Guide for Apache Hadoop 3.1.3, Apache Spark 2.4.4, Terraform 0.12 and AWS services on AWS EC2 Instances of Multiple Accounts

Giacomo Medda, Giacomo Meloni
April 14, 2020

Summary

Listings

1 Introduction

This tutorial will guide you through the configuration of *Apache Hadoop* (version 3.1.3) and *Apache Spark* (versione 2.4.4) on an *AWS* EC2 instance. It will be shown the creation of a cluster by means of an AWS Organization, which permits instances of multiple accounts to communicate with each other. Also *Terraform* (versione 0.12) usage will be covered to speed up the launching of several instances at once.

First of all AWS Organization and VPC services will be covered to set up our system in order to share resources between multiple accounts and make communicate EC2 instances.

2 AWS Organization, Private VPC and Subnet, Resource Share

This guide starts from the creation of an AWS Organization, which is similarly described on AWS website. First of all, this guide takes for granted that you are already logged in AWS console. For the actions that will be described AWS suggests logging in as IAM user, and not as a root user. In this way the master of the organization can manage who can access to a resource and can protect its own data and resources, since a root user has illimited access to resources. AWS explains how to create an IAM role and group and the IAM best practices, but this guide will not cover this topic and all actions will be performed as root user.

2.1 AWS Organization Creation

This part can be splitted in 3 steps:

1. Go to AWS Organization Introduction page
2. In the Introduction page left click on ”Create organization” and then left click ”Create organization” on the next screen to confirm (the master of the organization is labelled by a star)
3. AWS will send you an email to confirm your email address

2.2 Inviting accounts to join the organization

Now it is possible to invite new accounts to join the organization (some hours or a day could be necessary before being able to invite accounts because is not completely initialized, otherwise contact AWS Support). Inviting is done with the following steps:

1. Left click on ”Add account” in ”Accounts” tab
2. Choose ”Invite Account”
3. Write the e-mail address or the ID of the account to invite (multiple accounts should be divided by a comma) and optionally a note
4. Left click ”Invite”

2.3 Resource Sharing Enabling

Now it is necessary to enable the sharing of the resources (which will be created in next sections) among the members of the organization. This can be done in the settings of the *Resource Access Manager* (RAM), where the option ”Enable sharing within your AWS Organization” need to be checked. Then confirm it with ”Save settings”.

2.4 VPC Creation

Select the option *VPC* (Virtual Private Cloud) in the group ”Networking & Content Delivery” of the menu ”Services” and perform the following steps:

1. Left click on ”Launch VPC Wizard”, then ”Select” on the wizard page
2. Choose the name of the VPC in ”VPC name” and confirm with ”Create VPC”
3. Select the option ”Subnets” on the menu on the left. Here it is present a subnet called ”Public subnet”. This is the subnet created with the ”Launch VPC Wizard” with the same CIDR shown in the wizard, e.g. 10.0.0.0/24
4. Change the name of the subnet with one of your choice, but do not leave ”Public subnet”, as it is a default name and you may confuse it with other subnets you could create with the wizard
5. Select the subnet just created and left click on ”Actions” (or right click on the subnet) and select the option ”Modify auto-assign IP settings”
6. Check the box ”Auto-assign IPv4” in the screen that appears in order to make AWS assign public IPv4 to the instances and let us connect to them via SSH
7. Left click on ”Save” to confirm

Now we have our private subnet where we can create our instances and be sure that they communicate with each other using only private IPv4.

2.5 Creation of a Resource Share

Select *Resource Access Manager* in the group ”Security, Identity & Compliance” of the menu ”Services”. With the following steps we will share the new subnet with the members of our organization:

1. Left click on ”Create a resource share”
2. Choose the name of the resource, e.g. ”Project Subnet” (since we are sharing a subnet)
3. Choose the option ”Subnets” in the menu ”Select resource type”. We should see the subnet previously created with its chosen name
4. Right click on ”Show organization structure” in the section ”Shared principals” and choose the entire organization. The option ”Allow external accounts” needs to be checked.
5. Confirm all with ”Create resource share”

Now you are back on RAM main page where AWS confirms the creation of the resource share, but some minutes could be necessary so that it is available.

3 Creation of an AWS EC2 instance with Hadoop 3.1.3 and Spark 2.4.4

Select the option *EC2* in the group ”Compute” of the menu ”Services”. Select the option ”Instances” of the menu on the left and left click on ”Launch Instance”.

3.1 Launch Instance

The following steps will guide you on the launch of a new instance:

1. Choose the AMI (Amazon Machine Image) left clicking on ”Select” on the right of the AMI, in this guide Ubuntu 18.04 64-bit is chosen (When you create your own AMI, you will find it in ”My AMIs” in the menu on the left)
2. Choose the type of the instance. The unique instance valid for the free-tier is the ”t2.micro”. In this guide ”r5.large” is used, but the configuration is identical.
   Left click on ”Next: Configure Instance Details”
3. Let `1` as value of ”Number of instances”, other instances can be created in different ways. In the ”Network” option choose the VPC previously created labelled by the assigned name. The subnet will be automatically set by AWS.
   Left click on ”Next: Add Storage”
4. Here it is possible to choose the storage of the instance. AWS will charge you only for the used storage, not the one configured, so the limit of 30 GiB is only for the used storage. In this guide a value of 30 GiB is chosen.
   Left click on ”Next: Add Tags”
5. Left click on ”Next: Configure Security Group”
6. Leave the option ”Create a new security group” set and change the name of the field ”Security group name”. In this guide ”hadoop” is chosen.
   Left click on ”Review and Launch”
7. Left click on ”Launch” (ignore the warnings of AWS)
8. In the dialog that appears select the option ”Create a new key pair” and choose a name in ”Key pair name” (in this guide ”bigdata” is chosen). Then left click on ”Download Key Pair”. It will download the private key with extension `.pem` that will permit you to connect to the instances via SSH.
   Left click on ”Launch instances” to confirm

3.2 Instances and Security Group Settings

First, we need to change the name of out instances. This can be done selecting the option ”Instances” in the menu on the left in the EC2 console. We can see there is a blank value under the column ”Name”. Hover the mouse over it and click the pencil icon to set **namenode** as name, this will be the master of our cluster.

NOTE
instances’ state can be modified by selecting the instance and left clicking on ”Actions” (or right clicking the instance), going to ”Instance State” and we will see some options. The most important are:

- Stop: shutdown the instance and can be started, hence only RAM data will be erased
- Start: restart the instance if it is stopped
- Terminate: permanently delete the instance and cannot be started, hence RAM and volume data will be erased

##

Now, the security group of our instances need to be modified in order to make them communicate with every port. Select the option ”Security Groups” of the group ”NETWORK & SECURITY” in the menu on the left and modify the security group with the following steps:

1. Check the box next to the security group created during the instance launch. In this guide ”hadoop” was used. It should appear a menu below the group with the tabs *Details / Inbound rules / Outbound rules / Tags*
2. Select ”Inbound rules and left click on ”Edit inbound rules”
3. Left click on ”Add rule” on the screen that appears
4. Select ”All Traffic” as *Type*. In *Source* select ”Custom” and write the chosen CIDR for the previously created subnet, e.g. 10.0.0.0/24
5. Confirm with ”Save rules”

3.3 Further Concepts about Key Pairs and AMI sharing

Here they will be described the methods to share Key Pairs and AMI with other accounts.

3.3.1 Key Pair Sharing

Now we have the ”.pem” private key to access to the instances via SSH. If you want to let other accounts of the organization have the same key to access your instances or maybe to create a cluster of instances that share the same key, you need to send them the private key. However, it is not enough, they need also the public key. This can be generated with the command:

```
- ssh-keygen -y -f key.pem > key.pub
```

Listing 1: Private to Public EC2 Key

This code can be executed on a Linux shell or on Windows Powershell with ”Client OpenSSH” functionality installed. It takes a ”key.pem” file inside the folder where it is executed, so the name must be substituted with your private key file (it would be appropriate to choose the same name for the public key with extension ”.pub”)

Once you sent the key, the other accounts need to import it in their AWS EC2. Import phase is done with the following steps:

1. In EC2 Console left click on the option ”Key Pairs” in the group ”Network & Security” in the menu on the left.
2. Left clicking on ”Actions” will open a drop down menu where we need to select ”Import key pair”
3. Choose the same name of the original key in the field ”key pair” and select the file ”.pub” browsing your file system with the button ”Browse”
4. Once the public key is imported, AWS will recognize it with a green check mark. Confirm left clicking ”Import key pair”

3.3.2 AMI Sharing

The creation of the AMI will be described later in the guide, but we will give an explanation of the procedure to share a private AMI whether you choose the option to create one. The private AMI sharing is done with the following steps:

1. Select the option ”AMI” inside the group ”IMAGES” in the menu on the left in EC2 console
2. Select the AMI and left click on ”Actions” (or right click on the AMI) and left click on ”Modify Image Permissions”
3. Add the AWS Account Number of the accounts you want to share the AMI with, left click on ”Add Permission” for each account and confirm with ”Save”

3.4 Configuration of the Instance

3.4.1 Connection, Hostnames & SSH Configuration

First of all, we need to connect to our master, the first EC2 instance created. This can be done with any OS, in Windows it is necessary to install *Client OpenSSH* from "Optional Features" inside the menu "Apps & Features" (for Windows 10). Open a shell (Linux shell or Powershell) and use the following to connect to an ubuntu instance:

```
- ssh -i key.pem ubuntu@PUBLIC_DNS_ADDRESS
```

Listing 2: Connect to an Instance

"key.pem" and "PUBLIC_DNS_ADDRESS" must be substituted with your private key and the public DNS address of your instance respectively (e.g.
`ssh -i "bigdata.pem" ubuntu@ec2-34-227-83-101.compute-1.amazonaws.com`).
The public DNS address is shown when the instance is selected in EC2 console.

Now we need to copy our private key inside our instance. This will permit instances to communicate with each other. To do it we need to create a new shell and execute the following command (valid for an ubuntu instance):

```
- scp -i 'key.pem' key.pem ubuntu@PUBLIC_DNS_ADDRESS:/home/ubuntu/.ssh
```

Listing 3: Copying Private Key to an Instance

An example of this command could be: `scp -i "bigdata.pem" bigdata.pem ubuntu@ec2-34-227-83-101.compute-1.amazonaws.com/home/ubuntu/.ssh`
Close this shell and get back to the previous one where we are connected to our instance.

Now we need to change the permissions of the key to be only readable by the owner with the command:

```
- chmod 400 /home/ubuntu/.ssh/key.pem
```

Listing 4: Private Key Permissions Change

where "key.pem" must be substituted with the one copied into the instance.

Now we set the new hostnames of our instance. Execute the command:

```
- sudo nano /etc/hosts
```

in this file you will see a part dedicated to the IPv4 addresses and the "localhost" hostname set. Write the following lines below localhost (leaving a blank line between IPv4 and IPv6 addresses, it will be useful if you choose to use Terraform later):

```
PRIVATE_IP_NAMENODE namenode
PRIVATE_IP_NAMENODE datanode1
```

Listing 5: Hostnames Configuration

where "PRIVATE_IP_NAMENODE" must be substitued with the private IPv4 of our master instance.
In these lines we can see "namenode" and "datanode1": they are the master, we use two names to create a logic difference because our instance will be the master of our cluster, but also a worker. Therefore, in following sections we will use "namenode" in master configuration files and "datanode1" in workers configuration files, but this is just a **logic** difference since they are converted into the same hostname when processed.

Now we configure SSH in order to ease the connection among the instances.
Execute the following command (valid for ubuntu):

```
- nano /home/ubuntu/.ssh/config
```

and write in:

```
Host namenode
HostName namenode
User ubuntu
IdentityFile /home/ubuntu/.ssh/my-key.pem
Host datanode1
HostName namenode
User ubuntu
IdentityFile /home/ubuntu/.ssh/my-key.pem
```

Listing 6: SSH Configuration

where "my-key.pem" must be substituted with your private key name.

3.4.2 Apache Hadoop Configuration

First we need to update the packages of our instance and install Java:

```
- sudo apt-get update && sudo apt-get dist-upgrade
- sudo apt-get install openjdk-8-jdk
```

Listing 7: Update Packages and Java Installation

Then with the following commands we will download *Apache Hadoop* in `tar.gz` format, extract it and move it in a more suitable directory:

```
- wget http://mirror.nohup.it/apache/hadoop/common/hadoop-3.1.3/hadoop
-3.1.3.tar.gz
- tar -xvzf ./hadoop-3.1.3.tar.gz
- sudo mv ./hadoop-3.1.3 /home/ubuntu/hadoop
- rm hadoop-3.1.3.tar.gz
```

Listing 8: Hadoop Installation

Now we will modify environment variables to ease the execution of some Hadoop scripts. The commands are:

```
- sudo nano /etc/environment
```

and the content of the document must become:

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
games:/usr/local/games:/home/ubuntu/hadoop/bin:/home/ubuntu/hadoop/sbin"
JAVA_HOME="/usr/lib/jvm/java-8-openjdk-amd64"
```

Listing 9: Environment Variables Update

Execute the following command to update the current shell environment with the new values:

```
- source /etc/environment
```

Now we will create some new environment variables to ease the access to the Hadoop folders. First execute the command:

```
- nano /home/ubuntu/.profile
```

and write at the end of the file:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=$PATH:$JAVA_HOME/bin
export HADOOP_HOME=/home/ubuntu/hadoop
export HADOOP_CONF_DIF=/home/ubuntu/hadoop/etc/hadoop
```

Listing 10: Hadoop Environment Variables

Execute the following command to update the current shell environment with the new values:

```
- source /home/ubuntu/.profile
```

Now we will modify the proper Hadoop configuration files. First, it is better to start by saying that the following configuration files contain a tag `<configuration>`, and we will write the values to be inserted in these files with this tag, but each file must contain one only `<configuration>` tag.

The 1st file is `hdfs-site.xml`. To open it execute the command:

```
- nano $HADOOP_CONF_DIF/hdfs-site.xml
```

and write inside the configuration tag (without duplicating it):

```
<configuration>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>/home/ubuntu/hadoop/data/namenode</value>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>/home/ubuntu/hadoop/data/datanode</value>
  </property>
  <property>
    <name>dfs.replication</name>
    <value>2</value>
  </property>
</configuration>
```

Listing 11: hdfs-site.xml Configuration

The 2nd file is `core-site.xml`. To open it execute the command:

```
- nano $HADOOP_CONF_DIF/core-site.xml
```

and write inside the configuration tag (without duplicating it):

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://namenode:9000</value>
  </property>
</configuration>
```

Listing 12: core-site.xml Configuration

The 3rd file is `yarn-site.xml` and concerns the configuration of *YARN*. To open it execute the command:

```
- nano $HADOOP_CONF_DIF/yarn-site.xml
```

and write inside the configuration tag (without duplicating it):

```
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
      <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class</name>
      <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>namenode</value>
  </property>
  <property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
  </property>
</configuration>
```

Listing 13: yarn-site.xml Configuration

The 4th file is `mapred-site.xml`. To open it execute the command:

```
- nano $HADOOP_CONF_DIF/mapred-site.xml
```

and write inside the configuration tag (without duplicating it):

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>yarn.app.mapreduce.am.env</name>
        <value>HADOOP_MAPRED_HOME=${HADOOP_HOME}</value>
    </property>
    <property>
        <name>mapreduce.map.env</name>
        <value>HADOOP_MAPRED_HOME=${HADOOP_HOME}</value>
    </property>
    <property>
        <name>mapreduce.reduce.env</name>
        <value>HADOOP_MAPRED_HOME=${HADOOP_HOME}</value>
    </property>
</configuration>
```

Listing 14: mapred-site.xml Configuration

The 5th file is `masters`. To open it execute the command:

```
- nano $HADOOP_CONF_DIF/masters
```

and write inside:

```
namenode
```

Listing 15: masters Configuration

The 6th file is `workers`. To open it execute the command:

```
- nano $HADOOP_CONF_DIF/workers
```

and delete "localhost" and write inside:

```
datanode1
```

Listing 16: workers Configuration

NOTE
The Hadoop configuration of all the nodes in the cluster need to be identical, so also the file `workers` need to be the same.
##

3.4.3 Apache Spark Configuration

First, we will download *Apache Spark* in format `tar.gz`, extract it and move it in a more suitable folder with the following commands:

```
- wget https://archive.apache.org/dist/spark/spark-2.4.4/spark-2.4.4-bin-
hadoop2.7.tgz
- tar -xvzf spark-2.4.4-bin-hadoop2.7.tgz
- sudo mv ./spark-2.4.4-bin-hadoop2.7 /home/ubuntu/spark
- rm spark-2.4.4-bin-hadoop2.7.tgz
```

Listing 17: Spark Installation

Now we modify a spark configuration file, that is `spark-env.sh`. The commands to create it and open it are:

```
- sudo cp spark/conf/spark-env.sh.template spark/conf/spark-env.sh
- sudo nano spark/conf/spark-env.sh
```

and write at the end of the file:

```
export SPARK_MASTER_HOST=namenode
export HADOOP_CONF_DIR="/home/ubuntu/hadoop/etc/hadoop"
export PYSPARK_PYTHON="/usr/bin/python3"
```

Listing 18: Spark Environment Variables

The next step is the creation of the file `slaves`, which permit us to start all Spark slaves with just one command. First, we create the file:

```
- nano spark/conf/slaves
```

and write inside:

```
datanode1
```

Listing 19: slaves Configuration

4 Creation of the Cluster

Now the guide splits in two parts:

1. Manual creation of the cluster, namely, creating the instance by means of the AMI[1] for each instance in the cluster
2. Configuration of Terraform. This toll permits us to create automatically a set number[2] of instances and we can automatically update the configuration files of all the instances in the cluster, master included, taking advantage of some scripts

4.1 Creation of the AMI

This section will make you create an AMI (Amazon Machine Image) of your instance "namenode". During this procedure, AWS will create a *snapshot* of namenode: if the size of this snapshot is more than 1GB AWS will charge you for a small amount of money. The instance we just configured should have a size a little less than 1GB.
This permits us to create new instances as image of the first one and speed up the creation of the cluster. As previously mentioned we can avoid this step creating new instances manually as the one just created.

To create the AMI go to EC2 console in the group "Instances", select the instance "namenode", left click on "Actions" (or right click the instance), select "Image" and "Create Image". Choose a name for the AMI and confirm with "Create Image" button.

4.2 Manual Creation of the Cluster

This section describes how to launch one[3] new instance using the AMI of the previous one and configure the files in order to make them communicate with each other, and so create the cluster.

4.2.1 Launch of an Instance with the AMI

To launch an instance select the option "AMI" inside the group "IMAGES" in the menu on the left in EC2 console. Select our new AMI and click "Launch". We will be brought to the same procedure to launch an instance, but now the AMI is already set. The next settings need to be the same of the other instance:

- **VPC (Subnet)**: same VPC, hence same subnet
- **Storage**: the storage size you prefer
- **Security Group**: same security group. This means you need to select the option "Select an existing security group" and choose the one you created with this guide
- **Key Pair**: same key pair. This means you need to select "Choose an existing key pair" and choose the one created with this guide
- **Name**: in EC2 console choose "datanode2" as name of the new instance

[1] This step is not strictly necessary. You can create another instance following the steps you previously followed (but in EC2 console rename the instances as "datanode2", "datanode3", etc.) and updating the configuration files as shown in the subsection 4.2.2

[2] Check the limits of your region for EC2. The standard limit of EC2 in us-east-1 is 32 vcores, which corresponds to a maximum of 16 r5.large instances beacuse each one has 2 vcores

[3] More instances can be launched in the same way, but the name of each instance need to be different in each file. Each "datanode2" used in the following configuration files must be substituted with the name you will assign to each new instance

4.2.2 Manual Configuration of the Cluster

The next commands need to be executed in the shell of the master. In this guide we will never use the shells of datanodes to be the least confusing possible. By means of SSH we will reflect the same modifications into ”datanode2”.

Firstly, we update the hostnames. Command to open the file of hostnames:

```
- sudo nano /etc/hosts
```

and write at the end of the IPv4 section (always leave a blank line between IPv4 and IPv6 sections, this is useful in case you will be using Terraform later):

```
PRIVATE_IP_DATANODE2 datanode2
```

Listing 20: Datanode Hostnames Configuration

where PRIVATE_IP_DATANODE2 must be subsituted with the private IPv4 of the instance ”datanode2”.

Now we modify the SSH configuration. Command to open the file:

```
- nano /home/ubuntu/.ssh/config
```

and write at the end of the file:

```
Host datanode2
HostName datanode2
User ubuntu
IdentityFile /home/ubuntu/.ssh/my-key.pem
```

Listing 21: Datanode SSH Configuration

where my-key.pem must be substitued with your private key.

The next step is the Hadoop workers file configuration. Command to open the file:

```
- nano $HADOOP_CONF_DIF/workers
```

and write at the end of the file:

```
datanode2
```

Listing 22: Datanode workers Configuration

The next file is the Spark slaves one. Remember that it is necessary to configure this file only in the master, doing it in the slaves would have no effect. Command to open the file:

```
- nano spark/conf/slaves
```

and write at the end of the file:

```
datanode2
```

Listing 23: Datanode slaves Configuration

We can test that SSH is working with the following commands (CTRL-D to close each connection):

```
- ssh datanode1     (to connect to the current host, that is ”namenode”)
- ssh datanode2     (to connect to the instance ”datanode2”)
```

Now from the master shell we will execute the commands to reflect the modification just done in "datanode2":

```
- cat /etc/hosts | ssh datanode2 "sudo sh -c 'cat >/etc/hosts'"
(to update hostnames)
- cat /home/ubuntu/.ssh/config | ssh datanode2 "sudo sh -c 'cat >/home/
ubuntu/.ssh/config'"     (to update SSH conf.)
- cat /home/ubuntu/hadoop/etc/hadoop/workers | ssh datanode2 "sudo sh -c '
cat >/home/ubuntu/hadoop/etc/hadoop/workers'" (to update workers conf.)
```

Listing 24: Update Modifications in Datanode

4.3 Creation of the Cluster with Terraform

4.3.1 Terraform Installation

Firstly, we need to download Terraform for Linux (in this guide we used Ubuntu 64-bit version) at downloads page of Terraform. We need to copy the link of the download and use wget in our shell to download it. In this guide the available versione was 0.12.24, so the command we used is:

```
- wget https://releases.hashicorp.com/terraform/0.12.24/terraform_0.12.24
_linux_amd64.zip
```

Use the link you can find in Terraform website in the same way.

Terraform is inside the archive we downloaded, but it is a `.zip` format, so we need `unzip` library to extract it. If not already installed, `unzip` can be installed with the command:

```
- sudo apt install unzip
```

Now we can extract the files from the archive and move them in a more suitable directory:

```
- unzip terraform_0.12.24_linux_amd64.zip
- rm terraform_0.12.24_linux_amd64.zip
- mkdir Terraform
- mv terraform Terraform/
```

Listing 25: Terraform Installation

Now we add terraform to the environment variables to make it callable anywhere in the system. To open the environment file the command is:

```
- sudo nano /etc/environment
```

and add at the end of the value of PATH the following line:

```
:/home/ubuntu/Terraform        (the colon ":" is necessary to divide paths)
```

Listing 26: Environment Variables Update

and after saving the file execute the following command to update the current shell environment with the new values:

```
- source /etc/environment
```

From now on we can use `terraform` command in any folder. Calling it without arguments will show all the possible usable arguments.

4.3.2 Terraform Configuration

However, we want to leave clean our home, so we will create the configuration files in the folder named `Terraform` that we previously created. Here terraform commands must be called in order to make terraform read our file. The configuration file `main.tf` can be created with the command:

```
- nano Terraform/main.tf
```

and write inside:

```
provider "aws" {
  profile = "default"
  region = "REGION"
}

resource "aws_instance" "testInstances" {
   ami = "AMI_ID"
   instance_type = "INSTANCE_TYPE"
   subnet_id = "SUBNET_ID"
   vpc_security_group_ids = [
      "SECURITY_GROUP_ID",
   ]
   count = NUM_INSTANCES
}
```

Listing 27: Terraform Configuration

All the names in uppercase must be substituted with your values (in particular, NUM_INSTANCES is the number of instances Terraform will create when you execute it). These lines add a provider, which permits Terraform to use some data to access the resources of our AWS account, and create a Terraform resource of type "aws_instance" called "testInstances" (leave "testInstances" as name to avoid problems later with the Terraform configuration, if you modify it you need to modify also the next files we will show in this guide).
An example of a `main.tf` file is:

```
provider "aws" {
  profile = "default"
  region = "us-east-1"
}

resource "aws_instance" "testInstances" {
   ami = "ami-01926394e20264954"
   instance_type = "r5.large"
   subnet_id = "subnet-00fbaf25f9b79f847"
   vpc_security_group_ids = [
      "sg-0980d39f95d96a1b1",
   ]
   count = 2
}
```

Listing 28: Example of main.tf

4.3.3 AWS CLI Installation and AWS Access Key

Terraform needs to use our AMI to create the instances, so we must give him a way to access it. This can be done giving Terraform an *Access Key*. But first, we will install AWS CLI (Command Line Interface), which is an easy way to create the files that Terraform will use to get the access key.
AWS has a specific page that will guide you on the installation of AWS CLI. In this guide we installed the last version, which was the 2nd. The commands to install it are:

```
- curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "
awscliv2.zip"
- unzip awscliv2.zip
- sudo ./aws/install
- rm awscliv2.zip
```

Listing 29: AWS CLI Installation

AWS CLI is now installed, but we need to create our access key before using the CLI. To create it go back to AWS console and left click on your username on the appbar on the top. Next to "Global" you should find your username and left clicking it should open a drop down menu where you need to select "My Security Credentials".
In the page of "Your Security Credentials" open the tab "Access keys (access key ID and secret access key)" and left click on "Create New Access Key". In the dialog box that appears select "Download Key File" and then close the dialog box.
You have now downloaded a CSV file containing an `AWSAccessKeyId` and an `AWSSecretKey`. These are necessary to configure our AWS CLI.
Therefore, you must come back to the master shell and execute the command:

```
- aws configure
```

and write the data one by one as the following example:

```
AWS Access Key ID [None]: IN THE CSV AS AWSAccessKeyId
AWS Secret Access Key [None]: IN THE CSV AS AWSSecretKey
Default region name [None]: REGION (e.g. us-east-1)
Default output format [None]: json
```

Listing 30: AWS CLI Configuration

Terraform will automatically find the credentials to access the resources of our AWS account.

4.3.4 Terraform Usage and Datanode Files Update for the Cluster

Now we will see how to use Terraform to create the datanodes of our cluster and how to update the files of each datanode in order to setup the cluster. First we enter in our Terraform directory with the command:

```
- cd Terraform
```

and execute the following commands:

```
- terraform init
- terraform apply
```

Listing 31: Terraform Init and Apply

write yes and press Enter to confirm.
If `main.tf` is correctly configured the first command will print on the shell the resources found in `main.tf` and the second command should show you an overview of the instances that will be created accordingly to your configuration.

Once the instances are created, they can be destroyed with the command:

```
– terraform destroy
```

Listing 32: Terraform Destroy

this command must always be executed in the same folder where ”init” and ”apply” commands were used, in our case in the folder ”Terraform”.
If you modify the configuration and you already applied some instances, you need to destroy them and re-create them.

Now we need to update the files in the master and in the datanodes. For this part we must move back to our home with the command:

```
– cd ..   (since we are in the folder Terraform) or   – cd /home/ubuntu
```

4 files need to be updated in the master.

The 1st one is `/etc/hosts/`. Command to open it:

```
– sudo nano /etc/hosts
```

and at the end of the IPv4 section write:

```
PRIVATE_IP datanode*
```

the star ”*” must be substituted with the instance number, and this line must be written for each datanode created, so for 4 instances ”datanode2”, ”datanode3”, ”datanode4”, ”datanode5” (datanode1 corresponds to namenode, the master).

The 2nd one is `/home/ubuntu/.ssh/config`. Command to open it:

```
– nano /home/ubuntu/.ssh/config
```

and at the end of the file write:

```
Host datanode*
HostName datanode*
User ubuntu
IdentityFile /home/ubuntu/.ssh/my–key.pem
```

the star ”*” must be substituted with the instance number, and these lines must be written for each datanode created, so for 4 instances ”datanode2”, ”datanode3”, ”datanode4”, ”datanode5” (datanode1 corresponds to namenode, the master).
”my-key.pem” must be substituted with your private key.

The 3rd one is $HADOOP_CONF_DIF/`workers`. Command to open it:

```
– nano $HADOOP_CONF_DIF/workers
```

and at the end of the file write:

```
datanode*
```

the star ”*” must be substituted with the instance number, and this line must be written for each datanode created, so for 4 instances ”datanode2”, ”datanode3”, ”datanode4”, ”datanode5” (datanode1 corresponds to namenode, the master).

The 4th one is `/home/ubuntu/spark/conf/slaves`. Command to open it:

```
- nano /home/ubuntu/spark/conf/slaves
```

and at the end of the file write:

```
datanode*
```

the star "*" must be substituted with the instance number, and this line must be written for each datanode created, so for 4 instances "datanode2", "datanode3", "datanode4", "datanode5" (datanode1 corresponds to namenode, the master).

Now we need to reflect these modifications into the datanodes and we will do it from the shell of the master. The following commands need to be executed for each datanode:

```
- cat /etc/hosts | ssh datanode* "sudo sh -c 'cat >/etc/hosts'"
- cat /home/ubuntu/hadoop/etc/hadoop/workers | ssh datanode* "sudo sh -c '
cat >/home/ubuntu/hadoop/etc/hadoop/workers'"
- cat /home/ubuntu/.ssh/config | ssh datanode* "sudo sh -c 'cat >/home/
ubuntu/.ssh/config'"
```

the star "*" must be substituted with the instance number, and this line must be written for each datanode created, so for 4 instances "datanode2", "datanode3", "datanode4", "datanode5" (datanode1 corresponds to namenode, the master).

4.3.5 Automation & Improvement of Terraform Configuration

The procedure explained in 4.3.4 can be long-lasting with the increase of instances. We can automate using some Terraform features and some scripts.

First we modify the file `main.tf`. Command to open it from the home:

```
- nano Terraform/main.tf
```

and change all the content to:

```
provider "aws" {
  profile = "default"
  region = "REGION"
}

resource "aws_instance" "testInstances" {
   ami = "AMI_ID"
   instance_type = "INSTANCE_TYPE"
   subnet_id = "SUBNET_ID"
   vpc_security_group_ids = [
      "SECURITY_GROUP_ID",
   ]
   count = NUM_INSTANCES
}

resource "null_resource" "testInstances" {
   provisioner "local-exec" {
      command = join("_", aws_instance.testInstances.*.private_ip)
      interpreter = ["bash", "/home/ubuntu/clusterSetup.sh", "MY-KEY", "
INDEX_START"]
   }

   provisioner "local-exec" {
      when = destroy
      command = NUM_INSTANCES
      interpreter = ["bash", "/home/ubuntu/clusterClean.sh", "INDEX_START"
]
      on_failure = continue
   }
}
```

Listing 33: Configuration Upgrade of main.tf

All the names in uppercase must be substituted with your values as we have seen before. MY-KEY must be substituted with your private key, e.g. "bigdata.pem", INDEX_START with the index your datanode names' count starts with, for instance if you set it to 7 then the first datanode will named "datanode7" and so on until all the NUM_INSTANCES instances are created.
The usage of the field "interpreter" in this configuration is logically incorrect, but writing the script with its parameters inside the field "command", as shown in Terraform guide, did not work correctly with the version of Terraform that we used, and it always raised an error saying Terraform could not find our script. So this seemed a bug and taking advantage of the field "interpreter" solved the issue.

This configuration uses two scripts: `clusterSetup.sh` and `clusterClean.sh`. We are going to create them. Command to create the first one (create it in the home to make the `main.tf` configuration| work properly):

```
- nano clusterSetup.sh
```

and write inside:

```
#!/bin/bash

cat /etc/hosts > /home/ubuntu/.tmpHosts
cat /home/ubuntu/.ssh/config > /home/ubuntu/.tmpSSHConfig


index=$2
IFS='_' read -ra IPs <<<$3
for i in ${IPs[@]}; do
    awk -v ip="$i" -v idx="$index" '!x{x=sub(/^$/,ip" datanode"idx"\n")}1'
/etc/hosts > _tmp && sudo mv _tmp /etc/hosts
    echo -e "Host datanode${index}\nHostName datanode${index}\nUser ubuntu
\nIdentityFile /home/ubuntu/.ssh/${1}.pem" >> /home/ubuntu/.ssh/config
    echo "datanode${index}" | sudo tee -a /home/ubuntu/hadoop/etc/hadoop/
workers
    echo "datanode${index}" | sudo tee -a /home/ubuntu/spark/conf/slaves
    index=$((index + 1))
done
```

Listing 34: clusterSetup.sh

in the previous sections we explained that a blank line was necessary between the parts of IPv4 and IPv6 inside the file `/etc/hosts` because this script we created need that blank line to insert the IPv4 of each datanode created by Terraform (this permits `/etc/hosts` to be correctly structured), otherwise it will not work properly.

Command to create the second file (create it in the home to make the `main.tf` configuration| work properly):

```
- nano clusterClean.sh
```

and write inside:

```
#!/bin/bash

n_datanodes=$2
END=$((n_datanodes+2))
for ((i=$1;i<END;i++)); do
     ssh-keygen -f "/home/ubuntu/.ssh/known_hosts" -R "datanode"$i
done

sudo rm -r /home/ubuntu/hadoop/data/namenode
sudo rm -r /home/ubuntu/hadoop/data/datanode
sudo echo "datanode1" > /home/ubuntu/hadoop/etc/hadoop/workers
sudo echo "datanode1" > /home/ubuntu/spark/conf/slaves
sudo mv /home/ubuntu/.tmpHosts /etc/hosts
sudo mv /home/ubuntu/.tmpSSHConfig /home/ubuntu/.ssh/config
sudo rm -r /tmp/*
```

Listing 35: clusterClean.sh

Now we change the permissions of the files to be sure they work without problems:

- `chmod 777 clusterSetup.sh`
- `chmod 777 clusterClean.sh`

The next step is to start Terraform, so we need to move to the folder Terraform and execute the commands previously shown (if you have instances created by Terraform and still running, you need to destroy them before executing these commands):

- `cd Terraform`
- `terraform init`
- `terraform apply`

write yes and press Enter to confirm.

Now we go back to our home with the command:

- `cd ..` (since we are in the folder Terraform) or - `cd /home/ubuntu`

The master configuration files will be updated automatically by our scripts, but we need to update the files of the datanodes and this can be automated with another script. Choose the name you prefer, e.g. `updateDatanodes.sh`, and create it with the command:

- `nano updateDatanodes.sh`

and write inside:

```
#!/bin/bash

n_datanodes=$2
END=$((n_datanodes+2))
for ((i=$1;i<END;i++)); do
    cat /etc/hosts | ssh -oStrictHostKeyChecking=no datanode$i "sudo sh -c
 'cat >/etc/hosts'"
    cat /home/ubuntu/hadoop/etc/hadoop/workers | ssh -
oStrictHostKeyChecking=no datanode$i "sudo sh -c 'cat >/home/ubuntu/hadoop
/etc/hadoop/workers'"
    cat /home/ubuntu/.ssh/config | ssh -oStrictHostKeyChecking=no
datanode$i "sudo sh -c 'cat >/home/ubuntu/.ssh/config'"
done
```

Listing 36: updateDatanodes.sh

Save it and change its permissions with the command:

- `chmod 777 updateDatanodes.sh`

To use the script execute the command:

- `bash updateDatanodes.sh INDEX_START NUM_INSTANCES`

where INDEX_START must be substituted with the first index of your datanodes name, e.g. 4 if the first datanode is "datanode4" (this value is the same you used in `main.tf`) and NUM_INSTANCES with the number of instances created with Terraform (this value is the same you used in `main.tf`)

Note that when you execute the command "terraform destroy" every configuration file in the master will be brought back to its original version before Terraform was executed thanks to the script `clusterClean.sh`. Then, you do not need to modify any files and you can execute again "terraform init" and "terraform apply" without problems.

4.3.6 Unrefined Desciption of Terraform Usage with Instances of Multiple Accounts

If you want to use Terraform in this way and create a cluster, but with instances of 2 or more accounts, you need to create a "false master" in the other accounts (false because namenode will be always the master of the cluster).

For instance we will describe it with only 2 accounts.
This can be accomplished sharing your AMI, make the other account create one only instance with that AMI, inside the subnet you created, with the same security group and with the same key pair (this can be easy as described in the first sections if you be a part of the same organization) and for instance name it "datanode17". Terraform need to be installed inside this instance "datanode17" and configured with the scripts described above. Then you ask the other account the private IPv4 of the instance "datanode17".

Now you need to modify the configuration files only of "datanode17".
The private IPv4 needs to inserted in the file `/etc/hosts` as hostname "datanode17" and you need to modify the SSH config file with this new "datanode17" hostname, add "datanode17" to the file `workers` of Hadoop.

Now you need to execute "terraform init" and "terraform apply" only in "datanode17", but be careful with INDEX_START and NUM_INSTANCES because you cannot create duplicates, all the datanodes need to have a different name, so different index.

The next step is to copy the configuration files of "datanode17" into the ones of "namenode" with the following commands:

- **cat** /etc/hosts | ssh namenode "sudo sh -c 'cat >/etc/hosts'" (to update hostnames)
- **cat** /home/ubuntu/.ssh/config | ssh namenode "sudo sh -c 'cat >/home/ubuntu/.ssh/config'" (to update SSH conf.)
- **cat** /home/ubuntu/hadoop/etc/hadoop/workers | ssh namenode "sudo sh -c 'cat >/home/ubuntu/hadoop/etc/hadoop/workers'" (to update workers conf.)

After this step we can go back to the master shell (namenode). Here we can add "datanode17" to the file `slaves` of Spark, then execute "terraform init" and "terraform apply".

The last step is to execute the script `updateDatanode.sh` but we need to use the lower INDEX_START between the one in "namenode" and the one in "datanode17" and the sum of NUM_INSTANCES of "namenode" and "datanode17" plus 1, because we need to count also "datanode17". It is necessary that the names of the datanodes are contiguous in order to make the script work, e.g. the last datanode created by "namenode" must be "datanode16" and the datanodes created by "datanode17" must start from "datanode18".

When you execute "terraform destroy" in both of the instances, some of configuration files of "namenode" need to be cleaned: `/etc/hosts` and `/home/ubuntu/.ssh/config` because the script `clusterClean.sh` will bring these files back to its original version, but their original version is modified with the values copied from "datanode17". The files `workers` and `slaves` will contain only the value "datanode1" as from the beginning.

4.4 Hadoop (HDFS), YARN and Spark Usage

4.4.1 HDFS Usage

Now we need to format *HDFS (Hadoop Distributed File System)*. The command is:

```
- hdfs namenode -format
```

Listing 37: Format HDFS

And start the nodes of the cluster:

```
- start-dfs.sh
```

Listing 38: Start Hadoop Cluster

@ Test: create a directory inside HDFS and check its existence with the following commands:

```
- hadoop fs -mkdir /test
- hadoop fs -ls /       (the output should contain "/test")
```

Listing 39: Directory Test HDFS

@ Test: check how many datanodes are live. They should be exactly the number of instances contained in the file `workers`. The command for the test is:

```
- hdfs dfsadmin -report
```

Listing 40: Dfsadmin Report Test

4.4.2 Bad Formatting of HDFS

It can happen that the command 40 shows an output of 0 or less live datanodes that you expected. If your instances are active it could be probably due to some folders that need to be cleaned, especially when you reformat Hadoop because it seems the same Hadoop does not do it properly.
The folder $HADOOP_HOME/`data/datanode` needs to be emptied in each node, master included because it is also used as datanode.
The folder $HADOOP_HOME/`data/namenode` needs to be emptied only in the master.
The commands for these 2 operations are:

```
- rm -r $HADOOP_HOME/data/datanode       (for every node in the cluster)
- rm -r $HADOOP_HOME/data/namenode       (only for the master)
```

Listing 41: Hadoop Reformat Bug Solution

The 1st cited folder can be emptied in each datanode from the master taking advantage of SSH with the following command:

```
- ssh datanode* "sudo sh -c 'rm -r /home/ubuntu/hadoop/data/datanode'"
```

where the star (*) needs to be substituted with the index of the considered datanode. So this action need to be executed for each datanode.
However, this is not enough. The system folder `/tmp/` need to be emptied of the files created by Hadoop too. The command is:

```
- sudo rm -r /tmp/*
```

it could ask you for deleting some file that the system is using, so write choose "no" in that case.

This code need to be executed in every node of the cluster, and it can be done as well from the master for each node with the command:

```
- ssh datanode* "sudo sh -c 'sudo rm -r /tmp/*'"
```

where the star (*) needs to be substituted with the index of the considered datanode. So this action need to be executed for each datanode.

After these operations, HDFS can be re-formatted and the datanodes can be restarted with the commands 37 and 38.

4.4.3 YARN Usage

This short section is dedicated to the start of *YARN (Yet Another Resource Negotiator)*. The command is:

```
- start-yarn.sh
```

Listing 42: Start YARN

@ Test: check how many YARN nodes are active:

```
- yarn node -list
```

Listing 43: YARN Nodes Test

For a similar outcome the command `jps` can be used to see the running processes of the JVM, so Hadoop processes too.

To start both Hadoop nodes and YARN can be used the command:

```
- start-all.sh
```

but Apache suggests using `start-dfs.sh` and `start-yarn.sh`.

The respective commands to stop Hadoop nodes and YARN are:

```
- stop-dfs.sh
- stop-yarn.sh
- stop-all.sh        (to stop both)
```

4.4.4 Spark Usage

This section is dedicated to the start of the master and the slaves, and how to submit a script to execute it in a distributed way with YARN. The command to start the master is:

```
- ./spark/sbin/start-master.sh
```

Listing 44: Start Spark Master

The command to start the slaves is:

```
- ./spark/sbin/start-slaves.sh
```

Listing 45: Start Spark Slaves

Our cluster is ready to receive a script (in this guide we used a Python script) and the command to do it is:

```
- ./spark/bin/spark-submit --master yarn --deploy-mode client PATH_FILE
```

Listing 46: Spark Submit

The parameter `--num-executors X` with `X = number of executors` let you specify the number of nodes (executors) that will be used to execute the application. An example of usage could be:

```
- ./spark/bin/spark-submit --master yarn --deploy-mode client --num-executors 8 testSample.py
```

More options and specifications can be found in the Submitting Applications page of Spark.

If Spark shows the error `pyspark module not found` you could solve this problem setting 2 parameters in the SparkConf() object of your script. The 2 parameters can set this way:

```
.set('spark.yarn.dist.files','/home/ubuntu/spark/python/lib/pyspark.zip,/home/ubuntu/spark/python/lib/py4j-0.10.7-src.zip').setExecutorEnv('PYTHONPATH','pyspark.zip:py4j-0.10.7-src.zip')
```

Listing 47: Pyspark Module Not Found Error

that is, for instance, writing `SparkConf().set('spark.yarn.dist... etc)`

4.4.5 Spark Monitoring

Applications and Spark nodes can be monitored thanks to the Spark Web UI. First we need to modify the security group we created for our cluster and add our IP to let us enter in the Spark Web UI.
Select the option "Security Groups" of the group "NETWORK & SECURITY" in the menu on the left and modify the security group with the following steps:

1. Check the box next to the security group created during the instance launch. In this guide "hadoop" was used. It should appear a menu below the group with the tabs *Details / Inbound rules / Outbound rules / Tags*
2. Select "Inbound rules and left click on "Edit inbound rules"
3. Left click on "Add rule" on the screen that appears
4. Select "All Traffic" as *Type*. In *Source* select "My IP". AWS should find automatically your IP
5. Confirm with "Save rules"

Now we can use our browser and go to the link `http://PUBLIC_DNS_ADDRESS:PORT` where PUBLIC_DNS_ADDRESS is the public DNS of ous instance and PORT can be different values:

- 8080: you can see which nodes are active for Spark and other informations
- 8088: you can analyze the state of the submitted applications, if the application had problems or succeeded and the executors used during an application

More useful information can be found in the Monitoring and Instrumentation page of Spark.

5 Testing & Conclusions

This section is dedicated to show some results from testing an application that process files of different sizes and being executed by a cluster with a different number of nodes for each test.

The script used is `testSample.py`, which loads a file of byte64 random characters saved in HDFS as RDD, transform it to a list of characters, map each character (the key) to a counter and reduce the data by the key to count the occurrences of each character. Finally, prints each value of the RDD.
The content of `testSample.py` is:

```
from pyspark import SparkConf, SparkContext

conf = (SparkConf() \
         .set('spark.yarn.dist.files','/home/ubuntu/spark/python/lib/
pyspark.zip,/home/ubuntu/spark/python/lib/py4j-0.10.7-src.zip') \
         .setExecutorEnv('PYTHONPATH','pyspark.zip:py4j-0.10.7-src.zip'))
sc = SparkContext(conf = conf)

rdd = sc.textFile("hdfs://namenode:9000/sample.txt")

# split textfile into characters
split_chars = rdd.flatMap(list)

chars_count = split_chars.map(lambda c: (c,1)).reduceByKey(lambda c1,c2:
c1+c2)

for x in chars_count.collect():
        print(x)
```

Listing 48: testSample.py

Test	Instance Type	N. Instances	File Size	N. Executors	Execution Time
1	r5.large	32	9.4 GB	32	5mins, 1sec
2	r5.large	16	9.4 GB	16	7mins, 57sec
3	r5.large	32	2.1 GB	32	1mins, 51sec
4	r5.large	16	2.1 GB	16	2mins, 8sec
5	r5.large	12	2.1 GB	12	3mins, 21sec
6	r5.large	8	2.1 GB	8	3mins, 36sec
7	r5.large	4	2.1 GB	4	6mins, 35sec
8	r5.large	2	2.1 GB	2	12mins, 30sec
9	r5.large	1 (YARN)	2.1 GB	1	24mins, 27sec
10	r5.large	1 (local[*])	2.1 GB	1	23mins, 59sec

Table 1: Test Results of an Application Execution in a Cluster

The last two tests concerns the usage of `spark-submit.sh`. In the 9th test YARN has been used to execute the script, whereas in the 10th test local[*] has been used, which means the script is executed locally using all possible cores in the machine where the application is submitted, so our master.

In our tests we have seen that the executors are equal to the number of executors that we set plus 1, so using all instances we get all instances plus 1 as executors. In the table is shown that number of executors set using the parameter of `spark-submit.sh`.
We also noticed that the differences depend namely on the size of the job the cluster must do. Each job is divided in tasks and the number of tasks change with the size of the file processed by the script:

- 75 tasks for the file of 9.4 GB
- 17 tasks for the file of 2.1 GB

The time to process each task is nearly the same since all instances are of the same type and each task is assigned to an executor.
So each execution took the maximum of tasks at possible at a time: this can be called a round. The time of execution strictly depends on the number of rounds necessary to complete the job.
For instance, the 1st and the 2nd test have 75 tasks to complete.
In the 1st test 3 rounds are necessary because the cluster can use a maximum of 33 executors at a time, in the 2nd one 5 rounds are necessary beacuse the cluster can use a maximum of 17 executors at a time.
In the 1st case each round took around `1mins,40sec`, in the 2nd case each round took around `1mins,35sec`.
We can see that the time of execution of each round is quite the same for both the tests.

We also noticed that with this standard configuration not all cores of each node is used for the application, but just 1 for each node.
We modified the YARN configuration to use all the available cores, but the execution time was the same, so we did not add it to our tests and we processed all the tests with the standard configuration.
If you want to use all the available cores of each node this question of StackOverflow explains how. You will need also to modify the number of cores of each executor with Spark, which is 1 by default.