Big Data Project Report - Stefano Ziantoni 65171

Creating a standalone cluster on EC2 instances with Spark, saving the results on S3, with description of the development of an application in Python to query the results on S3 through Amazon Athena.

Summary:

0: Introduction:

Project Report for the 2020/21 Big Data Exam. In this report we will describe the process of creating a Standalone cluster in Spark managed with Terraform. The entire cluster construction process is described. Next, a Python script is run through the cluster which takes care of counting the occurrences per line of each word in a large text file using a map-reduce approach in Spark, which divides the work between the cluster machines. The computation results, as well as the input data, are saved on Amazon S3 storage.

In the next part of the paper the creation of a simple graphic interface in Django will be described, through which it will be possible to query the results of the computation using the SQL-Like queries executable in Amazon Athena on S3.

1: Cluster creation with Spark and Terraform:

1.1-VPC creation

The first step to take is to create a VPC (Virtual private Cloud) that allows us to start AWS resources in a customized virtual network.

- In the search bar of the AWS console, type “VPC” and select the first result.
- Click on “Launch VPC Wizard” and click on Select.
- Give the VPC a name and click on “Create VPC”.
- Once completed, click on "Subnets" in the menu on the left
- A public subnet called “Public Subnet” should appear. Now change the subnet name to make it recognizable.
- Select the subnet and click on “Actions” and select the item “Modify auto-assign IP settings”. Then select “Auto-assign IPv4”. This will allow you to communicate with instances via SSH because AWS will assign them public IPv4.
- We have therefore created the subnet within which we will create the instances.

1.2 Creating t2.micro instance with Spark

Primary Instance Creation:

From the AWS console select the “EC2” item. Then select the item "Instances" from the menu on the left. Then select "Launch Instances"

- For AMI (Amazon Machine Image) select Ubuntu 18.04 64-bit.
- Select the type of instance, in our case t2.micro
- Click on "Next: Configure Instance Details"
- Set the number of instances to 1, only the base instance will be created, and the others will be
  created through the AMI of the latter.
- Under "Network" choose the VPC that was created in the previous step.
- Click on "Next: Add Storage"
- Set 30 Giga of storage, the maximum allowed
- Click on "Next: Add Tags" and on "Next: Configure Security Group"
- Create a new security group via "Create a new security group" and set the name in the "Security group name" field
- Click on "Review and Launch" and then "Launch"
- Once clicked you will be asked to create a new key pair. Select the item "Create a new key pair" and set a name in "Key pair name", then click on "Download Key Pair" which will download the private key with the extension ".pem". Using the private key saved locally, it will be possible to connect to instances via SSH.

- Then click on "Launch Instances" to confirm.
- Once the instance has been created, go back to the “Instances” section, and give the instance a
  name by clicking on the pencil. In our case it will be "namenode".

Once this process is complete, you will need to change the Security Group. From the left menu of AWS select “Security Group” in Network & Security.
Once the security groups menu is loaded, select the custom security group you created earlier.
In the menu that appears at the bottom select "Inbound rules", click on "Edit inbound rules". Then click on “Add rule”.
Choose as option “All Traffic” and as Source select “Custom”, then write the CIDR chosen for the subnet (it should always be 10.0.0.0/24). Then save the rule with “Save rules”.

Connecting to the instance:

In the “Instances” menu click on the instance just created with the right text. Then click on “Instance State” in the menu, and then on “Start”.
Once the instance has started, right click on the instance again and then click "Connect" or "Connect".
Copy the string for the SSH connection.

Then open a Linux shell in the folder where the key was downloaded.
In the shell write "sudo" and then paste the connection string. (In the form: ssh -i key.pem ubuntu@PUBLIC DNS ADDRESS OF THE INSTANCE).
Once connected to the instance, open another shell in the same folder and type the command:

```
"scp -i 'key.pem' key.pem ubuntu@PUBLIC DNS ADDRESS OF THE
INSTANCE: /home/ubuntu/.ssh "
```

Close the shell once the key is sent and go back to the first shell, then type:

```
"chmod 400 /home/ubuntu/.ssh/chiave.pem"
```

Now, in the shell connected to the namenode instance write:

```
"sudo nano /etc/hosts"
```

And write in the file:

```
PRIVATE IP OF NAMENODE
PRIVATE IP OF NAMENODE datanode1
```

Save, then write:

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

And write in the file:

```
Host namenode
HostName namenode
User ubuntu
IdentityFile /home/ubuntu/.ssh/[KEYNAME].pem
Host datanode1
HostName namenode
User ubuntu
IdentityFile /home/ubuntu/.ssh/[KEYNAME].pem
```

Java and Spark set-up

Update the machine and install Java and Spark via the commands:

```
"sudo apt-get update && sudo apt-get dist-upgrade"

"sudo apt-get install openjdk-8-jdk"

"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"

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

Then change the environment variables**:**

```
"sudo nano /etc/environment"
```

And write in the file:

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:"

JAVA_HOME="/usr/lib/jvm/java-8-openjdk-amd64"
```

Then:

```
"source /etc/environment"
"nano /home/ubuntu/.profile"
```

And write in the file:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=$PATH:$JAVA_HOME/bin
```

And finally:

```
“source /home/ubuntu/.profile”
```

Then let's go to conclude the configuration of Spark:

```
“sudo nano spark/conf/spark-env.sh”
```

And write inside the file:

```
export SPARK_MASTER_HOST=namenode
export PYSPARK_PYTHON="/usr/bin/python3"
```

So let's create the slaves file that will be there to start all the slaves of the cluster with a single script.

```
“nano spark/conf/slaves”
```

Save the file without writing anything. In the case of this project, the namenode is master only and all the slaves for spark are created via Terraform. This last file will also be updated automatically via Terraform.

The last thing to do to finish configuring the instance is to install “pandas”. We will need this library for the testing phase.

Run the following commands:

```
“sudo apt install python3-pip”
“python3 -m pip install pandas”
```

1.3 Copy the AMI

At this point, after configuring the namenode instance, we will create an AMI copy, a snapshot of the instance that will allow Terraform to automatically recreate instances with the same namenode settings. It will be the instances used in the cluster.

To do this, from the EC2 menu right click on the namenode instance, then “Image” and “Create Image”. Choose a name for the image and click on “Create Image”.

Once the image has been created, it will be accessible from the menu on the left under "AMI" under "Images".

1.4 AWS CLI

We install the AWS CLI so that it can then be used by Terraform to access our resources:

```
"curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip"
-o "awscliv2.zip""

"unzip awscliv2.zip"

"sudo ./aws/install"

"rm awscliv2.zip"
```

Then open the AWS console. Click on your username at the top right and choose "My security credentials" Open the "Access keys" tab and select "Create new access key". A dialog box will open from which you can download the keys by clicking "Download key file".

Once the file has been downloaded, write to the shell:

```
"aws configure"
```

And enter the data one by one present in the downloaded file.
As region set that of your instance, as output format write "json".

1.5 Terraform configuration

At this point we can configure Terraform. In fact, it will not need to be present in the copy of the AMI because it will only be used on namenode.
Run the command:

```
"wget
https://releases.hashicorp.com/terraform/0.12.24/terraform_0.12.2
4_linux_amd64.zip"
```

Install unzip to extract the downloaded zip file

```
"sudo apt install unzip"
```

Then extract the contents of the zip file and remove it:

```
"unzip terraform_0.12.24_linux_amd64.zip"
"rm terraform_0.12.24_linux_amd64.zip"
```

So let's create the folder for Terraform and put the extracted folder inside it:

```
"mkdir Terraform"

"mv terraform/"
```

Now you will need to update the environment variables, type:

```
“sudo nano /etc/environment”
```

And write at the end of the PATH string added before:

```
“:/home/ubuntu/Terraform”"
```

Then:

```
“source /etc/environment”
```

At this point it will be necessary to create the Terraform configuration file, called "main.tf"

```
“nano Terraform/main.tf”
```

And write inside the file:

```
provider "aws" {
  profile = "default"
  region = "[REGION]"
}
resource "aws_instance" "testInstances" {
   ami = "[ID AMI]"
   instance_type = "t2.micro"
   subnet_id = "[ID SUBNET]"
   vpc_security_group_ids = [
      "[ID SECURITY GROUP]",
   ]
   count = [NUMBER OF INSTANCES TO CREATE]
}
resource "null_resource" "testInstances" {
   provisioner "local-exec" {
      command = join("_", aws_instance.testInstances.*.private_ip)
     interpreter = ["bash", "/home/ubuntu/Setup.sh", "[KEY NAME]",
     "[INDEX START]"]
   }
   provisioner "local-exec" {
      when = destroy
      command = [NUMBER OF INSTANCES TO CREATE]
     interpreter = ["bash", "/home/ubuntu/Clear.sh", "[INDEX
START]"]
      on_failure = continue
   }
}
```

As you can see, in this file there are a number of options to specify:

- [REGION]: region used (e.g. us-east-1)
- [AMI ID]: AMI ID which can be found in the AMI on AWS by clicking on it and opening the menu below.
- [SUBNET ID]: The ID of the VPC created initially, which can be found in the VPC menu by clicking on
  the VPC Created and opening the menu below.
- [ID SECURITY GROUP]: ID of the Security Group created previously, which can be found in the Security Group field of the AWS menu, by clicking on the security group and opening the menu below.
- [NUMBER OF INSTANCES TO CREATE]: The number of instances we want to use in the cluster.
- [KEY NAME]: the name of the key downloaded locally
- [INDEX START]: the starting index from which to name the new nodes, for example putting 2 and creating a cluster of three nodes, these will be called datanode2, datanode3, datanode4.

In addition, in the file, there are also two calls to external scripts, one of Cluster Setup, which sets the newly created instances automatically, and one of Cluster Cleanup which returns to the initial setting prior to creating the instances using Terraform.

Create the first file with the command:

"nano Setup.sh"

And write inside the file:

```
#!/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/spark/conf/slaves
             index=$((index + 1))
   done
```

The script takes care of configuring the ssh connection and the Spark configuration for each datanode.

Then we create the second file:

"nano Clear.sh"

And we write inside the file:

```
#!/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 echo " " > /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/*
```

Let's save and give permissions to the newly created files:

```
"chmod 777 Setup.sh"
"chmod 777 Clear.sh"
```

Finally, it will be necessary to create a third script, through which it will be possible to update the internal configuration of the datanodes once created:

```
"nano settingNewNodes.sh"
```

And write inside the file:

```
#!/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/.ssh/config | ssh -oStrictHostKeyChecking=no
       datanode$i "sudo sh -c 'cat >/home/ubuntu/.ssh/config'"
   done
```

Save the file and give permissions with the command:

```
"chmod 777 settingNewNodes.sh"
```

To run this script you will need to write:

```
"bash settingNewNodes.sh [STARTING INDEX] [NUMBER OF INSTANCES
CREATED]"
```

1.6 Cluster creation

At this point we will have everything set up to create the cluster.
After populating the main.tf file with the desired settings, navigate to the /
Terraform folder and type:

```
"terraform init"
"terrafrom apply"
```

Terraform will create instances according to the information specified in the main.tf file.
When terraform has finished creating the instances, go back to the home and run:

```
‶bash settingNewNodes.sh [INDEX START] [NUMBER OF INSTANCES CREATED]″
```

In case we want to delete the cluster created, it will be possible to execute "*terraform destroy* " in the terraform folder and thus bring everything back to the setting prior to the creation of the cluster.

2: Testing, results and saving on S3

2.1 Test introduction

In the case of this project, the cluster used is made up of 10 nodes. One master node and 9 slave nodes.
We chose not to use the master also as a slave because it created problems when submitting to Spark.

Then, once specified in Terraform 9 as the number of instances to be created and 2 as the starting index, the cluster was created using the Terraform scripts exposed above.

The test carried out on this cluster uses a text file as input, and through a mapreduce algorithm it counts the occurrences per line of each word, resulting in an array of pairs (word, N ° of lines in which it appears).

The input test file is saved on S3, and the computation result will also be saved as CSV on S3.

2.2 S3:

To use Amazon S3 storage the steps are quick and easy.

- In the search bar of the AWS console, type S3 and select the first result.
- Click on the text "Create Bucket"
- Give the bucket a name and click “Create Bucket” at the bottom of the page.
- Once the bucket is created, click on it and open it
- Next, click on “Upload” and follow the procedure for uploading the text file by selecting the “dump.txt” file in the Gitub repository after downloading it.

Once the text file is uploaded to S3, you will be able to use it in Spark.

In case the dump.txt file is too large, you can split it with the command:

```
‶split dump.txt -b [x]m –additional-suffix=.txt″
```

Where [x] indicates the megabytes in size of each splitted file.

2.3 Submit to cluster

The script that will be used is the “example.py” script present in the GitHub repository.

```
import […]

if __name__ == "__main__":
       data = []
       sc = SparkContext()
       file_ = sc.textFile("s3a://[MY BUCKET]/dump.txt")

       counts = file_.map(lambda line: [(i, 1) for i in set(line.split("
       "))]).flatMap(lambda x: x).reduceByKey(lambda x, y: x + y).collect()

       for i in range(0, 1000):
           print(counts[i])

       #df = pandas.DataFrame(counts, columns=['word', 'count'])

       #btw = df.to_csv(index = False).encode()

       #fs = s3fs.S3FileSystem(key=[AWS ACCESS KEY ID]', secret=[AWS SECRET KEY ID])

       #with fs.open('s3://[BUCKET TO SAVE RESULTS]/file_test.csv', 'wb') as f:
       #       f.write(btw)

       sc.stop()
```

The script uses sc.textFile to read the file from S3. In order to use "s3a" we will have to include this framework in the submit command that will be exposed shortly, and also we will need to export our AWS credentials so it will be necessary to type and send the commands:

```
export AWS_ACCESS_KEY_ID=[ AWS ID]
export AWS_SECRET_ACCESS_KEY=[ AWS SECRET KEY]
```

Both credentials can be found in the file downloaded earlier in the configuration of the CLI.
Once this is done it will be possible to send the script to the cluster in Standalone mode. To do this, first of all, you will need to start Spark with the commands to run in the home:

```
“./spark/sbin/start-master.sh”
“./spark/sbin/start-slaves.sh”
```

Once the commands have been executed, check that all slave nodes have been started correctly. To do this, first of all, let's go back to the AWS console and click on “Security Group”. We select our security group, and in the bottom menu select "Inbound" and add the rule "All Traffic - my IP". You will then be able to view the cluster started on the website:

http://ADDRESS_PUBLIC_DNS_ NAMENODE:8080

If the cluster is started successfully, you can submit the script with the command:

```
./spark/bin/spark-submit --packages com.amazonaws:aws-java-
sdk:1.7.4,org.apache.hadoop:hadoop-aws:2.7.7 --master
spark://namenode:7077 --executor-memory 1G  --total-executor-
cores 9 example.py
```

We note how the packages necessary to use s3a are included in the command, and also the options " total-executors-cores" which allows us to choose how many cores to use (in our case each machine has only one core so this number coincides with the number of machines among which to divide the work), and the option "Executor-memory", which instead allows us to specify how much memory to allocate to each core.

In the script, the commented part after printing the counts result is the part of code used to write the results on S3 once the computation is completed (obviously they are written only once and not in all tests). This step transforms the result into Dataframe and then uses the S3fs library to write a CSV file to S3. You will need to provide AWS CLI login credentials to the filesystem.
In order to save the results, it will be necessary to create a bucket on S3 to contain them, in the case of this project the bucket "resultsziantoni".

2.4 Results

The tests carried out see the use of different cluster configurations, with more or less nodes and more or less memory. Three files were used, one 500MB, one 1GB both portions of the third file from 1.5 GB, the results are:

Application ID	Name	Cores	Memory per Executor	Submitted Time	Duration	Size
app-20210601171042-0000	example.py	9	1024.0 MB	01/06/2021 17:10	29 s	500MB
app-20210601171133-0001	example.py	5	1024.0 MB	01/06/2021 17:11	36 s	500MB
app-20210601171237-0002	example.py	3	1024.0 MB	01/06/2021 17:12	45 s	500MB
app-20210601171408-0003	example.py	2	1024.0 MB	01/06/2021 17:14	60 s	500MB
app-20210601171607-0004	example.py	1	1024.0 MB	01/06/2021 17:16	1.7 min	500MB
app-20210601172146-0000	example.py	9	500.0 MB	01/06/2021 17:21	49 s	1GB
app-20210601172321-0001	example.py	5	500.0 MB	01/06/2021 17:23	1 min	1GB
app-20210601172508-0002	example.py	3	500.0 MB	01/06/2021 17:25	1.3 min	1GB
app-20210601172653-0003	example.py	2	500.0 MB	01/06/2021 17:26	1.8 min	1GB
app-20210601172915-0004	example.py	1	500.0 MB	01/06/2021 17:29	3.4 min	1GB
app-20210601161732-0000	example.py	9	1024.0 MB	01/06/2021 16:17	42 s	1GB
app-20210601161901-0001	example.py	5	1024.0 MB	01/06/2021 16:19	59 s	1GB
app-20210601162029-0002	example.py	3	1024.0 MB	01/06/2021 16:20	1.3 min	1GB
app-20210601162215-0003	example.py	2	1024.0 MB	01/06/2021 16:22	1.8 min	1GB
app-20210601162434-0004	example.py	1	1024.0 MB	01/06/2021 16:24	3.4 min	1GB
app-20210601183914-0005	example.py	9	500.0 MB	01/06/2021 18:39	50 s	1.35GB
app-20210601184031-0006	example.py	5	500.0 MB	01/06/2021 18:40	1.2 min	1.35GB
app-20210601184208-0007	example.py	3	500.0 MB	01/06/2021 18:42	1.7 min	1.35GB
app-20210601184418-0008	example.py	2	500.0 MB	01/06/2021 18:44	2.4 min	1.35GB
app-20210601184708-0009	example.py	1	500.0 MB	01/06/2021 18:47	4.5 min	1.35GB
app-20210601182601-0000	example.py	9	1024.0 MB	01/06/2021 18:26	49 s	1.35GB
app-20210601182719-0001	example.py	5	1024.0 MB	01/06/2021 18:27	1.2 min	1.35GB
app-20210601182857-0002	example.py	3	1024.0 MB	01/06/2021 18:28	1.7 min	1.35GB
app-20210601183107-0003	example.py	2	1024.0 MB	01/06/2021 18:31	2.5 min	1.35GB
app-20210601183406-0004	example.py	1	1024.0 MB	01/06/2021 18:34	4.6 min	1.35GB

3: Interface in Django for queries with AWS Athena on S3

At this point the creation of the Django environment will be described and the graphical interface and query management from Python to Amazon Athena will be described.

3.1 Django configuration

The Django environment and the application were developed locally in a Python 3.8 environment, obviously the same configuration can be reproduced on an EC2 instance with Python as the ones used so far. The decision to develop everything locally was made to contain the costs of the AWS Free Tier.

In general, both configurations will still be covered.

First of all, we need to check for Python and Pip.

In the EC2 instance we should have already configured them previously. Locally you need to run the following commands:

```
"sudo apt-get update && sudo apt-get -y upgrade"
"sudo apt-get install python3"
"sudo apt-get install -y python3-pip"
```

The next step is to install virtualenv:

```
"pip3 install virtualenv"
```

Django locally and on EC2:

Django will be installed via pip3:

We create from directory to contain the Django application:

```
"mkdir django-apps"
"cd django-apps"
```

And inside the newly created folder, we create a virtual environment, called "env":

```
"virtualenv env"
```

And we activate the environment with the command:

```
". env/bin/activate"
```

We can deactivate the environment when we want it with `". env/bin/deactivate"`
You will notice that the environment was started by the presence of the (env) tag at the beginning on the command line.

Once the environment is activated, the next step will be to install Django:

```
"pip install django"
```

And once installed we check the version:

```
"django-admin --version"
```

Now it will be possibly creating the project, let's create a Django project with the name "testsite":

```
"django-admin startproject testsite"
```

Let's go to the testsite folder and check what's inside:

```
“cd testsite”
“ls”
Output: [manage.py    testsite]
“cd testsite”
“ls”
Output: [__init__.py   settings.py    urls.py           wsgi.py]
```

- *__init__.py* serves as the entry point for the Python project.
- *settings.py* describes the configuration of the Django installation and lets Django know what settings are available.
- *urls.py* contains a urlpattern that maps URLs to views in the views.
- *wsgi.py* contains the configuration for the web server gateway interface. The Web Server Gateway Interface (WSGI) is the Python platform standard for deploying web servers and applications.
-

The last configuration step is to start the local server and view the Django sample website. The command to use is *runserver.*

Before starting the site, you will need to add the server's IP address to the ALLOWED_HOST list. List that we can find in the file ***setting.py.***

Starting from the /*django-apps:*

```
“nano testsite/testsite/setting.py”
```

And in the file look for ALLOWED_HOST:

```
"""
Django settings for testsite project.

Generated by 'django-admin startproject' using Django 2.0.
...
"""
...
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

# Edit the line below with your server IP address
ALLOWED_HOSTS = ['your-ip']
...
```

At this point, the first configuration difference:

In case the configuration is local (our case), instead of 'your-ip' it will be necessary to enter:

ALLOWED_HOST = ['*127.0.0.1*'] or ALLOWED_HOST = [' *']

If, on the other hand, you are configuring the application on the EC2 machine, you will need to enter:

ALLOWED_HOST = ['EC2_DNS_NAME']

And then you will need to enter the IP address or DNS name to ALLOWED_HOST in settings.py.

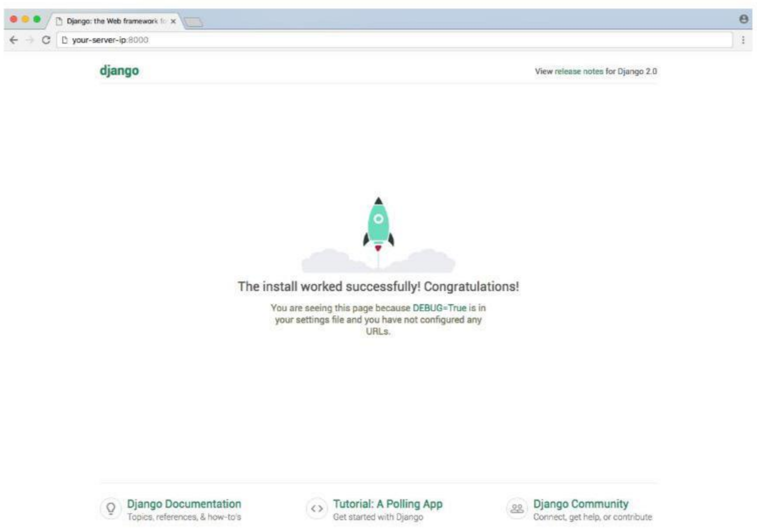
At this point, returning to the local configuration, make sure you are back in the folder it contains *manage.py*

```
“cd ~/django-apps/testsite/”
```

And we launch the website:

```
“python manage.py migrate”
“python manage.py runserver 127.0.0.1:8000”
```

At this point it will be possible to view the site at: http://127.0.0.1:8000/


Django: the Web framework
your-server-ip:8000
django
View release notes for Django 2.0
The install worked successfully! Congratulations!
You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.
Django Documentation
Topics, references, & how-to's
Tutorial: A Polling App
Get started with Django
Django Community
Connect, get help, or contribute

Instead, to start the site on the EC2 machine, on which obviously as locally we must also have activated the virtual environment, the command will be:

```
“python manage.py runserver 0.0.0.0:8000”
```

It will be possible to view the site at:: http: // PUBLIC_DNS_ADDRESS: 8000 /

However, when the instance is shut down, it will be necessary to repeat the previous configuration steps at the next start, updating the DNS addresses of the machine.

3.2 Amazon Athena

Amazon Athena is a service provided by Amazon AWS to analyse data stored on Amazon S3 through interactive queries that respect the SQL standard, it is a serverless service that you only pay for query time. To use Athena, you will need to access the AWS portal, create a table by specifying a data source in S3.

In the test script that was used there was a piece of code that took care of saving the data on a previously created S3 bucket, in CSV format.

```
import […]

if __name__ == "__main__":
       data = []
       sc = SparkContext()
       file_ = sc.textFile("s3a://[MY BUCKET]/dump.txt")

       counts = file_.map(lambda line: [(i, 1) for i in set(line.split("
"))]).flatMap(lambda x: x).reduceByKey(lambda x, y: x + y).collect()

       for i in range(0, 1000):
           print(counts[i])

       df = pandas.DataFrame(counts, columns=['word', 'count'])

       btw = df.to_csv(index = False).encode()

       fs = s3fs.S3FileSystem(key=[AWS ACCESS KEY ID]', secret=[AWS SECRET KEY ID])

       with fs.open('s3://[BUCKET TO SAVE RESULTS]/file_test.csv', 'wb') as f:
              f.write(btw)

       sc.stop()
```

The result of the map-reduce operation is transformed into a 2-column Dataframe, and then the s3fs library is used to access the bucket and write the csv file called "*file_test.cv* ".
In order to use this part of the script it will therefore be necessary to install the s3fs library.
Execute the command locally or on the ec2 machine:

```
"pip3 install s3fs"
```

Another necessary library is boto3, we run the command in the / testsite folder:

```
"pip3 install boto3"
```

If there are no pandas in the environment yet, install it with the command:

```
"pip3 install pandas"
```

After installing the library, submit the script to the cluster with the writing part on S3 not commented, and at this point, at the end of the computation, the file should appear in the S3 bucket.

If everything is done correctly, you will be able to configure Athena.

```
CREATE EXTERNAL TABLE IF NOT EXISTS sampledb.result_table (
  `Word` string,
  `Count` int
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES (
  'serialization.format' = ',',
  'field.delim' = ','
) LOCATION 's3://[BUCKET TO SAVE RESULTS]/'
TBLPROPERTIES ('has_encrypted_data'='false', "skip.header.line.count"="1");
```

With this query, once executed, the result_table table will be created in the sampledb database (the base database in Athena). The input file for the table is the file_test.csv previously saved in the bucket specified in BUCKET TO SAVE RESULTS.
Once the query has been executed, the table to be queried will have been created through our Python application with SQL query to Athena.

It must be said that it is also possible to query this table directly from the Athena interface, by writing an SQL query in one of the boxes present. The results are shown below the Query Panel.

3.3 Query Application

Once the Django environment and the Athena table to be queried have been configured, we will describe how the simple graphical interface for querying the data has been organized.

To use the files on GitHub in the django-apps folder, you must first copy the "s3_website" folder into the "djangoapps / testsite /" folder in your project

In s3_website we have all the files necessary for the functioning of the GUI which will be described shortly.

Once this is done, to allow the Django application to run the site, it will first be necessary to modify the urls file, from the django-apps folder we run:

```
“nano testsite/testsite/urls.py”
```

And we replace the content with:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('s3_website.urls'))
]
```

Let's save, then open the settings file:

```
“nano testsite/testsite/settings.py”
```

We add in INSTALLED_APPS:

```
INSTALLED_APPS = [
      […],
      's3_website.apps.S3WebsiteConfig',
      'testsite'
         ]
```

Always in the same file replace TEMPLATES with:

```
TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 's3_website/templates'],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',

'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

Then we close and save the file.

You should now be able to run and view the website:

```
"python manage.py runserver 127.0.0.1:8000"
```

Query:

By opening the "views.py" file in the "s3_website" folder, it will be possible to view the system for querying Athena.

The parameters of the functions that we will see shortly are defined in params:

```
params = {
    'region': '[YOUR REGION]',
    'database': 'sampledb',
    'bucket': '[BUCKET TO SAVE ATHENA RESULTS]',
    'path': 'Unsaved/2021/output',
    'query': 'SELECT * FROM ' + db + ' LIMIT 100'
}
```

We note that there are some parameters to define:

[YOUR-REGION]: the region of the ec2 machines

[BUCKET-TO-SAVE-ATHENA-RESULTS]: a new, third, bucket, which will need to be created to allow Athena to save the results. In fact, every time Athena executes a query, a .csv file with the results with a unique name is generated, and this file will be saved in the specified bucket. Thanks to this save made by Athena, it will be possible to recover this file in S3 after completing the query and view the results in tabular form on the website.

The three main functions are query, athena_query, athena_to_s3, and cleanup:

```
def query():
   session = boto3.Session()
   s3_filename = athena_to_s3(session, params)
   time.sleep(2)
   client = boto3.client('s3',
                         aws_access_key_id='[AWS ACCESS KEY]',
                         aws_secret_access_key=’[AWS SECRET ACCESS KEY]’, )
   # Create the S3 object
   obj = client.get_object(
      Bucket='[BUCKET TO SAVE ATHENA RESULTS]’
      Key='Unsaved/2021/output/' + str(s3_filename)
   )
   print('Unsaved/2021/output/' + str(s3_filename))
   table = pandas.read_csv(obj['Body']).to_html()
   return table


def athena_query(client, params):
   response = client.start_query_execution(
      QueryString=params["query"],
      QueryExecutionContext={
         'Database': params['database']
      },
      ResultConfiguration={
         'OutputLocation': 's3://' + params['bucket'] + '/' + params['path']
      }
   )
   return response


def athena_to_s3(session, params, max_execution=5):
   client = session.client('athena', region_name=params["region"])
   execution = athena_query(client, params)
   execution_id = execution['QueryExecutionId']
   state = 'RUNNING'

   while max_execution > 0 and state in ['RUNNING', 'QUEUED']:
      max_execution = max_execution - 1
      response = client.get_query_execution(QueryExecutionId=execution_id)

      if 'QueryExecution' in response and \
              'Status' in response['QueryExecution'] and \
              'State' in response['QueryExecution']['Status']:
         state = response['QueryExecution']['Status']['State']
         if state == 'FAILED':
            return False
         elif state == 'SUCCEEDED':
            s3_path = response['QueryExecution']['ResultConfiguration']['OutputLocation']
            filename = re.findall('.*\/(.*)', s3_path)[0]
            return filename
      time.sleep(1)

   return False



def cleanup(session, params):
   s3 = session.resource('s3', aws_access_key_id='[AWS ACCESS KEY]',
                         aws_secret_access_key=’[AWS SECRET ACCESS KEY]')
   my_bucket = s3.Bucket(params['bucket'])
   for item in my_bucket.objects.filter(Prefix=params['path']):
      item.delete()
```

You will need to specify the CLI keys in the clenup function and in the other functions.

To perform a query, the query() function is called. This function starts the boto3 session and calls the "athena_to_s3" function passing it the parameters in params and the session. The function will have to return the name of the file containing the results of the query if the query was successful.

The "athena_to_s3" function starts an Athena session in the region specified in the parameters. It then calls the "athena_query" function to which it passes the client and the set of parameters. The "athena_query" function sends the query to Athena, submitting the query specified in the 'query' field of params to the database specified in 'database'. It also defines the path in which to find the output and returns the response to the query.

In the second part of the "athena_to_s3" function, the function makes a maximum of 5 attempts to find the csv file that should contain the results of the query. If the file is found, it returns the name. The file name will be returned to the query() function, which through boto3 will open the bucket and read the results file, converting it into Dataframe and then into html with the .to_html() function, so that it can then be redirected to the Django site page .

Finally, the cleanup function can be used to clean up the result bucket when the space limit in S3 is near.

In general, the strategy chosen to query Athena is always this, what changes is the 'query' parameter in params.

For example:

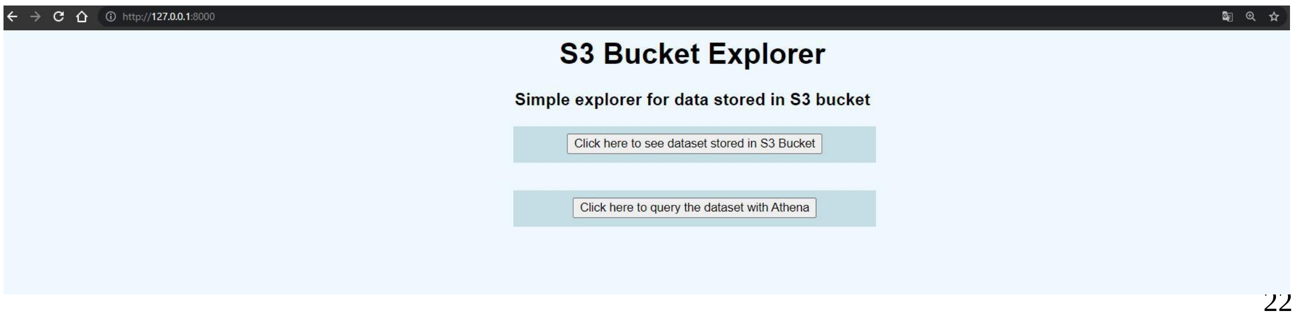
```
def getMoreCommon(request):
   params['query'] = 'SELECT * FROM ' + db + ' ORDER BY count DESC LIMIT 30'
   table = query()
   return render(request, 's3_website/athena.html', {'result' : table})
```

It is the function that acts on the button in the interface to obtain the most common words.
In the views.py file present in the repository it is necessary to fill in the fields delimited by "[]" with your configuration.

The rest of the configuration is normal Django code, with the definition of the urls in the urls.py file, the html templates in the templates folder in s3_website, and the entire site in general. Everything is easily understood by exploring the files in the GitHub repository.

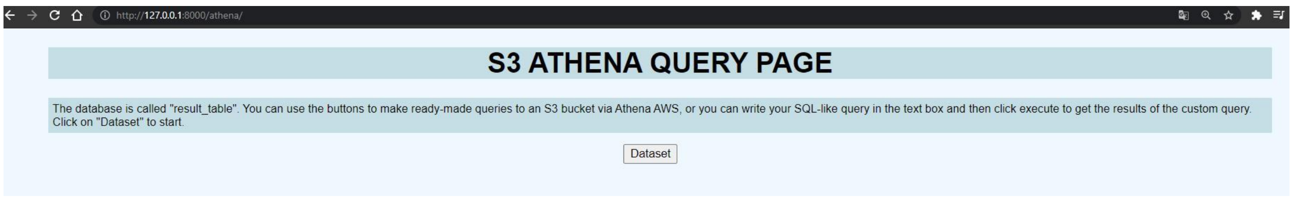
3.4 GUI description

The GUI is mainly composed of two pages, the first, the main one that appears at startup is:


http://127.0.0.1:8000
S3 Bucket Explorer
Simple explorer for data stored in S3 bucket
Click here to see dataset stored in S3 Bucket
Click here to query the dataset with Athena

By clicking on the first button it will be possible to view the entire database of results in a single page.

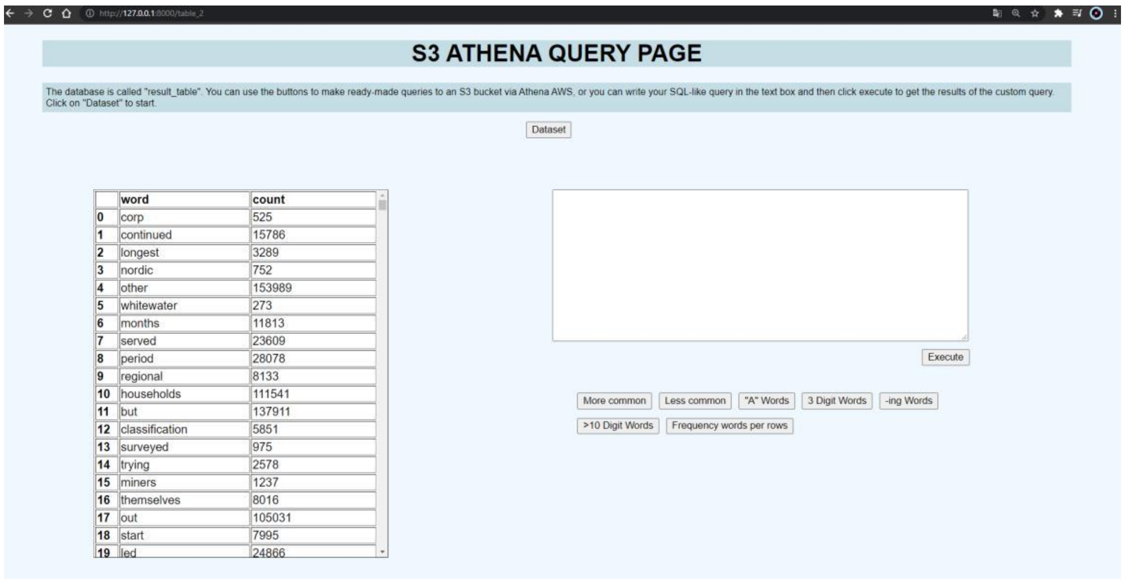
By clicking on the second button, however, the query page will open:



S3 ATHENA QUERY PAGE

The database is called "result_table". You can use the buttons to make ready-made queries to an S3 bucket via Athena AWS, or you can write your SQL-like query in the text box and then click execute to get the results of the custom query. Click on "Dataset" to start.

By clicking on Dataset the dataset will be displayed on the right and the buttons and the box to write the queries on the left:

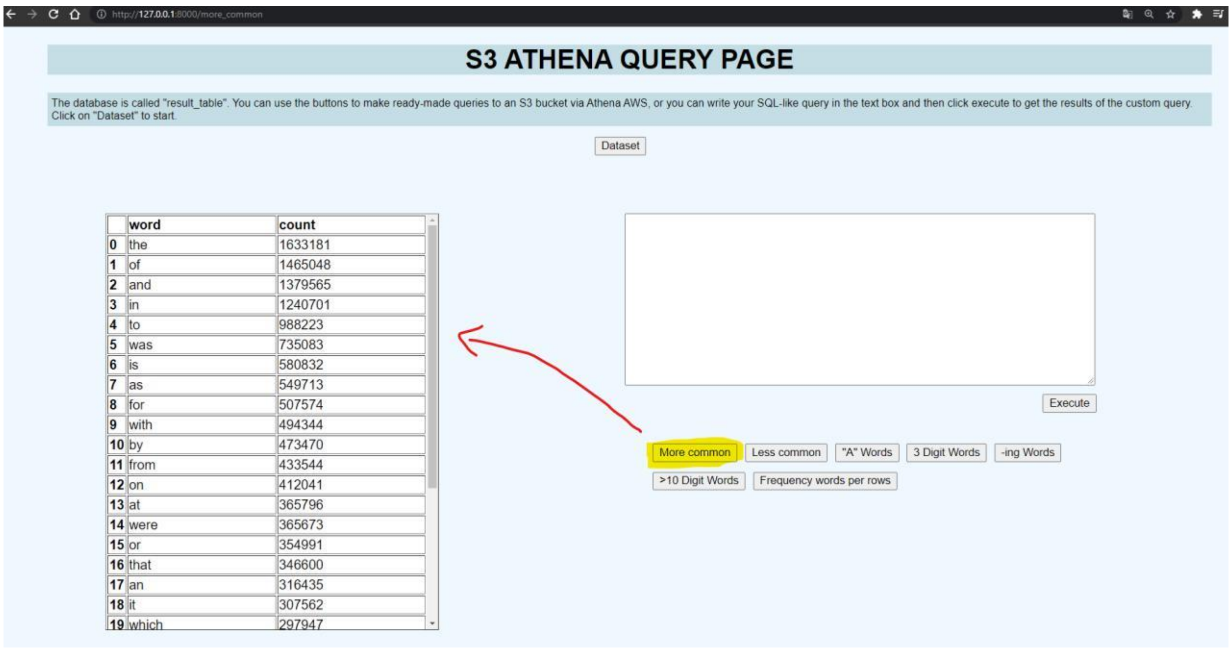


S3 ATHENA QUERY PAGE

The database is called "result_table". You can use the buttons to make ready-made queries to an S3 bucket via Athena AWS, or you can write your SQL-like query in the text box and then click execute to get the results of the custom query. Click on "Dataset" to start.

	word	count
0	corp	525
1	continued	15786
2	longest	3289
3	nordic	752
4	other	153989
5	whitewater	273
6	months	11813
7	served	23609
8	period	28078
9	regional	8133
10	households	111541
11	but	137911
12	classification	5851
13	surveyed	975
14	trying	2578
15	miners	1237
16	themselves	8016
17	out	105031
18	start	7995
19	led	24866

If one of the buttons below the textbox is clicked, the table on the left will be replaced with the results table, same thing if an SQL query is written in the Textbox and then execute is clicked:



S3 ATHENA QUERY PAGE

The database is called "result_table". You can use the buttons to make ready-made queries to an S3 bucket via Athena AWS, or you can write your SQL-like query in the text box and then click execute to get the results of the custom query. Click on "Dataset" to start.

	word	count
0	the	1633181
1	of	1465048
2	and	1379565
3	in	1240701
4	to	988223
5	was	735083
6	is	580832
7	as	549713
8	for	507574
9	with	494344
10	by	473470
11	from	433544
12	on	412041
13	at	365796
14	were	365673
15	or	354991
16	that	346600
17	an	316435
18	it	307562
19	which	297947