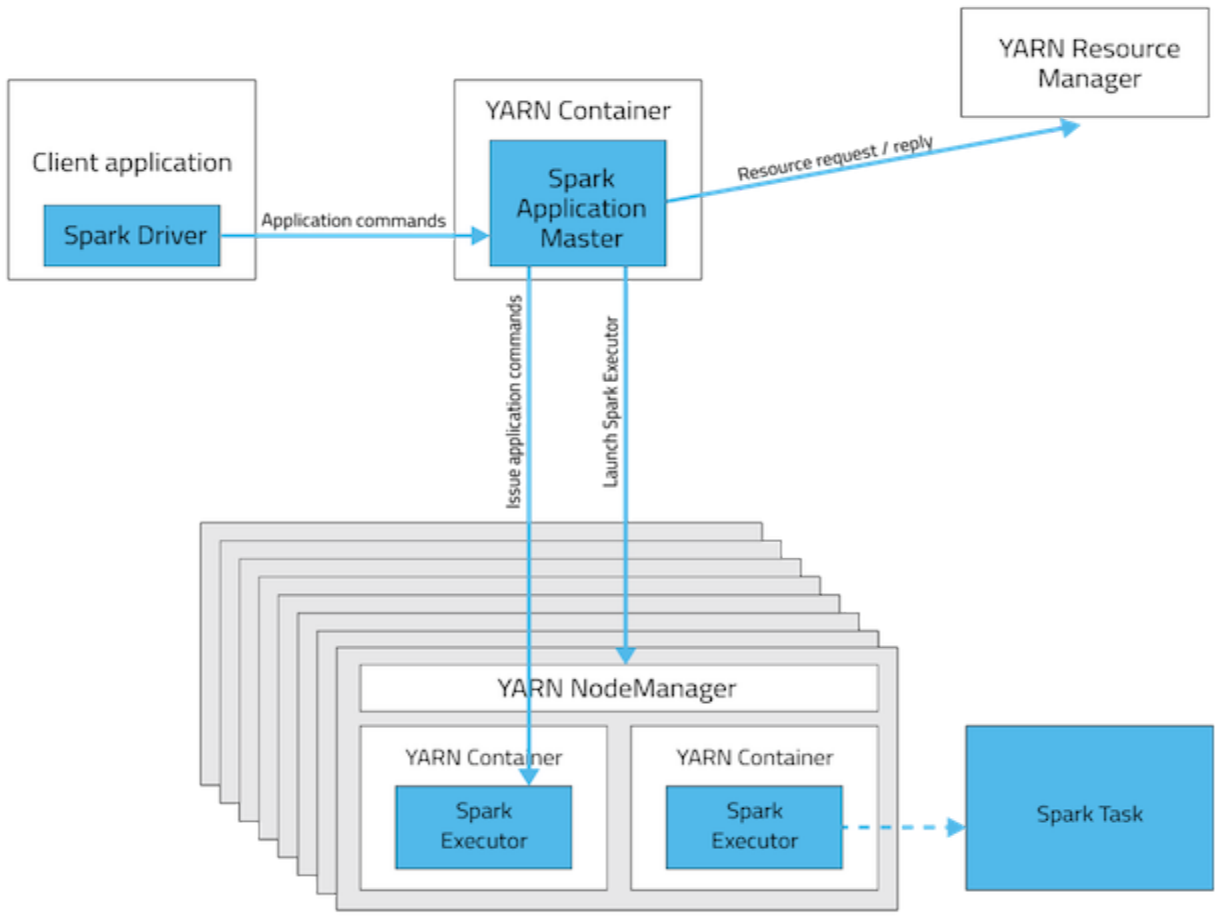

Figure 12. Yarn-client mode

Chapter 6

Service Catalog

In this chapter we describe the service catalog, which initially offers basic data processing scripts designed to be executed in a distributed storage-processing architecture such as the one we described in Chapter 5, based on Spark, Yarn and HDFS.

Service Catalog Components

A service catalog is required in order to offer different services for the users of the final product. Those services are related to Big Data, such as processing a huge data set with a determined algorithm. In order to do so, we need to implement different functionalities.

Each functionality can be included into components, these components are:

- ❏ Storage
- ❏ Processing

Practically, to implement these components we should return to the data processing architecture we mentioned in Chapter 5. So we would use HDFS, YARN and Spark. This process takes the following steps as activity flow:

1. Start Spark session.
2. Load the required dataset.
3. Run the corresponding data processing script.

1. Spark Session

SparkSession is the entry point to Spark SQL. It is one of the very first objects you create while developing a Spark SQL application.

Generally, a session is an interaction between two or more entities. In computer parlance, its usage is prominent in the realm of networked computers on the internet. First with TCP session, then with login session, followed by HTTP and user session, so no surprise that we now have *SparkSession*, introduced in Apache Spark 2.0.

Beyond a time-bounded interaction, *SparkSession* provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with DataFrame and Dataset APIs. Most importantly, it curbs the number of concepts and constructs a developer has to juggle while interacting with Spark.

1.1 Exploring SparkSession's Unified Functionality

First, we will examine a Spark application, SparkSessionZipsExample, that reads zip codes from a JSON file and do some analytics using DataFrames APIs, followed by issuing Spark SQL queries, without accessing SparkContext, SQLContext or HiveContext.

1.2 Creating a SparkSession

In previous versions of Spark, you had to create a SparkConf and SparkContext to interact with Spark, as shown here:

```
//set up the spark configuration and create contexts
val sparkConf = new
SparkConf().setAppName("SparkSessionZipsExample").setMaster("local")
// your handle to SparkContext to access other context like SQLContext
val sc = new SparkContext(sparkConf).set("spark.some.config.option", "some-value")
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

Whereas in Spark 2.0 the same effects can be achieved through SparkSession, without expliciting creating SparkConf, SparkContext or SQLContext, as they're encapsulated within the SparkSession. Using a builder design pattern, it instantiates a SparkSession object if one does not already exist, along with its associated underlying contexts.

```
// Create a SparkSession. No need to create SparkContext
// You automatically get it as part of the SparkSession
val warehouseLocation = "file:${system:user.dir}/spark-warehouse"
val spark = SparkSession
   .builder()
   .appName("SparkSessionZipsExample")
   .config("spark.sql.warehouse.dir", warehouseLocation)
   .enableHiveSupport()
   .getOrCreate()
```

At this point you can use the *spark* variable as your instance object to access its public methods and instances for the duration of your Spark job.

In our case we define a function which creates a spark session, called `start_spark_session`

```
/**

* start_spark_session, a function used to start a SparkSession and give us the
* possibility to work with spark functionalities.
*
* Better Understainding for How to Intialize Spark
* ------------------------------------------------
*
* * In order to start a Spark Application, we should follow
* * ------------------------------------------------------
*
* * * 1) Start a SparkSession and then get its SparkContext:
* * * ----------------------------------------------------
* * * A SparkSession give a lot more functionalities than a normal SparkContext, like
      wokring with Dataframes, ....

* * * SparkSessions provide all the functionalities provided by a SparkContext
*
* Or
*
* * * 2) Create a SparkConf and then create a SparkContext:
* * * ----------------------------------------------------
* * * create a SparkConf object,  hat contains information about your application, this
      called, conf.

* * * create a SparkContext object which tells Spark how to access a cluster, using
      "conf".
*
* */

def start_spark_session(): Unit = {
this.mySparkSession = SparkSession
      .builder() // Create a SparkSession.Builder for constructing a SparkSession.
      .appName(this.AppName) // Specify the application name
      .master(this.Master_IP) // Specify the master
      .getOrCreate() // Get an exisiting session or Create a new one

// Get the SparkContext of mySparkSession
this.mySparkContext = mySparkSession.sparkContext
}
```

2. Loading Datasets

In order to load a dataset to be processed, we first load it into HDFS, so it can be loaded in distributed-storage architecture. A distributed-storage file system is required to get the improved results with a distributed-processing engine, such as Spark.

In order to do so, we implemented a case class, called HDFS_Coordinator, which implements the following functionalities:

- ❏ Write to HDFS
- ❏ Save an existing file in HDFS
- ❏ Remove data from HDFS
- ❏ Get a file from HDFS
- ❏ Create a folder in HDFS

In this case we make use of Hadoop File System API in Scala, using the libraries:

- ❏ org.apache.hadoop.conf.Configuration
- ❏ org.apache.hadoop.fs.{FileSystem, Path}

```
case class HDFS_Coordinator(){
     private val conf = new Configuration()
     /**
     * write, a function used to write a file to hdfs.
     *
     * Example of Execution
     * --------------------
     * @example Hdfs.write("hdfs://0.0.0.0:9000/user/hadoop/", "test.txt", "Hello
       World".getBytes)
     *
     * @param uri, a String, represents the default filesystem path. Normally it comes as
       follows:
     *           hostname:port/path
     *           hostname: yarn.resourcemanager.hostname.
     *           port: port that the resource manager runs on.
     *           path: path where the file to be written.
     *
     * @param filePath, a String, represents the path of the file to store.
     * @param data, Array of Bytes, represents the content to write in the file to store.
     * @param hadoop_user, a String, represents Hadoop's user name.
     *
     */
    def write_HDFS(uri: String, filePath: String, data: Array[Byte],
                   hadoop_user: String): Unit = {
        // Setting the name of Hadoop's user.
        System.setProperty("HADOOP_USER_NAME", hadoop_user)

        val path = new Path(filePath)
        // sets the default file system path to the given URI
        conf.set("fs.defaultFS", uri)
        val fs = FileSystem.get(conf)
        val os = fs.create(path)
        os.write(data) // write the data into the file
        fs.close() // close the filesysten
    }
    /**
    * saveFile, a function used to save an existing file, in HDFS.
    *
    * @param filepath, a String, represents the path of the file to store.

    def saveFile(filepath: String, srcFilePath:String, destFilePath:String):
```

```
        Unit = {
        val hdfs = FileSystem.get(conf)
        val srcPath = new Path(srcFilePath)
        val destPath = new Path(destFilePath)
        hdfs.copyFromLocalFile(srcPath, destPath)
    }
    /**
    * removeFile, a function used to remove a file stored in HDFS.
    *
    * @param filepath, a String, represents the path of the file to store.
    */
    def removeFile(filepath: String): Boolean = {
        val hdfs = FileSystem.get(conf)
        val path = new Path(filepath)
        hdfs.delete(path, true)
    }
    /**
     * getFile, a function used to get a file stored in HDFS.
     *
     * @param filepath, a String, represents the path of the file to store.
     */
     def getFile(filepath: String): InputStream = {
         val hdfs = FileSystem.get(conf)
         val path = new Path(filepath)
         hdfs.open(path)
     }
     /**
     * createFolder, a function used to create a folder in HDFS file structure.
     *
     * @param folderPath, a String, represents the folder path to created
     */
     def createFolder(folderPath: String): Unit = {
         val hdfs = FileSystem.get(conf)
         val path = new Path(folderPath)
         if (!hdfs.exists(path)) {
              hdfs.mkdirs(path)
         }
     }
}

object HDFS_Coordinator{
     def apply(): HDFS_Coordinator = new HDFS_Coordinator()
}
```

Once we get the dataset to process into HDFS, then we can load it into any of data structures offered in Spark. Spark offers different options when choosing a data structure, such as :

- **RDD**
  - **Resilient Distributed Dataset** (aka **RDD**) is the primary data abstraction in Apache Spark and the core of Spark (that I often refer to as "Spark Core").
- **Dataframe**
  - Dataframe was the second addition to Spark. It is a distributed collection of data organised into named columns.
- **Dataset**
  - As we've seen so far dataframes are kind of nice because they provide a declarative API along with query plan optimisation however they lack the type safety of the RDDs.

- **Tungsten**
  - As dataset knows which data types it holds it can provides hints to generate encoders to save and operate on data in tungsten format.
- **Graphframe**
  - The last data structure I'm going to cover is the graphframe. Graphframes are dedicated to graph storage and manipulation.

We will focus on RDDs.

RDD

Spark started with only one data structure: the RDD which stands for Resilient Distributed Dataset. This data structure has interesting properties (as indicated by its name):

- Resilient: A RDD is resilient and can be recomputed in case of failure
- Distributed: A RDD can be partitioned and distributed over several nodes.
- Immutable: A RDD can't be modified. Instead we apply a transformation to generate a new RDD.
- Lazy: A RDD represents a computation result but doesn't trigger any computation. That means one can describe a whole chain of computation (or DAG – Directed Acyclic Graph) by transforming a RDD into another RDD.
- Statically typed: A RDD has a type (e.g. RDD[String] or RDD[Person])

Working with RDD comes naturally for anyone familiar with the Scala collection API. The whole computation is triggered when an action is applied to a RDD. The action triggers all the operations required to compute the final RDD. [23]

```
sc.textFile("README.md")
   .flatMap(_.split(" "))
   .filter(_ != "")
   .groupBy(_.toLowerCase)
   .mapValues(_.size)
   .foreach { case (w,c) => println(s"$w: $c") }
```

RDDs are the lowest-level data structures in Spark. RDDs and their transformations describe how to compute things (similar to what a map-reduct job does) but doesn't show what it does directly (similar to a definition language like SQL). [23]

As a result there is no optimisations performed on RDD operations.

Each operation is executed where it appears in the DAG (i.e. the DAG is not optimised).

In this case we implemented different functions to do the following task:

- ❑ **Read a text dataset into RDD of Strings.**
  - ❑ to read a text dataset into RDD of Strings.
  - ❑ Note, that reading a dataset into RDD, does not imply to store the read dataset into memory.
  - ❑ this way only creates an RDD that says "we will need to load this file". The file is not loaded at this point.
- ❑ **Load a RDD into memory**
  - ❑ RDD.cache, is A lazy operation. The file is still not read. In this case, the RDD says "read this file and then cache the contents".
  - ❑ If we then run rdd.count the first time, the file will be loaded, cached, and counted.
  - ❑ If we call rdd.count a second time, the operation will use the cache. It will just take the data from the cache and count the lines.
  - ❑ Note that The cache behavior depends on the available memory.
    - ❑ If the file does not fit in the memory, for example, then textFile.count will fall back to the usual behavior and re-read the file. [24]
- ❑ **Parse RDD vector as an RDD of LabeledPoint**

```
/**
* read_dataset_into_RDD, a function used to read a text dataset into RDD of Strings.
*
* Note, that reading a dataset into RDD, does not imply to store the read dataset into
  memory.

* this way only creates an RDD that says "we will need to load this file". The file is
  not loaded at this point.
*
* @param dataset_path, a String, represents the path to the dataset to read.
*/
def read_dataset_into_RDD(dataset_path: String): RDD[String]= {
    val textFile: RDD[String] = this.mySparkContext.textFile(dataset_path)
    textFile
}

/**
* load_rdd_in_memory, a function used to load a RDD into memory
* RDD.cache, is A lazy operation. The file is still not read.
* In this case, the RDD says "read this file and then cache the contents".
*
* If we then run rdd.count the first time, the file will be loaded, cached, and counted.
* If we call rdd.count a second time, the operation will use the cache.
* It will just take the data from the cache and count the lines.
*
* Note that The cache behavior depends on the available memory.
* If the file does not fit in the memory, for example, then textFile.count will fall back
* to the usual behavior and re-read the file.
*
* @param rdd, the RDD to be loaded into memory
*/
```

```
def load_rdd_in_memory[T <: BaseType](rdd: RDD[T]): Unit = {
    rdd.cache
}
/**
*
* parse_dataset_as_rdd_LabledPoint, parse RDD vector as an RDD of LabeledPoint
* @param data, the data to parse as RDD
* @return an object of RDD[LabeledPoint]
*
* */
def parse_dataset_as_rdd_LabledPoint(data: RDD[String]): RDD[LabeledPoint] = {
    import org.apache.spark.mllib.linalg.Vectors
    data.map { line =>
          val parts = line.split(',').map(_.toDouble)
          LabeledPoint(parts(0), Vectors.dense(parts.tail))
    }
}
```

Once a dataset is loaded into RDD, it's practically prepared to start working with it to be processed.

3. Data Processing

In order to make to the data processing over any dataset, we decided to use MLlib library of Spark. Apache Spark MLlib is the Apache Spark scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, and underlying optimization primitives. Spark MLLib seamlessly integrates with other Spark components such as Spark SQL, Spark Streaming, and DataFrames and is installed in the Databricks runtime. [25]

Its goal is to make practical machine learning scalable and easy. At a high level, it provides tools such as:

- ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
- Featurization: feature extraction, transformation, dimensionality reduction, and selection
- Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
- Persistence: saving and load algorithms, models, and Pipelines
- Utilities: linear algebra, statistics, data handling, etc.

An example of SVM (support virtual machine) can be:

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
// scalastyle:off println
package org.apache.spark.examples.mllib
import org.apache.spark.sql.SparkSession
import org.apache.spark.{SparkConf, SparkContext}
import spark_catalog_package.{Algorithms, Catalog_Coordinator}
// $example on$
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.util.MLUtils
// $example off$
object SVMWithSGDExample {
      def main(args: Array[String]): Unit = {
      val ss = SparkSession
            .builder() // Create a SparkSession.Builder for constructing a SparkSession.
            .appName("myApp") // Specify the application name
            .master("local") // Specify the master
            .getOrCreate() // Get an exisiting session or Create a new one
      val sc = ss.sparkContext
      val spark_home = sys.env("SPARK_HOME")
      val dataset_path = spark_home.concat("/data/mllib/sample_libsvm_data.txt")

      val data = MLUtils.loadLibSVMFile(sc, dataset_path)
      // Split data into training (60%) and test (40%).
```

```
      val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
      val training = splits(0).cache()
      val test = splits(1)
      // Run training algorithm to build the model
      val numIterations = 100
      val model = SVMWithSGD.train(training, numIterations)
      // Clear the default threshold.
      model.clearThreshold()
      // Compute raw scores on the test set.
      val scoreAndLabels = test.map { point =>
            val score = model.predict(point.features)
            (score, point.label)
      }
      // Get evaluation metrics.
      val metrics = new BinaryClassificationMetrics(scoreAndLabels)
      val auROC = metrics.areaUnderROC()
      println(s"Area under ROC = $auROC")
      // Save and load model
      model.save(sc, "target/tmp/scalaSVMWithSGDModel")
      val sameModel = SVMModel.load(sc, "target/tmp/scalaSVMWithSGDModel")
      // $example off$
            sc.stop()
            }
}
// scalastyle:on println
```

Chapter 7

Auto Scaling

This chapter explains the importance of durability and scalability for a cloud-based applications. In most cases, really achieving these qualities means automating tasks such as scaling and other operational tasks.

One of the main characteristics of a Cloud Computing platform is to be able to offer auto scaled clusters. That means offering an adaptive computing services which can responsive to the client demand. In our case we decided to use OpenStack Heat to implement the required functionalities.

1. OpenStack Orchestration

The mission of the OpenStack Orchestration program is to create a human- and machine-accessible service for managing the entire lifecycle of infrastructure and applications within OpenStack clouds. [26][27][28]

1.1 Heat

Heat is the main project in the OpenStack Orchestration program. It implements an orchestration engine to launch multiple composite cloud applications based on templates in the form of text files that can be treated like code. A native Heat template format is evolving, but Heat also endeavours to provide compatibility with the AWS CloudFormation template format, so that many existing CloudFormation templates can be launched on OpenStack. Heat provides both an OpenStack-native ReST API and a CloudFormation-compatible Query API. [26][27][28]

Why 'Heat'? It makes the clouds rise!

1.1.1 How it works

- A Heat template describes the infrastructure for a cloud application in a text file that is readable and writable by humans, and can be checked into version control, diffed, &c.
- Infrastructure resources that can be described include: servers, floating ips, volumes, security groups, users, etc.
- Heat also provides an auto scaling service that integrates with Telemetry, so you can include a scaling group as a resource in a template.
- Templates can also specify the relationships between resources (e.g. this volume is connected to this server). This enables Heat to call out to the OpenStack APIs to create all of your infrastructure in the correct order to completely launch your application.
- Heat manages the whole lifecycle of the application - when you need to change your infrastructure, simply modify the template and use it to update your existing stack. Heat knows how to make the necessary changes. It will delete all of the resources when you are finished with the application, too.
- Heat primarily manages infrastructure, but the templates integrate well with software configuration management tools such as Puppet and Chef. The Heat team is working on providing even better integration between infrastructure and software.

1.1.2 Architecture

Heat comprises a number of Python applications:

- **heat**

The heat tool is a CLI which communicates with the heat-api to execute AWS CloudFormation APIs. Of course this is not required—developers could also use the Heat APIs directly.

- **heat-api**

The heat-api component provides an OpenStack-native ReST API that processes API requests by sending them to the heat-engine over RPC.

- **heat-api-cfn**

The heat-api-cfn component provides an AWS-style Query API that is compatible with AWS CloudFormation and processes API requests by sending them to the heat-engine over RPC.

- **heat-engine**

The heat engine does the main work of orchestrating the launch of templates and providing events back to the API consumer.

2. Auto Scaling for Compute

Automatically scale out your Compute instances in response to system usage.

This guide describes how to automatically scale out your Compute instances in response to heavy system usage. By using pre-defined rules that consider factors such as CPU or memory usage, you can configure Orchestration (heat) to add and remove additional instances automatically, when they are needed

2.1. Architectural Overview

2.1.1. Orchestration

The core component providing automatic scaling is Orchestration (heat). Orchestration allows you to define rules using human-readable YAML templates. These rules are applied to evaluate system load based on Telemetry data to find out whether there is need to add more instances into the stack. Once the load has dropped, Orchestration can automatically remove the unused instances again. [26][27][28]

2.1.2. Telemetry

Telemetry does performance monitoring of your OpenStack environment, collecting data on CPU, storage, and memory utilization for instances and physical hosts. Orchestration templates examine Telemetry data to assess whether any pre-defined action should start.

2.1.3. Key Terms

- **Stack** - A stack stands for all the resources necessary to operate an application. It can be as simple as a single instance and its resources, or as complex as multiple instances with all the resource dependencies that comprise a multi-tier application.

- **Templates** - YAML scripts that define a series of tasks for Heat to execute. For example, it is preferable to use separate templates for certain functions:

  - **Template File** - This is where you define thresholds that Telemetry should respond to, and define the auto scaling group.
  - **Environment File** - Defines the build information for your environment: which flavor and image to use, how the virtual network should be configured, and what software should be installed.

2.2. Example: Auto Scaling Based on CPU Usage

In this example, Orchestration examines Telemetry data, and automatically increases the number of instances in response to high CPU usage. A stack template and environment template are created to define the needed rules and subsequent configuration. This example makes use of existing resources (such as networks), and uses names that are likely to differ in your own environment. [26][27][28]

2.2.1 Creating the Auto Scaling requirements

1. Create the environment template, describing the instance flavor, networking configuration, and image type and save it in the template `/home/<user>/stacks/example1/cirros.yaml` file. Please, replace the `<user>` variable with a real user name. In my case the user is CC_rashad.

```
heat_template_version: 2016-10-14
description: Template to spawn an cirros instance.

parameters:
  metadata:
     type: json
  image:
     type: string
     description: image used to create instance
     default: cirros
  flavor:
     type: string
     description: instance flavor to be used
     default: m1.tiny
  key_name:
     type: string
     description: keypair to be used
     default: mykeypair
  network:
     type: string
```

```
    description: project network to attach instance to
    default: internal1
  external_network:
    type: string
    description: network used for floating IPs
    default: external_network

resources:
  server:
    type: OS::Nova::Server
    properties:
    block_device_mapping:
    - device_name: vda
        delete_on_termination: true
        volume_id: { get_resource: volume }
    flavor: {get_param: flavor}
    key_name: {get_param: key_name}
    metadata: {get_param: metadata}
    networks:
    - port: { get_resource: port }

port:
    type: OS::Neutron::Port
    properties:
    network: {get_param: network}
    security_groups:
    - default

floating_ip:
    type: OS::Neutron::FloatingIP
    properties:
    floating_network: {get_param: external_network}

floating_ip_assoc:
    type: OS::Neutron::FloatingIPAssociation
    properties:
    floatingip_id: { get_resource: floating_ip }
    port_id: { get_resource: port }

volume:
    type: OS::Cinder::Volume
    properties:
    image: {get_param: image}
    size: 1
```

2. Register the Orchestration resource in `~/stacks/example1/environment.yaml`:

```
resource_registry:

    "OS::Nova::Server::Cirros": ~/stacks/example1/cirros.yaml
```

3. Create the stack template, describing the CPU thresholds to watch for, and how many instances should be added. An instance group is also created, defining the minimum and maximum number of instances that can participate in this template. [26][27][28]

Save the following values in `~/stacks/example1/template.yaml`:

```
heat_template_version: 2016-10-14
description: Example auto scale group, policy and alarm
resources:
  scaleup_group:
     type: OS::Heat::AutoScalingGroup
     properties:
     cooldown: 300
     desired_capacity: 1
     max_size: 3
     min_size: 1
     resource:
     type: OS::Nova::Server::Cirros
     properties:
          metadata: {"metering.server_group": {get_param: "OS::stack_id"}}

scaleup_policy:
     type: OS::Heat::ScalingPolicy
     properties:
     adjustment_type: change_in_capacity
     auto_scaling_group_id: { get_resource: scaleup_group }
     cooldown: 300
     scaling_adjustment: 1

scaledown_policy:
     type: OS::Heat::ScalingPolicy
     properties:
     adjustment_type: change_in_capacity
     auto_scaling_group_id: { get_resource: scaleup_group }
     cooldown: 300
     scaling_adjustment: -1

cpu_alarm_high:
     type: OS::Aodh::GnocchiAggregationByResourcesAlarm
     properties:
     description: Scale up if CPU > 80%
     metric: cpu_util
     aggregation_method: mean
     granularity: 300
     evaluation_periods: 1
     threshold: 80
     resource_type: instance
     comparison_operator: gt
     alarm_actions:
     - str_replace:
          template: trust+url
          params:
               url: {get_attr: [scaleup_policy, signal_url]}
     query:
     str_replace:
          template: '{"=": {"server_group": "stack_id"}}'
          params:
          stack_id: {get_param: "OS::stack_id"}
```

```
cpu_alarm_low:
    type: OS::Aodh::GnocchiAggregationByResourcesAlarm
    properties:
    metric: cpu_util
    aggregation_method: mean
    granularity: 300
    evaluation_periods: 1
    threshold: 5
    resource_type: instance
    comparison_operator: lt
    alarm_actions:
    - str_replace:
         template: trust+url
         params:
              url: {get_attr: [scaledown_policy, signal_url]}
    query:
    str_replace:
         template: '{"=": {"server_group": "stack_id"}}'
         params:
         stack_id: {get_param: "OS::stack_id"}

outputs:
  scaleup_policy_signal_url:
     value: {get_attr: [scaleup_policy, signal_url]}

  scaledown_policy_signal_url:
     value: {get_attr: [scaledown_policy, signal_url]}
```

4. Run the following OpenStack command to build the environment and deploy the instance:

```
$ openstack stack create  -t template.yaml -e environment.yaml example
+---------------------+--------------------------------------------+
| Field               | Value                                      |
+---------------------+--------------------------------------------+
| id                  | 248a98bb-f56e-4934-a281-fffde62d78d8       |
| stack_name          | example                                    |
| description         | Example auto scale group, policy and alarm |
| creation_time       | 2017-03-06T15:00:29Z                       |
| updated_time        | None                                       |
| stack_status        | CREATE_IN_PROGRESS                         |
| stack_status_reason | Stack CREATE started                       |
+---------------------+--------------------------------------------+
```

5. Orchestration will create the stack and launch a defined minimum number of cirros instances, as defined in the `min_size` parameter of the `scaleup_group` definition. Verify that the instances were created successfully:

```
$ openstack server list
+--------------------------------------+-------------------------------------------------------------+--------
+------------+-------------+--------------------------------------+
| ID                                   | Name                                                        | Status
| Task State | Power State | Networks                             |
+--------------------------------------+-------------------------------------------------------------+--------
+------------+-------------+--------------------------------------+
| e1524f65-5be6-49e4-8501-e5e5d812c612 | ex-3gax-5f3a4og5cwn2-png47w3u2vjd-server-vaajhuv4mj3j | ACTIVE
| -          | Running     | internal1=10.10.10.9, 192.168.122.8 |
+--------------------------------------+-------------------------------------------------------------+--------
```

6. Orchestration also creates two cpu alarms which are used to trigger scale-up or scale-down events, as defined in `cpu_alarm_high` and `cpu_alarm_low`. Verify that the triggers exist:

```
$ openstack alarm list
+--------------------------------------+----------------------------------------------+----------
--------------------------+------------------+----------+---------+
| alarm_id                             | type                                         | name
      | state            | severity | enabled |
+--------------------------------------+----------------------------------------------+----------
--------------------------+------------------+----------+---------+
| 022f707d-46cc-4d39-a0b2-afd2fc7ab86a | gnocchi_aggregation_by_resources_threshold |
example-cpu_alarm_high-odj77qpbld7j | insufficient data | low      | True|
| 46ed2c50-e05a-44d8-b6f6-f1ebd83af913 | gnocchi_aggregation_by_resources_threshold |
example-cpu_alarm_low-m37jvnm56x2t  | insufficient data | low      | True|
+--------------------------------------+----------------------------------------------+----------
--------------------------+------------------+----------+---------+
```

2.2.2. Test Automatic Scaling Up Instances

Orchestration can scale instances automatically based on the `cpu_alarm_high` threshold definition. Once the CPU utilization reaches a value defined in the `threshold` parameter, another instance is started to balance the load. The `threshold` value in the above `template.yaml` file is set to 80%. [26][27][28]

1. Login to the instance and run several `dd` commands to generate the load:

```
$ ssh -i ~/mykey.pem cirros@192.168.122.8
$ sudo dd if=/dev/zero of=/dev/null &
$ sudo dd if=/dev/zero of=/dev/null &
$ sudo dd if=/dev/zero of=/dev/null &
```

2. Having run the `dd` commands, you can expect to have 100% CPU utilization in the cirros instance. Verify that the alarm has been triggered:

```
$ openstack alarm list
+--------------------------------------+----------------------------------------------+----------
--------------------------+-------+----------+---------+
| alarm_id                             | type                                         | name
      | state | severity | enabled |
+--------------------------------------+----------------------------------------------+----------
--------------------------+-------+----------+---------+
| 022f707d-46cc-4d39-a0b2-afd2fc7ab86a | gnocchi_aggregation_by_resources_threshold |
example-cpu_alarm_high-odj77qpbld7j | alarm | low      | True|
| 46ed2c50-e05a-44d8-b6f6-f1ebd83af913 | gnocchi_aggregation_by_resources_threshold |
example-cpu_alarm_low-m37jvnm56x2t  | ok  | low        | True|
+--------------------------------------+----------------------------------------------+----------
--------------------------+-------+----------+---------+
```

3. After some time (approximately 60 seconds), Orchestration will start another instance and add it into the group. You can verify this with the `nova list` command:

```
$ openstack server list
+--------------------------------------+-------------------------------------------------------
+--------+------------+-------------+--------------------------------------+
| ID                                   | Name                                                  |
Status | Task State | Power State | Networks                             |
+--------------------------------------+-------------------------------------------------------
+--------+------------+-------------+--------------------------------------+
| 477ee1af-096c-477c-9a3f-b95b0e2d4ab5 | ex-3gax-4urpikl5koff-yrxk3zxzfmpf-server-2hde4tp4trnk
| ACTIVE | -      | Running    | internal1=10.10.10.13, 192.168.122.17 |
| e1524f65-5be6-49e4-8501-e5e5d812c612 | ex-3gax-5f3a4og5cwn2-png47w3u2vjd-server-vaajhuv4mj3j
```

```
| ACTIVE | -      | Running    | internal1=10.10.10.9, 192.168.122.8   |
+--------------------------------------+-------------------------------------------------------
+--------+------------+-------------+---------------------------------------+
```

4. After another short period, you will observe that Orchestration has auto scaled again to three instances. The configuration is set to three instances maximally, so it will not scale any higher (the `scaleup_group` definition: `max_size`). Again, you can verify that with the above mentioned command:

```
$ openstack server list
+--------------------------------------+-------------------------------------------------------
+--------+------------+-------------+---------------------------------------+
| ID                                   | Name                                                  |
Status | Task State | Power State | Networks                                |
+--------------------------------------+-------------------------------------------------------
+--------+------------+-------------+---------------------------------------+
| 477ee1af-096c-477c-9a3f-b95b0e2d4ab5 | ex-3gax-4urpikl5koff-yrxk3zxzfmpf-server-2hde4tp4trnk
| ACTIVE | -      | Running    | internal1=10.10.10.13, 192.168.122.17 |
| e1524f65-5be6-49e4-8501-e5e5d812c612 | ex-3gax-5f3a4og5cwn2-png47w3u2vjd-server-vaajhuv4mj3j
| ACTIVE | -      | Running    | internal1=10.10.10.9, 192.168.122.8   |
| 6c88179e-c368-453d-a01a-555eae8cd77a | ex-3gax-fvxz3tr63j4o-36fhftuja3bw-server-rhl4sqkjuy5p
| ACTIVE | -      | Running    | internal1=10.10.10.5, 192.168.122.5   |
```

Chapter 8

Conclusions

8.1 Conclusions

As a main conclusion drawn from the work performed, we can state that the objectives have been reached. A catalogue for Big Data services on cloud platforms have been designed. This catalogue includes different data processing scripts inspired by MLlib library of Spark and using the required APIs libraries of Spark and HDFS to work with the different types of datasets.

The proposal has been implemented over a cloud platform with OpenStack as IaaS and, Hadoop and Spark as Big Data processing frameworks. A number of flexible scripts ---written in Ansible--- to automatically deploy Big Data clusters have been developed and thoroughly tested.

A portal scripts were designed and written in Python, which offer different functionalities to work with OpenStack.

The proposal designed in this memory can make much easier the use of Big Data services and can be easily extended towards serverless Big Data computation. This is a promise line that we have set as a future work

Bibliography

[1] Judith Hurwitz, Robin Bloor, Marcia Kaufman, Fern Halper, “Cloud Computing For Dummies” 2009, Wiley Publishing, Inc. - ISBN: 978-0-470-48470-8. pp 24.

[2] Christine Doig, Michele Chambers, Ian Stokes-Rees, “Breaking Data Science Open, 1st Edition”, 2017, O'Reilly Media, Inc. - ISBN: 9781491990735. pp 9.

[3] SAS analytics company. A definition for BigData

[4] Oracle. A definition for BigData

[5] International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 17 (2017) pp. 6970-6982 © Research India Publications. http://www.ripublication.com 6970. Cloud Computing and Big Data is there a Relation between the Two: A Study. Nabeel Zanoon1, Abdullah Al-Haj2, Sufian M Khwaldeh3 Nabeel Zanoon1, Abdullah Al-Haj2, Sufian M Khwaldeh

[6] Mell, P.; Grance, T. (September 2011). "The NIST Definition of Cloud Computing. NIST Special Publication 800-145 (September 2011). National Institute of Standards and Technology, U.S. Department of Commerce"

[7] Van V.N., Chi L.M., Long N.Q., Nguyen G.N., Le DN. (2016) A Performance Analysis of OpenStack Open-Source Solution for IaaS Cloud Computing. In: Satapathy S., Raju K., Mandal J., Bhateja V. (eds) Proceedings of the Second International Conference on Computer and Communication Technologies. Advances in Intelligent Systems and Computing, vol 380. Springer, New Delhi.

[8] OpenStack Official Documentation.

[9] OpenStack Official Documentation, Compute.

[10] OpenStack Official Documentation, Keystone.

[11] OpenStack Official Documentation, Glance.

[12] OpenStack Official Documentation, Neutron.

[13] OpenStack Official Documentation. Compute API.

[14] Michael DeHaan’s Blog, The Origins of Ansible.

[15] Ansible Design.

[16] Ansible Architecture.

[17] Distributed Storage and Processing Method for Big Data Sensing Information of Machine Operation Condition Fan Zhang, Zude Zhou, Wenjun Xu School of Information Engineering, Wuhan University of Technology, Wuhan 430070, China; Key Lab. of Fiber Optic Sensing Technology and Information Processing, Ministry of Education, Wuhan 430070, China JOURNAL OF SOFTWARE, VOL. 9, NO. 10, OCTOBER 2014

[18] Hadoop File System - HDFS Storage

[19] Apache YARN Architecture

[20] Apache Spark Resource Management and YARN App Models

[21] Spark Applications

[22] Running Spark over YARN

[23] Data structures in Spark

[24] Loading RDDs in Spark

[25] MLLib Machine Learning Library of Spark

[26] Orchestration Service API in OpenStack

[27] Orchestration in OpenStack

[28] Auto Scaling for Compute in OpenStack

[29] Service Catalog Code - Personal Github Repository

[30] OCCML Github Repository