Cloud Computing: Servicios y Aplicaciones
Máster en Ingeniería Informática.
Dept. CCIA - Universidad de Granada.
Profesores: José M. Benítez Sánchez y Manuel J. Parra-Royón
Curso: 2020-2021

Spark-MLlib con Python

Acceso al servidor

El acceso al servidor se realiza con ssh usando:

```
ssh ccsa<DNI>@hadoop.ugr.es
```

Necesitas tu clave de acceso para conectar con el cluster.

Si trabajas con Windows, puedes instalar `Putty` o bien usar los recursos Linux de Windows 10 para utilizar directamente ssh.

Verificación del acceso a HDFS

Comprueba que tienes acceso a tu directorio home de **HDFS**, con

```
hdfs dfs -ls ./
```

o bien

```
hdfs dfs -ls /user/tuusuario/
```

Debes ver los ficheros que creamos en la sesión anterior. En el caso de no aparecer nada, puede ser que no creases los ficheros de prueba. Para verificar que tienes tu home en HDFS, escribe `hdfs dfs -ls /user/` y comprueba que tus cuenta está ahí.

Lectura y Escritura en HDFS.

Todo el trabajo de entrada y salida de datos es importante, ya que tiene que estar en tu HDFS, no en tu HOME local, es decir en tu directorio HOME de usuario. Por tanto si queremos leer el fichero `./a.txt`, se intentará leer el fichero desde `hdfs://user/<tuusuario>/a.txt` y no desde `/home/a.txt`.
Esto suele ser motivo de muchos errores al inicio del trabajo con Spark.

Ejecución de Scripts/Aplicaciones en Python/R/Scala en el cluster

Spark context

Punto de entrada principal para la funcionalidad de Spark. Un SparkContext representa la conexión a un cluster de Spark, y puede ser usado para crear RDDs, acumuladores y variables en ese cluster.

Para nuestro cluster usaremos el siguiente:

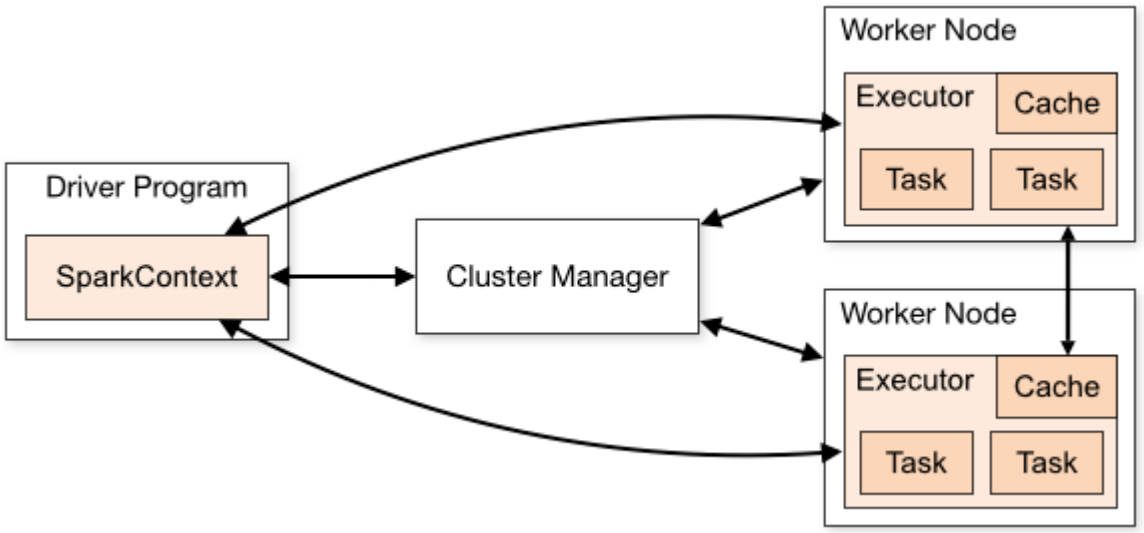
Spark Context en Python:

```
conf = SparkConf().setAppName("Word Count - Python").set("spark.hadoop.yarn.resourcemanager.address", "hadoop-master:8032")
```

Ejecución en modo local, standalone o Yarn

Para que Spark funcione necesita recursos. En el modo autónomo se inician los workers y el maestro de Spark y la capa de almacenamiento puede ser cualquiera --- HDFS, Sistema de Archivos, Cassandra etc. En el modo YARN lo que ocurre es que se está pidiendo al clúster YARN-Hadoop que administre la asignación de recursos y la gestión del mismo de forma mucho más óptima.

Cuando se usa master como local[2] pides a Spark que use 2 cores y ejecute el maestro y los trabajadores en la misma JVM. En el modo local todas las tareas relacionadas con el trabajo de Spark se ejecutan en la misma JVM. Por tanto, la única diferencia entre el modo Standalone y el local es que en el modo Standalone está definiendo "contenedores" para que el worker y el maestro de Spark corran en su máquina (así puede tener 2 workers y sus tareas pueden ser distribuidas en la JVM de esos dos workers...) pero en el modo local está corriendo todo en la misma JVM en la máquina local.



Cómo enviar un trabajo al cluster

La sintaxis es la siguiente:

```
spark-submit
  --class <main-class>
  --master <master-url>
  --deploy-mode <deploy-mode>
  --conf <key>=<value>
 ….
  <application-jar>
  [application-arguments]
```

Para enviar un ejemplo al cluster necesitamos hacer lo siguiente:

1.- Descargamos un conjunto de datos en nuestra cuenta:

```
wget https://raw.githubusercontent.com/mattf/joyce/master/james-joyce-ulysses.txt
```

2.- Mover el fichero a hdfs:

```
hdfs dfs -put james-joyce-ulysses.txt ./
```

3.- Descargar el código fuente del programa:

```
wget
https://gist.githubusercontent.com/manuparra/f6e2924e26b50a9d8028f
f25a3f32495/raw/351518089e099f77ac1ccbb0bf8a297e99ac0c58/wordcount
.py

cat wordcount.py
```

4.- Editar el código fuente del programa y modificar los datos de salida al directorio HDFS destino: Esto es necesario hacerlo cada vez que obtengamos resultados en HDFS ya que Spark por defecto no sobrescribe resultados y el proceso termina en error cuando lo intenta.

5.- Enviar el programa a ejecución.

```
/opt/spark-2.2.0/bin/spark-submit --master
spark://hadoop-master:7077 --total-executor-cores 5
--executor-memory 1g wordcount.py
```

Ejemplo detallado: Contar palabras (Word Count)

Veamos el ejemplo del programa WordCount en https://gist.githubusercontent.com/manuparra/f6e2924e26b50a9d8028ff25a3f32495/raw/351518089e099f77ac1ccbb0bf8a297e99ac0c58/wordcount.py

En este ejemplo creamos un SparkContext para conectar el Driver que funciona localmente:

```
sc = SparkContext("local","PySpark Word Count")
```

A continuación, leímos el archivo de texto de entrada usando la variable SparkContext y creamos un mapa plano de palabras. words es del tipo PythonRDD.

```
words = sc.textFile("./james-joyce-ulysses.txt").flatMap(lambda
line: line.split(" "))
```

hemos dividido las palabras usando un solo espacio como separador.

Luego mapearemos cada palabra a un par clave:valor de palabra:1, siendo 1 el número de ocurrencias.

```
words.map(lambda word: (word, 1))
```

The result is then reduced by key, which is the word, and the values are added.

```
reduceByKey(lambda a,b:a +b)
```

The result is saved to a text file.

```
wordCounts.saveAsTextFile("...")
```

¿Dónde están los resultados?

Los resultados de la ejecución está dentro de la carpeta HDFS destino que se ha indicado en `saveAsTextFile()`. Al ver los ficheros generados usando: `hdfs dfs -ls directorio_destino`, aparece lo siguiente:

```
part-00000
part-00001
part-00002
…
```

Cada uno de los ficheros contiene los pares claves (palabra) - valor (suma del número de ocurrencias).

Si queremos unirlos todo el conjunto de resultados hay que usar la función merge de HDFS, que fusiona los resultados de las partes de la salida de datos:

```
hdfs dfs -getmerge ./james-joyce-ulysses.txt ./results_wc_joyce
```

Para que el propio programa devuelva los datos ya procesados, debemos tener en cuenta que estamos tabajando con datos que están distribuidos y han de ser “recogidos” de los nodos donde se encuentran.

En el siguiente ejemplo podemos ver como hacer la misma función que el comando anterior, pero usando `.collect()` para recolectar los datos y mostrarlos en pantall o guardarlos en un fichero de nuevo. Hay que tener cuidado con collect(), pues convierte un RDD a un objeto python equivalente, por lo que está limitado a los recursos del nodo:

https://gist.githubusercontent.com/manuparra/f6e2924e26b50a9d8028ff25a3f32495/raw/351518089e099f77ac1ccbb0bf8a297e99ac0c58/wordcount_collect.py

Consola interactiva PySpark

Para conectar a la consola interactiva, es decir un shell de Python pero con el contexto ya disponible para trabajar en Spark usando la variable spark que lo controla:

Hay que ejecutar:

```
/opt/spark-2.2.0/bin/pyspark --master spark://hadoop-master:7077
```

Dentro del entorno interactivo puedes realizar pruebas con Python antes de hacer un Submit a Spark.

Trabajo con RDDs / SparkDataFrames

Resilient Distributed Datasets -RDD- (concepto que sustenta los fundamentos de Apache Spark), pueden residir tanto en disco como en memoria principal.

Cuando cargamos un conjunto de datos (por ejemplo proveniente de una fuente externa como los archivos de un directorio de un HDFS, o de una estructura de datos que haya sido previamente generada en nuestro programa) para ser procesado en Spark, la estructura que usaremos internamente para volcar dichos datos es un RDD. Al volcarlo a un RDD, en función de cómo tengamos configurado nuestro entorno de trabajo en Spark, la lista de entradas que componen nuestro dataset será dividida en un número concreto de particiones (se "paralelizará" (`paralelize`)), cada una de ellas almacenada en uno de los nodos de nuestro clúster para procesamiento de Big Data. Esto explica el apelativo de "distributed" de los RDD.

A partir de aquí, entrará en juego una de las características básicas de estos RDD, y es que son inmutables. Una vez que hemos creado un RDD, éste no cambiará, sino que cada transformación (map, filter, flatMap, etc.) que apliquemos sobre él generará un nuevo RDD. Esto por ejemplo facilita mucho las tareas de procesamiento concurrente, ya que si varios procesos están trabajando sobre un mismo RDD el saber que no va a cambiar simplifica la gestión de dicha concurrencia. Cada definición de un nuevo RDD no está generando realmente copias de los datos. Cuando vamos aplicando sucesivas transformaciones de los datos, lo que estamos componiendo es una "receta" que se aplica a los datos para conseguir las transformaciones.

Ejemplo de RDD simple:

```
#Lista general en Python:
lista = ['uno','dos','dos','tres','cuatro']
#Creamos el RDD de la lista:
listardd = sc.parallelize(lista)
#Recolectamos los datos del RDD para convertirlo de nuevo en Obj
Py
print(listardd.collect())
```

Principales diferencias entre un RDD y Dataframes

- Tanto RDD como los dataframes provienen de datasets (listas, datos csv,...)
- Los RDD necesitan ser subidos a HDFS previamente, los dataframes pueden ser cargados en memoria directamente desde un archivo como por ejemplo un .csv
- Los RDD son un tipo de estructura de datos especial de Apache Spark, mientras que los dataframes son una clase especial de R.
- Los RDD son inmutables (su edición va generando el DAG), mientras que los dataframes admiten operaciones sobre los propios datos, es decir, podemos cambiarlos en memoria.
- Los RDD se encuentran distribuidos entre los cluster, los dataframes en un único cluster o máquina.
- Los RDD son tolerantes a fallos, los dataframes

Carga de datos desde CSV

Descarga este dataset de COVID-19:

`wget https://opendata.ecdc.europa.eu/covid19/casedistribution/csv/`

cámbiale el nombre a `covid19.csv` y muévelo a HDFS.

Para cargar como un CSV dentro de SPARK como un DataFrame

```
df = sc.read.csv("/home/stp/test1.csv",header=True,sep=",",inferSchema=True);
```

o

```
df = sc.read.csv("/home/stp/test1.csv",header=True,sep=",",inferSchema=False);
```

Una vez hecho esto df contiene un DataFrame para ser utilizado con toda la funcionalidad de Spark.

Manipulación de los datos con SparkSQL

Una vez creado el DataFrame es posible transformarlo en otro componente que permite el acceso al mismo mediante una sintaxis basada en SQL (ver https://spark.apache.org/docs/2.2.0/sql-programming-guide.html)

Usando la variable anterior:

```
df.createOrReplaceTempView("test")

sqlDF = spark.sql("SELECT * FROM test")
sqlDF.show()
```

Ejemplo plantilla

```
import sys

from pyspark import SparkContext, SparkConf
from pyspark.ml.classification import LogisticRegression

if __name__ == "__main__":

  # create Spark context with Spark configuration
  conf = SparkConf().setAppName("Practica 4")

  sc = SparkContext(conf=conf)

  df = sc.read.csv("/user/ccsaDNI/fichero.csv",header=True,sep=",",inferSchema=True)

  df.show()

  df.createOrReplaceTempView("sql_dataset")

  sqlDF = sc.sql("SELECT campo1, camp3, ... c6 FROM sql_dataset LIMIT 12")
  sqlDF.show()



  lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
  lrModel = lr.fit(sqlDF)
  lrModel.summary()
  #df.collect() <- NO!
```

```
sc.stop()
```