

Spark Streaming

Kafka. SOLUCION

Vamos a crear una estructura de Kafka STREAMING

Para ello vamos a usar la estructura de código PRODUCTOR y código CONSUMIDOR

Trabajo que debe desarrollar el PRODUCTOR. → Deberá mandar el contenido de un JSON que se os facilitará para que el consumidor que lo escuche pueda tratarlo.

Para la creación de dicho productor se os pedirá que lo hagáis de dos maneras

Usando la sintaxis vista en clase con el método **Kafka-console-producer.sh**

Trabajo que debe desarrollar el CONSUMIDOR → Nuestro consumidor deberá también tener dos entornos distintos de trabajo, para poder chequear que Kafka funciona correctamente.

1. Usando la sintaxis de **Kafka-console-consumer.sh**. En éste caso solo queremos comprobar que nuestro consumidor puede leerlo sin problema

Solución de la parte de terminal

Debemos iniciar sesión con el usuario KAFKA de tal manera que tengamos un terminal con el productor y en el otro el consumidor y lanzar en cada uno los comandos siguientes:

Productor

Cat bin/fichero.json | bin/kafka-console-producer.sh --broker-list localhost:9092 --topic **topicjson** > /dev/null

Consumidor

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic **topicjson** --from-beginning

Ojo la entrada del fichero lo podéis hacer como os lo muestro o con el Re direccionamiento de entrada (<) los dos formatos son válidos

.....

Solución de scala y productor

2. Creando un código en Scala donde vamos a ampliar nuestro trabajo. Deberá leerlo igual que en el caso anterior, pero en éste caso también deberá hacer un tratamiento especial de los datos que vayamos leyendo.
Para cada apartado que se os va a proponer debemos crear funciones de SCALA.
 - a. Queremos que filtre (que no aparezcan) del fichero JSON dos palabras que elegiréis cada uno (así evitamos tentaciones de copia jejej). Debo ver el JSON con esas palabras filtradas

PASOS A SEGUIR

1. Debemos modificar el JSON para que nos quede en formato legible, es decir eliminar los [] y poner cada objeto en una única línea
2. Debemos lanzar en el terminal el productor del apartado anterior para que nuestro consumidor SCALA lo trate

Productor

Cat bin/fichero.json | bin/kafka-console-producer.sh --broker-list localhost:9092 --topic **topicjson** > /dev/null

3. Crear el código en SCALA como hicimos en clase un ejemplo puede ser:

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.{IntegerType,
StringType, StructType}
import org.apache.spark.sql.functions.{from_json,col}
object solucionPractica {
def main(args: Array[String]):Unit={
// creamos una conexión con n nodos ( en clase
//usábamos 2 aquí también sirve)
val spark =
SparkSession.builder().appName("appEjemploConsum
").master("local[*]").getOrCreate()
```

// Leemos datos externos con formato kafka

```
val df = spark.readStream
.format("kafka")
.option("kafka.bootstrap.servers", "localhost:9092")
.option("subscribe", "topicpractica")
.option("startingOffsets", "earliest")
.load()
```

// Convertir el formato Kafka a un formato legible
//String

```
val res = df.selectExpr("CAST(value AS STRING)")
```

// creamos el schema de nuestro fichero

```
val schema = new StructType()
.add("id", IntegerType)
.add("first_name", StringType)
.add("last_name", StringType)
.add("email", StringType)
.add("gender", StringType)
.add("ip_address", StringType)
```

// leemos los datos para que se filtren dos campos,
he seleccionado los dos primero

```
val persona = res.select(from_json(col("value"),
schema).as("data")).select("data.*")
.filter("data.first_name != 'Giavani'")
.filter("data.last_name != 'Penddreh'")
```

```
// Finalmente escribimos el resultado por consola.
//Si querías podías redireccionarlo a un fichero
persona.writeStream
.format("console")
.outputMode("append")
.start()
.awaitTermination()
}
}
```

```
20/02/02 11:28:32 INFO DataWritingSparkTask: Committed partition 0 (task 0, attempt 0, stage 0.0)
20/02/02 11:28:32 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1814 bytes result sent to driver
20/02/02 11:28:32 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 745 ms on localhost (executor driver) (1/1)
20/02/02 11:28:32 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
20/02/02 11:28:32 INFO DAGScheduler: ResultStage 0 (start at solucionPractica.scala:37) finished in 1.285 s
20/02/02 11:28:32 INFO DAGScheduler: Job 0 finished: start at solucionPractica.scala:37, took 1.459203 s
20/02/02 11:28:32 INFO WriteToDataSourceV2Exec: Data source writer org.apache.spark.sql.execution.streaming.sources.MicroBatch
-----
Batch: 0
-----
20/02/02 11:28:32 INFO CodeGenerator: Code generated in 10.726724 ms
20/02/02 11:28:32 INFO CodeGenerator: Code generated in 39.839786 ms
-----+-----+-----+-----+-----+-----+
| id|first_name|last_name| email|gender| ip_address|
-----+-----+-----+-----+-----+-----+
| 3| Noell| Bea|nbea2@imageshack.us|Female|180.66.162.255|
| 4| Willard| Valek|vwalek3@vk.com|Male| 67.76.188.26|
-----+-----+-----+-----+-----+-----+
20/02/02 11:28:32 INFO WriteToDataSourceV2Exec: Data source writer org.apache.spark.sql.execution.streaming.sources.MicroBatch
20/02/02 11:28:32 INFO SparkContext: Starting job: start at solucionPractica.scala:37
20/02/02 11:28:32 INFO DAGScheduler: Job 1 finished: start at solucionPractica.scala:37, took 0.000044 s
20/02/02 11:28:32 INFO CheckpointFileManager: Writing atomically to file:/tmp/temporary-c61995fd-6d58-4072-8a35-0951cc621da5,
20/02/02 11:28:32 INFO CheckpointFileManager: Renamed temp file file:/tmp/temporary-c61995fd-6d58-4072-8a35-0951cc621da5/comp
20/02/02 11:28:32 INFO MicroBatchExecution: Streaming query made progress: {
  "id" : "5f8bfe24-01a8-478e-9ac4-bc3febe9f14a",
```

Parte de Investigación.

Si alguien quiere obtener una mejor nota, tenéis instalado en vuestra máquina Zeppelin.
La idea es que me expliquéis los pasos que deberíamos seguir para conseguir lo mismo que yo.

Necesito que me expliquéis los pasos más relevantes de la instalación, y la forma de configurar nuestro SPARK para que podamos trabajar en el note de zeppelin.

Una vez lo tengamos todo montado:

Debemos hacer un pequeño ejercicio para repasar la parte inicial de nuestro curso.

Deberemos trabajar con el csv “**amigos.csv**” y de él mostrar datos:

- Calcular el número de registros que tenemos

Solución ZEPELLIN

Parte 1. Explicación de los pasos más relevantes en la instalación de ZEPELLIN:

a. Debemos acceder a la ruta donde tenemos descargado Zeppelin



```
keepcoding@debian: ~  
Archivo Editar Ver Buscar Terminal Ayuda  
root@debian: /home/keepcoding/spark-2.4.4-bin-hadoop2.7#
```

b. Debemos acceder al directorio **conf** y editar uno de sus ficheros, concretamente el archivo **zeppelin-env.sh**.

Dentro de ese archivo debemos comentar la línea de
Export SPARK_HOME=/OPT/SPARK

c. En el terminal debemos lanzar el comando **bin/zeppelin-daemon.sh start** y

d. Abrimos el navegador y debemos escribir **la url localhost:8081**. De esta manera se abre el navegador de Zeppelin donde vamos a poder trabajar.

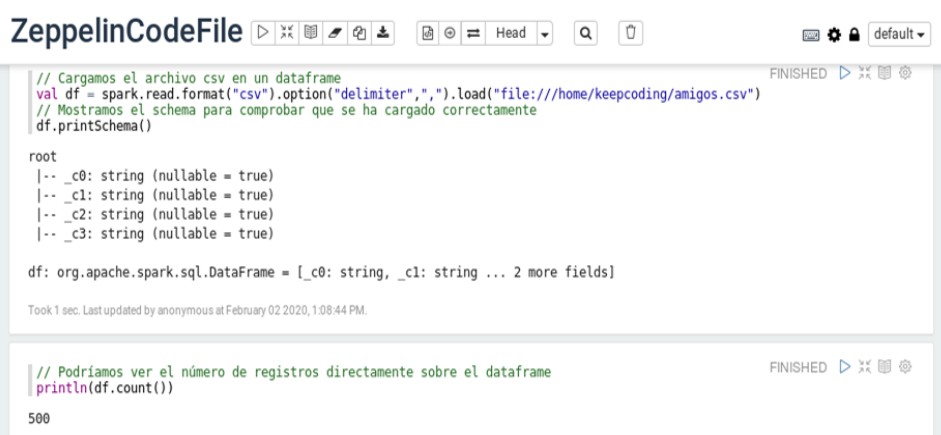
e. Modificamos la ruta para poder indicarle a SPARK donde tenemos nuestro zeppelin

name	value
SPARK_HOME	/home/keepcoding/spark-2.4.4-bin-hadoop2.7

PARTE DE CODIGO

Recordad que debemos **leer un fichero csv amigos.csv** y debemos mostrar cuantos amigos tenemos. En la solución se plantean dos maneras:

A. Trabajando directamente con el dataframe y que nos diga cuantos hay



```
// Cargamos el archivo csv en un dataframe
val df = spark.read.format("csv").option("delimiter", ",").load("file:///home/keepcoding/amigos.csv")
// Mostramos el schema para comprobar que se ha cargado correctamente
df.printSchema()

root
 |-- _c0: string (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)
 |-- _c3: string (nullable = true)

df: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 2 more fields]

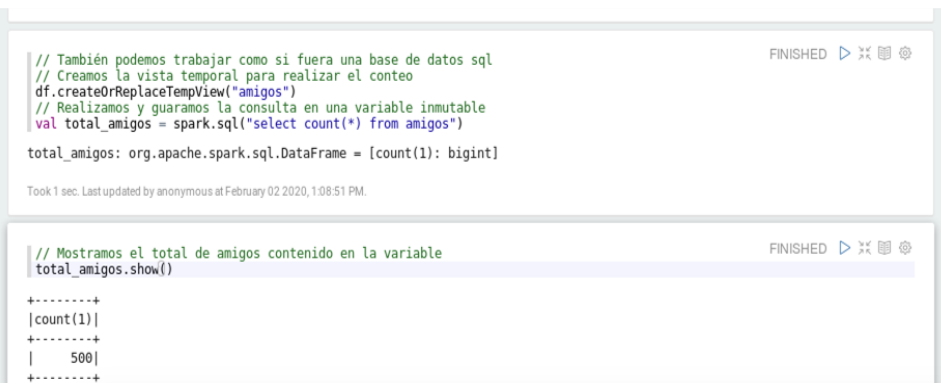
Took 1 sec. Last updated by anonymous at February 02 2020, 1:08:44 PM.

// Podríamos ver el número de registros directamente sobre el dataframe
println(df.count())

500
```

B.

Transformando el fichero en una tabla temporal, para poder trabajar directamente con consultas SELECT + COUNT



```
// También podemos trabajar como si fuera una base de datos sql
// Creamos la vista temporal para realizar el conteo
df.createOrReplaceTempView("amigos")
// Realizamos y guaramos la consulta en una variable immutable
val total_amigos = spark.sql("select count(*) from amigos")

total_amigos: org.apache.spark.sql.DataFrame = [count(1): bigint]

Took 1 sec. Last updated by anonymous at February 02 2020, 1:08:51 PM.

// Mostramos el total de amigos contenido en la variable
total_amigos.show()

+-----+
|count(1)|
+-----+
|    500|
+-----+
```