Universidad de Granada

MÁSTER EN INGENIERÍA INFORMÁTICA

Cloud Computing: Servicios y Aplicaciones

Practica 04 : Hadoop - Spark (BigData)

Autores

Hamada Bouhacida
177339003
bouhcidahamada@correo.ugr.es
hamadabouhcida34@gmail.com

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
----
Granada,Junio de 2021

Introduction :

As computer technology and services have evolved, we have realized the great value of the data they generate to help make applications more efficient. That is why, today, the vast majority of companies have tools capable of analyzing massive amounts of data in search of valuable information that helps them market their services. For this, it is necessary to have specialized platforms to apply Big Data techniques, such as Spark, which has a series of libraries with various functions related to machine learning.

Description of the problem :

Therefore, the objective of this practice is to solve a classification problem based on six variables assigned in a particular way to each student, making use of the MLLib library for data processing and learning that Spark has available. Also, since the dataset is extremely large, the above operations will be performed on Hadoop, using the clusters assigned to the students. For this we also have the HDFS file system with which Spark works to manage and operate with large volumes of data.

With this project we want to monitor the downtimes of the lines of a factory. It consists of 2 entities:

PlantMonitoring: It is in charge of reading the data from the lines and storing them in a database. The data that is stored is the moment in which a downtime occurs and its duration.

Notifier: Its function is to notify a list of contacts that a downtime has occurred and to notify again when the line is restored.

Architecture :

As architecture, an architecture based on microservices has been chosen, where each microservice corresponds to the entities described.

Languages and technologies used :

The project is developed using Python plus:

Flask for REST interface

Celery to manage events

MySQL and MongoDB as databases

Consul for distributed configuration

Prerequisites :

The Python versions compatible with the project are: Linux: -Minimum: 3.6 -Maximum: 3.8 and its development version

To be able to use the construction tool it is necessary:

Install it with:

pip install invoke

Install dependencies

pip install -r requirements.txt

In either case, it will be available

Construction tool :

buildtool: tasks.py

The construction tool used is Invoke.

To be able to use the construction tool it is necessary:

Install it with:

```
$ pip install invoke
```

Install dependencies

```
$ pip install -r requirements.txt
```

In either case, it will be available.

Four tasks have been configured. These are:

Install the required dependencies

```
$ invoke install
```

Install all the necessary dependencies for the project. If previously pip install -r requirements.txt has been run, it is not necessary to run this task.

You can check the dependencies used in the file requirements.txt

Run the unit tests

```
$ invoke test
```

Run all unit tests. For this, the Pytest framework has been used.

Run the coverage tests

```
$ invoke coverage
```

Run the coverage tests and store the results in a .coverage file. For the execution of these tests, the pytest-cov module provided by Pytest has been used.

Clean the files generated by the tests

```
$ invoke clean
```

Including .coverage file.

Lift the microservices

```
$ invoke start <ip> <port>
```

Build the microservices using Gunicorn, a WSGI HTTP server for Python. If the ip and the port where you want to link the service are not indicated by default, 0.0.0.0:5000 and 0.0.0.0:5051 will be set. > \ hello. This will return a Hello, World!

Stop microservices

```
$ invoke stop
```

Kill the process where the microservice was running.

At run time available tasks can be listed using invoke --list.

Databases :

PostgreSQL will be used to better manage and query the DATE and MongoDB data to store where the notifications should be sent, so that it is more convenient to manage the lists.

For the implementation of both databases, an ORM has been used to map the database objects to Python objects and mark the schemes that these must follow. For the two microservices the same pattern has been used: a class to define what the objects are like and that makes the connection to the database and another class that defines the operations with the database.

Performance evaluation :

Features: stress_test.yml

Taurus has been used to evaluate the performance of both microservices. The objective to be achieved is to reach 1000 requests / s with 10 concurrent users.

Create a virtual machine :

For the creation of the virtual machine we are going to use Vagrant plus Virtualbox. The first thing we have to do is download and install these two tools. Then to create the Vagrantfile we execute the vagrant init command in the root of the project.

You can consult the Vagrantfile file that contains explanatory comments about the creation of the virtual machine.

The virtual machine will be used to deploy previously owned docker containers.

Provisioning with Ansible :

Once we have the virtual machine configured, the next step is to do the provisioning. For this we will use Ansible.

In order to use Ansible we must download it first. In my case when using a version manager for Python it was installed with pip.

The next thing to do is create the Ansible configuration files:

A general configuration file ansible.cfg.

A file, inventory, with the name of the hosts to be used ansible_hosts.

The playbooks that we are going to use.

If we use Vagrant to automate provisioning, we must indicate where the file that we want to use as inventory is located. If not, Vagrant will create one by default.

After this in our Vagrantfile we have to indicate:

Which ansible hosts we are going to use.

Where are the playbooks that we are going to use.

Where is the inventory if we do not use the default.

Deployment to Azure :

The following steps have been followed for the deployment of the VM in Azure

Create a student account in Azure

Install Azure-CLI. For this, the following documentation has been followed.

We register the application in Azure. This will provide us with a series of keys to be able to create the VM from Vagrant.

```
$ az ad sp create-for-rbac
```

Get the ID of our azure subscription

```
$ az account list --query "[? isDefault] .id" -o tsv
```

We create the following environment variables to save the keys that the previous steps have provided us:

AZURE_TENANT_ID = tenant

AZURE_CLIENT_ID = appId

AZURE_CLIENT_SECRET = password

AZURE_SUBSCRIPTION_ID = SubscriptionId

We modified the Vagrantfile to use Azure as a provider. You can consult the file for more information. We will also have to add the necessary information to the inventory.

We install the Vagrant plugin vagrant-azure.

```
$ vagrant plugin install vagrant-azure
```

We install a box that Vagrant can use. This box doesn't really do anything.

```
$ vagrant box add azure https://github.com/azure/vagrant-azure/raw/v2.0/dummy.box --
provider azure
```

Finally we must execute vagrant up --privider = azure to create the machine.

Commands to provision the machine :

When we create the virtual machine with vagrant, the playbook provisioning.yml is executed automatically. What it does is update, install the necessary dependencies to be able to use Docker and Docker-Compose, download the necessary images and build the containers. Once the machine is built we can also run this playbook by doing:

```
$ vagrant provision
```

To start the services, the playbook start_services.yml has been made. To run it:

```
$ ansible-playbook -i ./provision/ansible_hosts ./provision/start_services.yml
```

To stop them there is the playbook stop_services.yml. To run it:

```
$ ansible-playbook -i ./provision/ansible_hosts ./provision/stop_services.yml
```

Local performance test :

For the vistual machine that has been deployed locally, the following configuration has been chosen:

2 CPUs

2 GB RAM

As the resources available are limited, we cannot expect a level of performance similar to that obtained by running the microservices locally.

Performance test for VM in Azure :

An attempt was made to perform a performance test for the virtual machine that is deployed in Azure. However, it has a default configuration that prevents DDoS attacks, so the benefits obtained when performing the tests are not very reliable.

The machine has the following configuration:

2 CPUs

8 GB RAM

They are hardware resources similar to those found in when the tests are passed locally (my computer has 4CPUs and 8GB of RAM).

Since it is not possible to deactivate the security option and there is no method that allows us to add secure IPs or the like, we cannot guarantee the level of services provided.