CREACIÓN DE UNA PLATAFORMA BIG DATA PARA INVESTIGADORES

APÉNDICE TÉCNICO – COMENTARIO DEL CÓDIGO

UNIVERSIDAD POLITÉCNICA DE VALENCIA

MÁSTER EN BIG DATA ANALYTICS 2015-2016

Trabajo de Fin de Máster

Antonio Eslava Polo

Dirección

Dr. Francisco M. Rangel Pardo

CREACIÓN DE UNA PLATAFORMA BIG DATA PARA INVESTIGADORES
APÉNDICE TÉCNICO

INTRODUCCIÓN

La plataforma creada está formada por una pieza principal y cuatro auxiliares:

El elemento principal es el programa de creación de clúster y cálculo de matrices: *calcCorrMatrix.py*.

Los auxiliares son:

- Imagen de AWS (AMI) a partir de la cual se crean los clústeres.
- Programa de generación de la matriz de apoyo: *prepareMatrix.py*.
- Programa de cálculo de la matriz: *corrsp.py*.
- Programa de eliminación de instancias que, por error, han permanecido activas: *instancesTerminate.py*.

En este apéndice se detalla cada uno de estos elementos. Para los programas, se comenta el código, justificando cada parte. Para la AMI, se detalla su proceso de creación.

PROGRAMA DE CREACIÓN DE CLÚSTER: CALCCORRMATRIX.PY

El programa comprende el ciclo completo: desde la creación del clúster hasta la finalización de las instancias, una vez calculada la matriz.

Se compone de varios bloques de tareas principales, cada uno con subgrupos de tareas:

1. Preparación del entorno.
   1.1. Carga de librerías, variables y objetos a utilizar.
   1.2. Verificación de existencia de la matriz de entrada.
   1.3. Creación de grupo de seguridad AWS.
2. Inicialización del clúster.
   2.1. Creación de los nodos.
   2.2. Configuración de accesos entre los nodos (SSH).
   2.3. Lanzamiento del clúster.
3. Cálculo de la matriz.
   3.1. Copia de scripts y matriz al máster y el resto de nodos.
   3.2. Preparación de la matriz auxiliar.
   3.3. Cálculo de la matriz.
   3.4. Depósito de la matriz en el ordenador local.
4. Eliminación de las instancias.

A continuación, se detallan cada una de estas tareas:

PREPARACIÓN DEL ENTORNO

Son las tareas generales previas a la creación de clúster. Además de las habituales cargas de módulos Python e inicialización de objetos como variables y listas, se realizan dos acciones importantes: **Verificación de existencia de la matriz de entrada** y **Creación del grupo de seguridad AWS**.

Carga de librerías, variables y objetos a utilizar

Junto a las librerías estándar, como *os*, *sys*, *time*y *datetime*, se utilizan ***boto3*** y ***paramiko***. ***Boto3*** permite la manipulación programática de todos los objetos de AWS, mientras que ***Paramiko*** permite la copia de archivos a los equipos de la nube mediante sftp, o el acceso mediante ssh.

```
import os

import sys

import time

import datetime



import boto3

import paramiko
```

También al inicio se definen los parámetros que el usuario va a poder modificar, y que definen los aspectos del clúster y de la matriz.

```
n = 10

ImageId = 'ami-33d00e25'

InstanceType = 'm3.xlarge'

Zone = "us-east-1c"

Matrix = "matrix.csv"
```

n es el número de nodos del clúster.

> El mínimo es de **3** y el máximo es de **20**. Para más de 20 nodos, es necesaria una solicitud a AWS, que no cumpliría con los requisitos de sencillez de uso de la herramienta.
>
> Más aún, si el usuaio dispone de otros nodos de EC2 activos, el máximo de 20 instancias simultáneas incluiría a estos nodos. Es por esto que, en el archivo de defecto, se ha propuesto un número de 10. Este número permite aprovechar suficientemente la potencia de cálculo en clúster, pero sin sobrepasar fácilmente la cifra máxima de instancias.
>
> Este parámetro puede ser modificado por el usuario a voluntad. Siendo consciente de que un mayor número de nodos incrementará la velocidad de cálculo, pero también el coste.

ImageId

Es el identificador de la imagen AMI que se utiliza para crear las instancias AWS. Si el usuario crea su propia AMI, deberá proporcionar su identificador. Si el usuario no desea crear su propia AMI, que sería lo adecuado por el requerimiento de sencillez, basta con que remita un mensaje de correo a la dirección indicada en el archivo README.TXT, indicando su identificador de cuenta de AWS. Inmediatamente será autorizado a utilizar la AMI creada por el autor. Además, una vez registrado el correo, será informado en caso de cambio de AMI por mejoras.

La utilización de la AMI permite incorporar cualquier mejora de una forma muy sencilla. Basta con crear la nueva imagen con las modificaciones e informar a los usuarios de su identificador, para que ellos mismos modifiquen el valor de la variable.

InstanceType

Es uno de los parámetros más importantes en cuanto a rendimiento y coste. La instancia puede ser creada sobre sistemas con diferente capacidad de CPU, memoria o RAM. Mayores capacidades suponen mayores costes por segundo, pero menores tiempos. Un análisis permite comprobar que las mejores configuraciones son las que equilibran ambos factores.

Los tipos de instancia con los que se ha probado la herramienta son:

m3.medium, m3.large, m3.xlarge.

No se ha probado con instancias más potentes y costosas, por economía durante el estudio, pero podrían ser utilizables.

Hay que considerar que, aunque se obtengan menos tiempos de cómputo son instancias más potentes, la construcción del clúster tarda entre cinco y diez minutos. Este tiempo no está siendo utilizado para computación y penaliza el coste en las instancias de mayor precio.

Zone

Puede ser utilizada cualquiera que desee el usuario. Se proponen las de EEUU Este (*us-east-1c*, por ejemplo) por ser las de menos coste. Esto es posible porque no se almacenan datos personales que podrían hacer necesario el alojar los servidores en espacios de la Unión Europea.

Matrix

Se recomienda que, por defecto, la matriz se denomine *matrix.csv*. En atención a la posibilidad de disponer de varias matrices sobre las que realizar los cálculos, se ha introducido esta posibilidad de parametrización. De este modo, no sería necesario modificar los nombres de las matrices, sino este parámetro.

Dos objetos Python que se inicializan en este momento son la lista “nodes” y el archivo “cluster.txt”. La lista “nodes” irá incorporando información de cada una de las instancias, a medida que se vayan creando. El archivo “cluster.txt” persistirá en disco esta lista de nodos. De este modo, el usuario dispone siempre del listado de

instancias más reciente, con sus direcciones DNS e IP, en caso de que desee realizar alguna averiguación sobre ellas:

```
# 1. Creation of file 'cluster.txt'
if os.path.exists('cluster.txt'):
    os.remove('cluster.txt')
f_cluster = open("cluster.txt", "a")


# 2. Creation of list 'nodes'
nodes = []
```

Otro objeto Python a crear es la conexión *boto3*.

```
ec2 = boto3.resource('ec2')
session = boto3.Session()
client = session.client('ec2')
```

boto3 es la biblioteca de acceso a AWS desde Python. Su estructura replica la de AWS, por lo que, para disponer de acceso a crear instancias EC2, son necesarios los siguientes pasos:

- Creación de un recurso, en este caso, 'ec2' (*ec2 = boto3.resource('ec2')*). Esta acción inicializa el entorno para poder tratar con el servicio de AWS indicado en el recurso. Para acceder a almacenamiento S3, por ejemplo, sería necesario crear otro recurso de este tipo.
- Inicialización de una sesión. Es simplemente la inicialización del entorno de conexión (*session = boto3.Session()*). Aún no se ha indicado el tipo de servicio ni el recurso al que asociarla.
- Asociación de conexión y recurso. Se indica ya que la conexión es una conexión al recurso creado, y el tipo de conexión, en este caso, *client*. (*client = session.client('ec2')*).

Verificación de existencia de la matriz de entrada

Es este un punto muy importante, por lo que se recoge aquí de modo separado. En cualquier programa es importante verificar al inicio que se dispone de los elementos necesarios. En este caso, es crucial, pues no sería aceptable lanzar los cinco o diez minutos de creación del clúster, con los costes económicos que representa, para encontrarse con que, en el ordenador que lanzó el programa, no existe una matriz con el nombre indicado.

```
if not os.path.exists('matrix.csv'):
    print("There is not any 'matrix.csv' file. Please provide one")
    sys.exit()
```

Creación de grupo de seguridad AWS

Cada instancia de AWS debe estar unida a un grupo de seguridad. Es éste un grupo de reglas de acceso, típicas de los firewalls o ACLs, que indican qué protocolos de comunicación pueden recibir sesión desde el exterior de la instancia, o establecerla hacia ese exterior. Las reglas regulan, no sólo el protocolo, sino la dirección del sistema externo.

La aproximación propuesta no necesita reglas de filtrado de entrada o salida (permite el acceso sin restricción, sólo controlado por usuario/clave). Pero, si no se especifica un grupo de reglas, AWS aplica, por defecto, otro grupo, más restrictivo.

Además, estos grupos se encuentran asociados a la cuenta del usuario AWS. Esto lleva a generar, para cada cluster, un grupo de usuario de acceso general. Ha sido denominado "BouncyHadoop", y se comprueba su existencia al inicio de la creación. Si ya existe (como en el caso en que el usuario haya ejecutado el programa otras veces), el proceso es obviado.

```
# 4. Security group creation.
try:
   SecGroup = client.create_security_group(
      DryRun=False,
      GroupName='BouncyHadoop',
      Description='All Traffic'
   )

   SecGroupId = SecGroup['GroupId']

   client.authorize_security_group_ingress(
      DryRun=False,
      GroupId=SecGroupId,
      IpProtocol='-1',
      FromPort=-1,
      ToPort=-1,
      CidrIp='0.0.0.0/0')

   print("\nSecurity group 'Bouncy Hadoop', created with Id "\
        , SecGroupId,"\n")
```

```
except:
    pass
```

Los valores "-1" para protocolo y puerto, indican que se permite cualquiera de ellos, mientras que la máscara '0.0.0.0/0' en la dirección, indica que el clúster puede ser accedido desde cualquier dirección IP.

INICIALIZACIÓN DEL CLÚSTER

Para cada uso de la herramienta, las instancias AWS son creadas de nuevo. Esta es la parte estructural del proceso, que consta de tres etapas principales y claramente diferenciadas:

Primera, la creación de los nodos AWS, en la cantidad y del tipo que se indica en las variables de inicio.

Segunda, la configuración de accesos. Es fundamental conseguir que el máster del clúster pueda acceder a los nodos secundarios (*slaves*) de forma automática. Este es un concepto sencillo que tiene una implementación compleja. Los nodos se comunican por SSH, y conseguir esta automatización requiere un complejo procedimiento. Por este procedimiento, se regeneran claves SSH de cada nodo, y se incluyen en los ficheros de control que son distribuidos a los nodos. Todo ello, según el orden al que obliga la configuración de SSH.

La tercera etapa es el lanzamiento del clúster. Hay que considerar que la imagen AMI que se clona para obtener las instancias, ya dispone de todo el software de Hadoop y Spark. Basta con recrear el fichero *nodes*, donde se relacionan las direcciones IP de los nodos que constituyen el clúster.

Creación de los nodos

En esencia, se trata de un bucle que repite n veces (siendo n el número de nodos) la creación de nodos AWS. En realidad, el nodo no se crea inmediatamente, sino que se "reserva". Unos segundos o minutos después, es cuando el nodo es activado. A partir de ese momento, ya dispone de dirección IP pública y puede ser accedido desde la herramienta.

El proceso de creación consta, a su vez, de tres partes:

- Reserva de las instancias.
- Obtención de la dirección IP pública y aplicado de etiquetas.
- Creación del fichero con la lista de instancias.

Éste es el proceso de reserva de instancias:

```
"""
Instance creation loop.
"""
for i in range(0, n):
    print("Instance:   ", i+1)
    Instance = client.run_instances(
```

```
        DryRun=False,
        ImageId=ImageId,
        MinCount=1,
        MaxCount=1,
        SecurityGroups=['BouncyHadoop'],
        UserData='ls > ls.txt',
        InstanceType=InstanceType,
        Placement={'AvailabilityZone': Zone}
        )

    InstanceId = Instance['Instances'][0]['InstanceId']
    InstanceIp = Instance['Instances'][0]['PrivateIpAddress']
    InstanceDns = Instance['Instances'][0]['PrivateDnsName']
    print("InstanceId: ",InstanceId)
    print("Private IP: ",InstanceIp)
    print("Public DNS: ",InstanceDns)
    print("")

# After having every instance's reservation data, add them to the
#  'nodes' list.
    nodes.append([InstanceId,InstanceIp,InstanceDns])
```

Se trata de un bucle que repite *n* veces la función *client.run_instances()*. Esta función devuelve cada vez una lista Python (denominada como *Instance*). De esta lista, para cada reserva, se obtienen:

- El identificador de la instancia (denominado como *InstanceId*).
- La dirección IP privada de la instancia (denominada como *PrivateIpAddress*).
- El nombre DNS privado de la instancia (denominado como *InstanceDns*).

Tras la iteración de cada nodo, se añaden estos datos a la lista *nodes*, donde se persistirá la información sobre el clúster:

(*nodes.append([InstanceId,InstanceIp,InstanceDns]*)

Obtención de la dirección pública y aplicación de etiquetas:

```
# Tags application to every node.
# The tag is "Project" and its value, "BouncyHadoop".
```

```
for node in nodes:
# Wait until the instance is running
    count = 0
    while True:
        if count < 10:
            blank = " "
        else:
            blank = ""
        if count%5 == 0:
            print("Instance starting, " + blank + str(count) + " seconds")
        count += 1
        time.sleep(1)
        if ec2.Instance(InstanceId).state['Name'] == 'running':
            print("Instance started")
            time.sleep(2)
            break

    ec2.Instance(node[0]).create_tags(\
        DryRun=False,\
        Tags=[{'Key': 'Project', 'Value': 'BouncyHadoop'}])
    print("Created label (Project = BouncyHadoop) for instance "\
          + str(nodes.index(node)+1))
    InstanceInfo = client.describe_instances(
        DryRun=False,
        InstanceIds=[node[0]]
        )
    InstanceId = node[0]
    InstanceIp = node[1]
    InstanceDns = node[2]
    InstanceIpPub = InstanceInfo['Reservations'][0]['Instances'][0]\
                    ['PublicIpAddress']
```

```
nodes[nodes.index(node)].append(InstanceIpPub)
print("Public IP of Instance " + str(nodes.index(node)+1) + ": ",InstanceIpPub)
```

Como se ha comentado, la primera acción de AWS, al recibir la solicitud de una instancia, es su reserva, pero no su puesta en marcha inmediata.

Para cada instancia solicitada, AWS devuelve un identificador (ID). Con este identificador se hará referencia, en lo sucesivo, a la instancia desde el programa.

Una vez solicitados, en el paso anterior, los *n* nodos, cuya información se encuentra en la lista *nodes*, se ha proporcionado a AWS tiempo para poder iniciarlos. El paso siguiente consiste en recorrer la lista *nodes* e intentar obtener la información del nodo, si ya se encuentra activo.

AWS puede tardar algún tiempo más en iniciar la primera instancia. Para confirmar que está activa, se ejecuta, dentro de un bucle *while*, la instrucción

if ec2.Instance(InstanceId).state['Name'] == 'running':

Si el nodo ya se encuentra activo, el bucle se detiene. En caso contrario, continúa hasta que aquél alcanza el estado de *running*.

Una vez iniciada la instancia, se le aplican las etiquetas (*tags*) que la relacionan con el proyecto. Estas etiquetas son atributos de la instancia que permitirán:

a) Obtener sus costes económicos por separado.

b) Identificar a una instancia, entre todas las del usuario, como perteneciente a este cluster.

Las etiquetas AWS adoptan la forma de pares, (*Key*, *Value*). Para este proyecto, la clave *Key* es *Project*, y la clave *Value* es *BouncyHadoop*.

```
ec2.Instance(node[0]).create_tags(\
    DryRun=False,\
    Tags=[{'Key': 'Project', 'Value': 'BouncyHadoop'}])
```

Una vez asignadas las etiquetas, se obtiene la dirección IP pública del nodo. Esta dirección no existe al hacer la reserva, y sólo es posible obtenerla una vez activa la instancia.

```
InstanceIpPub = InstanceInfo['Reservations'][0]['Instances'][0]\
    ['PublicIpAddress']
```

La dirección se incluye en la entrada de la lista *nodes* para esa instancia.

```
nodes[nodes.index(node)].append(InstanceIpPub)
```

Es importante realizar una precisión sobre las direcciones de las instancias en AWS.

Cada instancia tiene dos pares de direcciones: el privado y el público. Ambos están formados por una diección IP y una dirección DNS. Las direcciones privadas sólo son accesibles desde el interior de la red de AWS, mientras que las públicas lo son sólo

desde el exterior. Es por esto que, para esta herramienta, se utilizarán las direcciones privadas en todas las comunicaciones que los nodos tengan entre sí (fichero *hosts*, accesos SSH, …). En cambio, se utilizarán las direcciones públicas, principalmente la dirección IP, desde sistemas situados en el exterior de la red de AWS. Las utilizará el programa en las fases sucesivas, y las utilizará el usuario si, en algún momento, desea conectarse a alguna de las instancias mediante SSH.

Configuración de accesos

Para que las instancias constituyan un clúster, es necesario, y al mismo tiempo suficiente, con que las direcciones de los equipos que los forman, se encuentren en el archivo *slaves*, de la configuración de Hadoop.

Al mismo tiempo, es condición necesaria que el máster pueda acceder a los nodos por SSH sin necesidad de intervención humana.

Esto requiere una aclaración:

- Cuando desde un ordenador Linux se intenta acceder a otro mediante SSH, éste pide la clave de acceso. Para evitarlo, es necesario que, en un fichero denominado *known-hosts* del sistema accedido, se encuentre la clave pública del sistema que accede. De este modo, se expresa la confianza en el equipo remoto, evitando la petición de la clave.
- Cuando un ordenador Linux intenta acceder a otro mediante SSH, la primera vez indicará que ese ordenador no es conocido, y pedirá al usuario que guarde su identificador en el archivo *authorized_keys*. Para evitar esto, es necesario que exista ya la clave pública del sistema remoto en el ordenador que se conecta.

Además, cada vez que se crea el clúster, las direcciones IP de los equipos son diferentes. Para hacer referencia a ellos en los archivos auxiliares como los descritos, es necesario hacerlo mediante nombres DNS. Los nombres elegidos son *master*, para la primera instancia creada, y *slave1*, *slave2*, *slave3*, … hasta *slave(n-1)*, para el resto de instancias.

Este problema se resuelve creando un archivo *hosts* con las direcciones y nombres de las instancias, a partir de la información guardada en la lista *nodes*.

Estos archivos se crean en fases sucesivas. Se guarda una copia en la carpeta local, que será posteriormente distribuida a las instancias del clúster.

Creación del fichero *slaves.txt*. Un bucle recorre la lista *nodes* y obtiene la dirección IP privada.

```
# 1. Creation of file 'slaves.txt'
  # Using the element different of first in 'nodes' list
if os.path.exists('slaves.txt'):
   os.remove('slaves.txt')
f_slaves = open("slaves.txt", "a")
```

```
for node in range(0, len(nodes)):
    f_slaves.write(nodes[node][1] + '\n')


f_slaves.close()
print("")
print("'slaves.txt' file created")
```

Creación del archive *hosts.txt*. Se recorre el bucle creando el archivo con las entradas *master* y *node1, node2*, etc.

```
# 2. Creation of file 'hosts.txt'
    # With every element in 'nodes' list
    # For 'master', the name is 'master'
    # For 'slaves', the name is 'slave' plus a sequential number (slave1)
if os.path.exists('hosts.txt'):
    os.remove('hosts.txt')
f_hosts = open("hosts.txt", "a")
f_hosts.write('127.0.0.1 localhost' + '\n')
f_hosts.write(nodes[0][1] + ' master\n')


for node in range(1, n):
    f_hosts.write(str(nodes[node][1]) + ' slave' + str(node) + '\n')
f_hosts.close()
print("'hosts.txt' file created")
```

A continuación, se crean los archivos *known_hosts* y *authorized keys*. Pero es necesario un paso previo. Ambos archivos utilizan las claves públicas SSH de las instancias. La imagen a partir de la cual se crean ya tiene una clave pública. Y no es admisible que dos sistemas diferentes tengan la misma clave pública.

Por eso, antes de crear los archivos, es necesario regenerar las claves SSH de cada instancia, para que sean diferentes.

```
# 1. Recreation of SSH keys for each node
for node in range(0, n):
    if node == 0:
        print("Regenerating SSH keys for master")
    else:
```

```
        print("Regenerating SSH keys for slave" + str(node))
    instance = paramiko.SSHClient()
    instance.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    count = 1
    while True:
        try:
            instance.connect(nodes[node][3],username="hduser", \
                        password="hduser",timeout=1)
            break
        except:
            if count%5 == 0:
                print("Connecting to host, " + str(count) + " seconds")
            count += 1
            time.sleep(1)
            pass
    instance.exec_command("rm /home/hduser/.ssh/id_rsa")
    instance.exec_command("ssh-keygen -f /home/hduser/.ssh/id_rsa -q -N ''")
    instance.close()
print("SSH keys regenerated")
```

En este bucle, se comienza a utilizar la técnica de ejecución remota que va a ser la base de la programación en las siguientes fases.

Una vez creados los nodos, casi toda la actividad va a ser realizada en el master. Para ello, se utiliza la biblioteca *Paramiko*. Ésta es una biblioteca de acceso SSH para Python que permite dos tipos de acciones:

- Ejecutar órdenes en el sistema remoto (equivalente a hacer *login* mediante SSH con un usuario humano, y teclear las órdenes).
- Enviar y recibir archivos en los sistemas remotos, mediante la utilidad *scp*.

Para utilizar Paramiko, se crea un objeto cliente:

```
instance = paramiko.SSHClient()
```

al que se hace referencia para crear una conexión, en la que se incluye usuario y clave:

```
instance.connect(nodes[node][3],username="hduser", \
                        password="hduser",timeout=1)
```

Y, una vez active la conexión, se ejecuta cualquier orden del sistema operativo mediante *instance.exec_command("<orden del sistema>").*

Una vez regeneradas las claves, se crea el fichero *known-hosts*, recorriendo la lista nodes y utilizando Paramiko:

```
# 2. Creation of the 'known-hosts' archive in master
  # Download to local station as 'known_hosts.txt'
print("Creating archive 'known-hosts.txt' ")
instance = paramiko.SSHClient()
instance.set_missing_host_key_policy(paramiko.AutoAddPolicy())
instance.connect(nodes[0][3],username="hduser",password="hduser")
for node in range(0, n):
    instance.exec_command("ssh-keyscan " + nodes[node][1] \
                    + ">> ~/.ssh/known_hosts")
instance.close()


transport = paramiko.Transport((nodes[0][3],22))
transport.connect(username = "hduser", password = "hduser")
sftp = paramiko.SFTPClient.from_transport(transport)
sftp.get('./.ssh/known_hosts','./known_hosts.txt')
sftp.close()
transport.close()
print("Archive 'known_hosts.txt' created")
```

Inmediatamente después, se crea el fichero *authorized_keys.txt*, con un procedimiento similar:

```
# 3. Creation of the archive 'authorized_keys.txt'
print("Creating archive 'authorized_keys.txt'")
if os.path.exists('authorized_keys.txt'):
    os.remove('authorized_keys.txt')
for node in range(0, n):
        if node == 0:
            print("Connecting to master")
        else:
```

```
        print("Connecting to slave" + str(node))
    transport = paramiko.Transport((nodes[node][3],22))
    count = 1
    while True:
        try:
            transport.connect(username = "hduser", password = "hduser")
            break
        except:
            if count%2 == 0:
                print("Connecting to host, " + str(count) + " seconds")
            count += 1
            time.sleep(1)
            pass
    sftp = paramiko.SFTPClient.from_transport(transport)
    sftp.get('.ssh/id_rsa.pub','id_rsa.txt')
    os.system("type id_rsa.txt >> authorized_keys.txt")
    transport.close()
print("Archive 'authorized_keys.txt' created")
print("")
```

Una vez creados todos los archivos necesarios, son depositados en cada instancia, mediante las órdenes *sftp.put*, de Paramiko.

```
# 4. Distribution of cluster files to each node
print("Copying cluster files to master")
transport = paramiko.Transport((nodes[0][3],22))
transport.connect(username = "hduser", password = "hduser")
sftp = paramiko.SFTPClient.from_transport(transport)
sftp.put('./hosts.txt','/etc/hosts')
sftp.put('./authorized_keys.txt','./.ssh/authorized_keys')
sftp.put('./slaves.txt','/usr/local/hadoop/etc/hadoop/slaves')
sftp.close()
transport.close()
```

```
for node in range(1, n):
    print("Copying cluster files to node 'slave" + str(node) + "'")
    transport = paramiko.Transport((nodes[node][3],22))
    transport.connect(username = "hduser", password = "hduser")
    sftp = paramiko.SFTPClient.from_transport(transport)
    sftp.put('./hosts.txt','/etc/hosts')
    sftp.put('./authorized_keys.txt','./.ssh/authorized_keys')
    sftp.put('./known_hosts.txt','./.ssh/known_hosts')
    sftp.put('./slaves.txt','/usr/local/hadoop/etc/hadoop/slaves')
    sftp.close()
    transport.close()
print("Cluster files copied")
```

Lanzamiento del clúster

Una vez todos los archivos de interconexión han sido creados y distribuidos, son copiados a los lugares adecuados en cada instancia.

Los archivos originales son denominados con la extensión *.txt* en el ordenador donde se ejecuta el programa. Esto es para que puedan ser consultados con comodidad y no se confundan con cualquier archivo similar, propio del sistema local.

Cuando son copiados a las instancias, lo son sin extensión, y en los directorios correspondientes.

```
# 4. Distribution of cluster files to each node
print("Copying cluster files to master")
transport = paramiko.Transport((nodes[0][3],22))
transport.connect(username = "hduser", password = "hduser")
sftp = paramiko.SFTPClient.from_transport(transport)
sftp.put('./hosts.txt','/etc/hosts')
sftp.put('./authorized_keys.txt','./.ssh/authorized_keys')
sftp.put('./slaves.txt','/usr/local/hadoop/etc/hadoop/slaves')
sftp.close()
transport.close()


for node in range(1, n):
```

```
    print("Copying cluster files to node 'slave" + str(node) + "'")
    transport = paramiko.Transport((nodes[node][3],22))
    transport.connect(username = "hduser", password = "hduser")
    sftp = paramiko.SFTPClient.from_transport(transport)
    sftp.put('./hosts.txt','/etc/hosts')
    sftp.put('./authorized_keys.txt','./.ssh/authorized_keys')
    sftp.put('./known_hosts.txt','./.ssh/known_hosts')
    sftp.put('./slaves.txt','/usr/local/hadoop/etc/hadoop/slaves')
    sftp.close()
    transport.close()
print("Cluster files copied")
```

Una vez copiados los archivos, ya se está en disposición de lanzar el clúster.

Antes, es necesario copiar a los nodos los *scripts* que van a ser utilizados en los cálculos. Son distribuidos junto al programa, en un archivo denominado *files.zip*. Este archivo es copiado al máster y descomprimido.

```
# 1. Copy scripts and matrix to master
print("Copying scripts and matrix to master")
transport = paramiko.Transport((nodes[0][3],22))
transport.connect(username = "hduser", password = "hduser")
sftp = paramiko.SFTPClient.from_transport(transport)
sftp.put('./files.zip','files.zip')
sftp.put('./' + Matrix,'matrix.csv')
sftp.close()
transport.close()
print("Scripts and matrix copied")


# 2. Cluster launching
print("Uncompressing 'files.zip'")
instance = paramiko.SSHClient()
instance.set_missing_host_key_policy(paramiko.AutoAddPolicy())
instance.connect(nodes[0][3],username="hduser",password="hduser")
instance.exec_command("unzip -o files.zip")
```

```
stdin, stdout, stderr = instance.exec_command(\
   "exec cp mapred-site.xml /usr/local/hadoop/etc/hadoop/mapred-site.xml")
for line in stderr:
   print(line)
for line in stdout:
   print(line)
```

Una vez descomprimido, se copia un archivo de configuración de Hadoop, *mapred-site.xml*.

E inmediatamente, se inicia el clúster Hadoop con YARN:

```
print("dfs starting")
print("------------")
print()
stdin, stdout, stderr = instance.exec_command(\
   "exec /usr/local/hadoop/sbin/start-dfs.sh")
for line in stderr:
   print(line)
for line in stdout:
   print(line)

print("yarn starting")
print("-------------")
print()
stdin, stdout, stderr = instance.exec_command(\
   "exec /usr/local/hadoop/sbin/start-yarn.sh")
for line in stdout:
   print(line)

print("history server starting")
print("----------------------")
print()
stdin, stdout, stderr = instance.exec_command(\
```

```
    "exec /usr/local/hadoop/sbin/mr-jobhistory-daemon.sh start historyserver")
for line in stdout:
    print(line)


print("Cluster launched. The master public IP is: " + nodes[0][3])
```

Se lanzan sucesivamente los scripts:

- *start-dfs.sh*
- *start-yarn.sh*
- *mr-jobhistory-daemon.sh start historyserver*

Y se muestra el tiempo que ha costado llegar a este punto, desde el inicio del programa.

```
print("The creation and launch of cluster has elapsed "\
    + str((datetime.datetime.now() - init_time).total_seconds()) \
    + " seconds")
```

CÁLCULO DE LA MATRIZ

Una vez el clúster en marcha, es el momento de realizar el cálculo de la matriz.

Este proceso consta de los siguientes pasos:

- Copia de scripts y matriz al máster y el resto de nodos.
- Preparación de la matriz auxiliar.
- Cálculo de la matriz.
- Depósito de la matriz en el ordenador local.

Copia de scripts y matriz al máster y el resto de nodos

Aunque, conceptualmente, esta fase corresponde a la de cálculo de la matriz, las copias ya han sido realizadas anteriormente, junto con otros archivos, para aprovechar las conexiones.

Preparación de la matriz auxiliar

Consta de tres partes:

- Creación de la matriz auxiliar.
- Copia de la matriz auxiliar al sistema de ficheros de hdfs.
- Copia de la matriz original a cada uno de los nodos.

La matriz auxiliar (*mapMatrix.csv*), se crea a partir de la matriz dada (*matrix.csv*), mediante el script Python *prepareMatrix.py*.

Está formada por pares i,j, correspondientes a todas las combinaciones de dígitos desde 0 hasta n.

Esta matriz será la que se divida entre los distintos contenedores del clúster, para realizar el cálculo paralelizado. La información sobre este proceso se detalla más adelante.

```
#1. Matrix preparation
print("matrix of pairs creation")
print("-----------------------")
print()
stdin, stdout, stderr = instance.exec_command(\
   "exec python prepareMatrix.py")
for line in stdout:
   print(line)
```

Una vez creada, esta matriz auxiliar debe ser copiada al sistema de ficheros del clúster, en Hdfs. Con esto, el programa Python paralelizado podrá encontrar un segmento en cada uno de los nodos, y realizará los cálculos recorriendo las líneas de su fragmento de matriz.

```
print("copy of matrix of pairs to dfs")
print("-----------------------------")
print()
stdin, stdout, stderr = instance.exec_command(\
   "exec /usr/local/hadoop/bin/hdfs dfs -put mapMatrix.csv /mapMatrix.csv")
for line in stderr:
   print(line)
for line in stdout:
   print(line)
```

Además, es necesario crear una copia de la matriz original en cada uno de los nodos del clúster. Esto es así porque cada nodo debe disponer de toda la matriz para encontrar los valores que corresponden a los pares i,j que le corresponden.

```
print("Copy of matrix.csv to every node")
print("-------------------------------")
print()
for node in range(1, n):
   stdin, stdout, stderr = instance.exec_command(\
   "exec scp matrix.csv hduser@slave" + str(node) + ":/home/hduser")
   print("matrix copied to node slave" + str(node))
for line in stdout:
   print(line)
```

Cálculo de la matriz

La matriz de correlación es calculada mediante una orden Spark que ejecuta un script de Python, *corrsp.py*.

La orden incluye todas las opciones de Spark utilizadas. Son éstas:

- **--master: Yarn**. Indica que Spark va a utiliza Yarn como gestor del clúster.
- **--deploy-mode: cluster**. Deja al software elegir el nodo desde el que se va a lanzar la distribución de cálculo.
- **--num-executors: *n***. Es un parámetro importante de rendimiento. Se indica que, si lo necesita, el software utilice todos los *n* nodos del clúster. Esto garantiza que no se dispone de nodos sin utilizar.
- **--executor-cores: 7**. Es el número de cores que utilizará el cluster por cada uno de los nodos. Las instancias m3 propuestas cuentan con 8 cores. Se sigue aquí el criterio de aprovechar al máximo la capacidad, dejando un core para el sistema operativo. En caso de utilizar instancias con más cores, sería recomendable aumentar este valor al número de instancias por nodo menos uno.
- **--executor-memory: 7**. Es el número de GB de RAM dedicados a cálculos de Spark por cada instancia. Las instancias m3 propuestas cuentan con 8 GB de RAM. Se sigue aquí el criterio de aprovechar al máximo la capacidad, dejando 1 GB para el sistema operativo. En caso de utilizar instancias con más RAM, sería recomendable aumentar este valor al tamaño de RAM por nodo, menos 1GB.
- **corrsp.py**. Es el *script* de Python a ejecutar. Se enviará una copia a cada uno de los nodos, que utilizará el fragmento de matriz auxiliar que le habrá correspondido.

```
#2. Calculation of correlation matrix

print()

print("calculation of correlation matrix")

print("--------------------------------")

print()

stdin, stdout, stderr = instance.exec_command(\

   "exec spark/bin/spark-submit --master yarn --deploy-mode cluster \

    --num-executors " + str(n) + " --executor-cores 7 \

    --executor-memory 7g corrsp.py")


count = 1

for line in stderr:

   if line.find("Application report for application"):

      print("calculating ... " + str(count))
```

```
        count += 1
    else:
        print(line)
```

El concepto de ejecutores y número de ejecutores es importante en este contexto. Spark divide las tareas en lo que denomina *contenedores*. Cada contenedor es una unidad de cálculo. El software contempla el conjunto de contenedores disponibles, uno por cada core.

Depósito de la matriz en el ordenador local

Una vez realizados los cálculos en cada uno de los nodos, éstos depositan, en el sistema de archivos del clúster (*hdfs*), el fragmento resultante para cada uno. Estos fragmentos se encuentran en un directorio denominado *output*. Su nombre está formado por el prefijo *part-* más un número. Este número corresponde al número del contenedor que ha realizado la tarea que da lugar al archivo.

Afortunadamente, la matriz auxiliar es dividida y distribuida entre los contendores de modo ordenado. Así, los resultados del archivo *part-0001* serán los primeros, continuándose sucesivamente por el *part-0002* y siguientes.

Puesto que el resultado que se desea es un archivo único, se utiliza una función de Spark, *getmerge*, que reúne todos los fragmentos en uno sólo. En este caso, un archivo denominado *output.txt*.

```
#3. Results retrieval


# getting parts
print()
print("assembling result parts from master")
print("--------------------------------")
print()
stdin, stdout, stderr = instance.exec_command(\
    "/usr/local/hadoop/bin/hdfs dfs -getmerge /output/part-* ./output.txt")
for line in stderr:
    print(line)
```

Una vez se dispone del archivo, es copiado al ordenador local mediante sftp de Paramiko.

```
transport = paramiko.Transport((nodes[0][3],22))
transport.connect(username = "hduser", password = "hduser")
sftp = paramiko.SFTPClient.from_transport(transport)
```

```
sftp.get('./output.txt','output.txt')
sftp.close()
transport.close()
```

Una vez en el ordenador local, se transforma el archivo *output.txt*, eliminando los símbolos auxiliares de Spark ("(", ")", "[", "]", "").

Al mismo tiempo, se sustituyen las comas por símbolos de punto y coma, y los puntos por comas.

Esto se hace porque Spark devuelve las cifras en formato americano (puntos para decimales), separados por comas. Los programas preparados para Europa utilizan las comas como separador decimal. Esto hace que no pueda ser utilizado el estándar de separación de campos por comas; es por eso que se sustituyen éstas por puntos y comas.

```
print()
print("creating correlation matrix")
print("--------------------------------")
print()

output = open("corrMatrix.csv", "w")
input = open("output.txt")
for line in input:
    out_line = line.replace("(","") \
                .replace(")","") \
                .replace("[","") \
                .replace("]","") \
                .replace("'","") \
                .replace(",",";") \
                .replace(".",",")
    output.write(out_line)

input.close()
output.close()

print("correlation matrix written to local disk as 'corrMatrix.csv' file")
```

```
print()
```

Tras esto, se dispone, en el ordenador local, de un archivo, denominado *corrMatrix.csv*, en el que se contiene la matriz de correlaciones entre las líneas de la matriz proporcionada. Como añadido, la primera columna corresponde al número de la línea, con el objeto de que se pueda comparar y verificar el resultado con lo esperado.

ELIMINACIÓN DE LAS INSTANCIAS

Una vez que se dispone del resultado esperado en el ordenador local, el último proceso es importante para cumplir con los requisitos de economía. Se trata de la eliminación de las instancias de AWS, para evitar que se incremente el gasto de manera inútil.

Puesto que se dispone de la lista *nodes*, el proceso la recorre, utilizando la identificación de la instancia para ordenar su eliminación.

```
#1. Instances termination


print()
print("terminating instances")
print("---------------------")
print()


for i in range(0, n):
    print("Instancia: ",nodes[i][0])
    ec2.instances.filter(InstanceIds = [nodes[i][0]]).terminate()
    print("Instance " + str(i) + " terminated")


total_time = datetime.datetime.now() - init_time


print("Finished. " + str(n) + " instances of type " + InstanceType \
      + " have been used")
print("Total time elapsed has been " + str(total_time.seconds) \
      + " seconds, or " + str(int(total_time.seconds / 60)) + " minutes.")
```

Con esto, el programa de creación de cluster para cálculo de matrices queda finalizado. Habiendo entregado el resultado esperado.

IMAGEN DE AWS (AMI) A PARTIR DE LA CUAL SE CREAN LOS CLÚSTERES

Como se ha comentado, las instancias no son creadas *ex novo*, sino que lo son a partir de una imagen AMI creada previamente.

Una imagen de este tipo existe en AWS, y es ofrecida por el autor a quien desee utilizarla. Este extremo se indica en el archivo *Readme.txt*, que se distribuye junto con el software, tanto directamente como vía GitHub. Sólo basta enviar un mensaje al correo indicado en el archivo, solicitando utilizar la imagen, e indicando el identificador AWS del remitente. El autor proporcionará acceso al AMI al identificador indicado.

En caso de que el usuario desee crear su propia AMI, a continuación se indica la secuencia de acciones realizada. Para este proceso, se presupone una mínima familiaridad con el entorno de trabajo de AWS, especialmente con el servicio S3.

CREACIÓN DE LA INSTANCIA ORIGINAL

El primer paso es crear una instancia AWS. En este caso, la AMI utilizada como base ha sido la de Ubuntu Server 14.04 LTS, del tipo *m3.medium*.

Es posible que, en el momento de la creación, existan versiones de Ubuntu más actualizadas. La recomendación es utilizar la más reciente.

En cuanto al tamaño, se escoge el tipo *m3.medium* por ser el más económico, pues para la instalación no es necesaria potencia de cálculo. Existe un tipo inferior, *m3.micro*, pero sus reducidas prestaciones complican la configuración, sin aportar un significativo ahorro económico.

También es posible que, en el transcurso del tiempo, AWS sustituya las instancias tipo *m3* por otras equivalentes, (como las *m4*), de prestaciones mejoradas.

Hay que recordar que la decisión sobre el tipo de instancia es sólo para el proceso de creación de la imagen. Una vez disponible ésta, no queda asociada a ningún tipo, pudiendo variar éste en la creación de las instancias derivadas.

Durante la creación de la instancia, AWS solicita un conjunto de reglas de seguridad. Si no se es experto en la identificación de las reglas necesarias (acceso SSH, principalmente), puede utilizarse la opción de ALLOW ALL TRAFFIC FROM ANYWHERE. Esto permitirá todo tipo de acceso a la instancia. Pero, dado el poco tiempo que va a encontrarse activa, no es esperable que pueda ver su seguridad comprometida.

CONFIGURACIÓN DE USUARIO, GRUPO Y AUTENTICACIÓN

Una vez creada la instancia, el acceso se realiza utilizando la dirección IP pública (aparece en el panel AWS), mediante alguna herramienta de acceso SSH (en entornos Windows, la herramienta PuTTY es la más ampliamente utilizada).

Para ello, es necesario disponer en el ordenador local de las claves de acceso AWS del usuario, y realizar el acceso SSH basado en ellas.

Una vez accedido (usuario por defecto, *ubuntu*), el primer paso es crear el usuario (*hduser*) y grupo (*hadoop*) que va a ser utilizado durante las operaciones del cluster:

```
sudo addgroup hadoop

sudo adduser --ingroup hadoop hduser

(en este momento, pedirá una clave para el usuario 'hduser'. Ésta debe ser hduser).
```

Es importante el estricto ajuste a estas instrucciones, incluida la clave de usuario, pues los programas asumen estas informaciones.

```
sudo adduser hduser sudo

sudo nano /etc/ssh/sshd_config
```

La última orden mostrará el contenido del archivo *sshd_config*, para ser modificado.

En él hay que:

1) Localizar la línea que contiene la palabra '*PasswordAuthentication*', y sustituir '*no*' por '*yes*'.

2) Localizar la línea que contiene la palabra '*StrictHostKeyChecking*', y sustituir '*no*' por '*yes*'

3) Localizar la línea que contiene la palabra *'MaxSessions'*, e indicar en ella el valor *20* (*MaxSessions=20*).

(Utilizar la combinación de claves ctrl+O para guardar las modificaciones, y ctrl+X para salir del editor de textos).

```
sudo service ssh restart

sudo su - hduser
```

ACTUALIZACIÓN DE PAQUETES, CONFIGURACIÓN DE JDK Y SSH

```
sudo apt-get update

sudo apt-get install openjdk-7-jdk

cd /usr/lib/jvm

sudo ln -s java-7-openjdk-amd64/ jdk

ssh-keygen -t rsa -P ""

cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

ssh localhost

exit
```

DESCARGA DE HADOOP

Descargar la última versión de Hadoop:

```
Wget
http://mirror.cc.columbia.edu/pub/software/apache/hadoop/common/hadoop-2.7.3/hadoop-2.7.3.tar.gz
```

En caso de que la versión más reciente haya cambiado, se recomienda utilizarla.

```
sudo tar vxzf hadoop-2.7.3.tar.gz -C /usr/local

cd /usr/local

sudo mv hadoop-2.7.3 hadoop

sudo chown -R hduser:hadoop Hadoop
```

CONFIGURACIÓN DE VARIABLES DE ENTORNO PARA HADOOP

```
sudo nano .bashrc
```

Pegar las siguientes líneas al final del archivo:

```
#Hadoop Variables

export JAVA_HOME=/usr/lib/jvm/jdk/

export HADOOP_INSTALL=/usr/local/hadoop

export PATH=$PATH:$HADOOP_INSTALL/bin

export PATH=$PATH:$HADOOP_INSTALL/sbin

export HADOOP_MAPRED_HOME=$HADOOP_INSTALL

export HADOOP_COMMON_HOME=$HADOOP_INSTALL

export HADOOP_HDFS_HOME=$HADOOP_INSTALL

export YARN_HOME=$HADOOP_INSTALL

#End
```

Salir de la edición guardando las modificaciones (ctrl+O, ctrl+X)

```
source ~/.bashrc
```

CONFIGURACIÓN DE ARCHIVOS HADOOP

```
cd $HADOOP_INSTALL/etc/hadoop

sudo nano hadoop-env.sh
```

En este archivo, modificar la línea:

export JAVA_HOME=${JAVA_HOME}

y sustituirla por:

export JAVA_HOME=/usr/lib/jvm/jdk/

Guardar y cerrar.

Verificar la instalación de Hadoop mediante la orden

hadoop versión

Debería aparecer un mensaje como el siguiente:

```
Hadoop 2.7.3
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r baa91f7c6bc9cb92be5982de4719c1c8af91ccff

Compiled by root on 2016-08-18T01:41Z

Compiled with protoc 2.5.0

From source with checksum 2e4ce5f957ea4db193bce3734ff29ff4

This command was run using /usr/local/hadoop/share/hadoop/common/hadoop-common-2.7.3.jar
```

Si es correcto, pasar a editar los archivos de configuración:

```
sudo nano core-site.xml
```

En este archivo, situar, entre las líneas <configuration> y </configuration>, las siguientes:

```
<property>

 <name>fs.default.name</name>

 <value>hdfs://master:9000</value>

</property>
```

Guardar y salir.

```
sudo nano yarn-site.xml
```

En este archivo, situar, entre las líneas <configuration> y </configuration>, las siguientes:

```
<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.scheduler.address</name>
 <value>master:8030</value>
</property>
<property>
 <name>yarn.resourcemanager.address</name>
 <value>master:8032</value>
</property>
<property>
 <name>yarn.resourcemanager.webapp.address</name>
 <value>master:8088</value>
</property>
<property>
 <name>yarn.resourcemanager.resource-tracker.address</name>
 <value>master:8031</value>
</property>
<property>
 <name>yarn.resourcemanager.admin.address</name>
 <value>master:8033</value>
</property>
```

Guardar y salir.

```
sudo mv mapred-site.xml.template mapred-site.xml
sudo nano mapred-site.xml
```

En este archivo, situar, entre las líneas <configuration> y </configuration>, las siguientes:

```
<property>
 <name>mapreduce.framework.name</name>
 <value>yarn</value>
</property>
```

Guardar y cerrar.

```
sudo nano hdfs-site.xml
```

En este archivo, situar, entre las líneas <configuration> y </configuration>, las siguientes:

```
<property>
 <name>dfs.replication</name>
 <value>3</value>
 <description>default value is 3. It is the number of replicated files across HDFS</description>
</property>
<property>
 <name>dfs.namenode.name.dir</name>
 <value>file:/home/hduser/mydata/hdfs/namenode</value>
</property>
<property>
 <name>dfs.datanode.data.dir</name>
 <value>file:/home/hduser/mydata/hdfs/datanode</value>
</property>
<property>
 <name>dfs.http.address</name>
 <value>master:50070</value>
 <description>Enter your Primary NameNode hostname for http access.</description>
</property>
<property>
 <name>dfs.secondary.http.address</name>
 <value>slave:50090</value>
 <description>Optional. Only if you want you sec. name node to be on specific node</description>
</property>
```

Guardar y cerrar

```
sudo nano slaves
```

Eliminar la línea:

localhost

Sustituirla por la línea:

master

Guardar y cerrar

```
cd
sudo mkdir -p mydata/hdfs/namenode mydata/hdfs/datanode
sudo chown -R hduser:hadoop mydata
hdfs namenode -format
sudo chmod 757 /etc
sudo chmod 646 /etc/hosts
```

INSTALACIÓN DE PAQUETES PYTHON

Especialmente, *numpy*, utilizado para cálculos.

```
sudo apt-get install python-pip python-dev build-essential
sudo pip install --upgrade pip
sudo pip install mrjob
sudo apt-get install python-numpy
```

CREACIÓN DE LA IMAGEN

Una vez completada la instalación y configuración, detener la instancia y crear, a partir de ella, una AMI privada.

Esta AMI será la utilizada para crear los futuros clústers.

En caso de ser necesaria cualquier modificación en la AMI, será necesario crear una instancia a partir de ella, modificar la instancia y crear una nueva AMI, destruyendo posteriormente la anterior, ya inútil.

PROGRAMA DE GENERACIÓN DE LA MATRIZ DE APOYO

Invocado por el programa principal, el script Python *prepareMatrix.py* genera un archivo sintético, *mapMatrix.csv*. Aunque se le denomine como "matriz de apoyo", en realidad es un listado de pares de números. Cada par representa una de las posibles combinaciones de *n* elementos.

El objeto es que Spark fragmente este listado en tantas partes como "contenedores", o unidades de proceso, vayan a ser utilizadas. Cada unidad de proceso recorrerá linealmente la parte del archivo que le haya correspondido, y calculará la correlación entre el par de líneas de la matrix dada, indicado por el par de valores de la línea.

```
import time
```

```
mapMatrix = open("mapMatrix.csv","w")
with open("matrix.csv") as matrix:
    mx_rows = matrix.readlines()
st_time = time.time()


for i in range(0,len(mx_rows)):
    for j in range(0,len(mx_rows)):
        pair = str(i).zfill(3) + "," + str(j).zfill(3)
        mapMatrix.write(pair + "\n")
        if ((i % 50 == 0) and (j == 0)):
            print(i)


print("To prepare the matrix has costed: ",time.time() - st_time)
matrix.close()
mapMatrix.close()
```

PROGRAMA DE CÁLCULO DE LA MATRIZ

Invocado por el programa principal, el script Python *corrsp.py* recorre, en cada *contenedor* de Spark, el fragmento del archivo *mapMatrix.csv* correspondiente. Obtiene las líneas correspondientes del archivo *matrix.csv*, calcula la correlación, y devuelve un archivo con las correlaciones por cada línea.

Se trata de un programa Python que será ejecutado por Pyspark en el entorno Spark.

Utiliza las bibliotecas *linecache*, para obtener una línea de un archivo a partir de un número, y *Numpy*, para el procedimiento de correlación utilizado, *corrcoef* de Numpy.

El primer paso es la carga del archivo *mapMatrix.csv* en el entorno Spark.

```
mapMatrix = sc.textFile("/mapMatrix.csv",80)
```

El parámetro que sigue al nombre del archivo es importante. Se trata del número de particiones en que el sistema va a dividir el *DataFrame*, la entidad Spark que representa a la matriz.

El programa invocado se ejecutará en tantos *cores* como se indique en este parámetro. Tras varias pruebas, se ha estimado en 80 para un conjunto de 20 instancias con 8 cores cada una.

El rendimiento que ofrecen Spark depende mucho de este número de particiones. Si es bajo, se estará infrautilizando el clúster. Si es alto, el coste de fragmentar el

DataFrame y el tráfico de red entre los contenedores, hará descender el rendimiento. Empíricamente se comprueba que, en condiciones normales, el punto óptimo se encuentra entre un 60 y un 80% del conjunto total de *cores* disponibles en el clúster.

Puesto que el objeto de este trabajo es la simplicidad, se ha proporcionado esta cifra, que se estima eficaz para conjuntos de entre 10 y 20 nodos.

A continuación, se preparan dos funciones, *ind* y *line*:

```
def ind(i,pos):
    return i.encode('utf-8').split(",")[pos]
```

La función *ind* toma la línea del archivo *mapMatrix.csv*, compuesta por dos dígitos separados por comas, y devuelve el valor del primero, si *pos* es *0*, y el del segundo, si *pos* es *1*..

Puesto que Spark trabaja con codificación utf-8, es necesario realizar las conversiones oportunas.

```
def line(ind):
    return linecache.getline("/home/hduser/matrix.csv", int(ind)+1).strip("\n").split(",")
```

La función *line* toma un número y, con él, busca la línea correspondiente en el archivo *matrix.csv*. La línea es un conjunto de unos y ceros separados por comas, que son transformados en una lista Python con tantos elementos como números.

A continuación, se ejecuta la secuencia de transformaciones Spark:

```
mapMatrix.map(lambda x: (ind(x,0),np.corrcoef(line(ind(x,0)),line(ind(x,1)))[1][0])).groupByKey().mapValues(list).sortByKey().saveAsTextFile("/output")
```

Las transformaciones, desglosadas, son las siguientes:

```
mapMatrix.map(lambda x: (ind(x,0),np.corrcoef(line(ind(x,0)),line(ind(x,1)))[1][0]))
```

Esta transformación se aplica sobre el *DataFrame mapMatrix*. Para cada línea (*x*), se devuelve un par de valores: el número de línea *i* de la matriz auxiliar, y el coeficiente de correlación entre las líneas *i* y *j* de la matriz dada.

El número de línea es proporcionado por la función *ind* con índice *0*. Es decir, el primer valor del par *i*, *j*.

El coeficiente de correlación se obtiene de aplicar la función *corrcoef* de *Numpy* a dos vectores: *line(ind(x,0))* y *line(ind(x,1))*. Son las dos listas Python que proporciona la función *line* para los valores *i* (*ind(x,0)*) y *j* (*ind(x,1)*) del archivo *mapMatrix.csv*.

La función *corrcoef* de Numpy no proporciona un valor, sino una matriz de 2x2. En ella, el coeficiente buscado se encuentra, igual, en los valores fuera del eje. Se elige uno de ellos mediante las coordenadas [0][1].

La transformación anterior proporciona un *DataFrame* compuesto por pares: el primer valor es el número de la línea en la matriz dada, y el segundo, el coeficiente buscado.

Para obtener una matriz de esta lista lineal, se aplica la siguiente transformación:

```
groupByKey()
```

Esta transformación devuelve un *DataFrame* en el que cada línea está formada por una clave y un iterable. La clave es el número de línea, y la lista, el conjunto de coeficientes de correlación que corresponden a esa línea.

Este *DataFrame* ya tendrá sólo tantas líneas como la matriz dada.

La siguiente transformación es del tipo *mapValues()*, por lo que sólo se realiza sobre la lista de cada línea, no sobre la clave. Transforma el iterable en una lista Python.

```
mapValues(list)
```

Una vez creadas las listas Python, se ordenan por clave. La clave es siempre el número de orden de la línea en la matriz dada:

```
sortByKey()
```

Es importante para garantizar que el orden de salida es el adecuado. Una vez ordenadas, las líneas son guardadas como un archivo en el sistema de ficheros *hdfs* del clúster.

```
saveAsTextFile("/output")
```

Como ya se ha comentado en otro punto, cada contenedor crea un archivo, que es necesario reunir a los otros mediante el procedimiento indicado al comentar el programa *calcCorrMatrix.py*.

PROGRAMA DE ELIMINACIÓN DE INSTANCIAS

Si el proceso funciona correctamente, al finalizar, las instancias creadas serán eliminadas automáticamente.

Si, por cualquier causa, el proceso queda interrumpido, las instancias creadas hasta el momento permanecerán activas. Con el consiguiente coste económico, sin utilidad.

En caso de interrupción, o de duda por parte del usuario, se puede invocar en cualquier momento el programa *instancesTerminate.py*.

Este programa crea una lista con todas las instancias asociadas a la cuenta del usuario que incluyan el par de etiquetas *Key: Project* y *Value: BouncyHadoop*.

A continuación, recorre la lista, eliminando las instancias mediante su ID.

Si el usuario no ha modificado las etiquetas, se garantiza que nunca van a existir instancias activas asociadas a esta herramienta, más allá del tiempo necesario para realizar los cálculos.