Ejercicio1 transferir una frase desde el terminal STREAMINGContext

Como servidor de *streaming*, vamos a utilizar el binario de UNIX netcat en modo contínuo y emitiendo en el puerto 9999:

1. EN EL TERMINAL DE LINUX

```
root@debian:/home/keepcoding# cat file| nc -l localhost.localdomain -p 9999
```

2. EN INTELLIJ

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
object ejemplo {

  def main(args: Array[String]): Unit = {


      val ssc = new StreamingContext(new
SparkConf().setMaster("local[2]").setAppName("socketstream"), Seconds(10))
      val hostname = "localhost"

      val mystreamRDD = ssc.socketTextStream(hostname, 9999)
      mystreamRDD.print()
      ssc.start()
      ssc.awaitTermination()
    }



}
```

Ejercicios2 con transformaciones

Flatmap +map <u>contra las palabras que tenemos en un fichero</u>

```
root@debian:/home/keepcoding# cat file| nc -l localhost.localdomain -p 9999
```

```
import org.apache.spark.SparkConf

import org.apache.spark.streaming.{Seconds, StreamingContext}

object ejemplo {


  def main(args: Array[String]): Unit = {
```

```
      val ssc = new StreamingContext(new
SparkConf().setMaster("local[2]").setAppName("socketstream"), Seconds(10))
      val hostname = "localhost"

      val mystreamRDD = ssc.socketTextStream(hostname, 9999)
   val words = mystreamRDD.flatMap(_.split(" ")) // for each line it split the words by
space
   val pairs = words.map(word => (word, 1))
   val wordCounts = pairs.reduceByKey(_ + _)
   wordCounts.print()
   ssc.start() // Start the computation
         ssc.awaitTermination() // Wait for termination


  }
```

Se detiene parando la ejecución con el cuadrado rojo de intellij

EJERCICIO3 CON FILTER

```
import org.apache.spark.SparkConf
import org.apache.spark.streaming.{Seconds, StreamingContext}
object ejemplo {

  def main(args: Array[String]): Unit = {
      val ssc = new StreamingContext(new
SparkConf().setMaster("local[2]").setAppName("socketstream"), Seconds(10))
      val hostname = "localhost"
      val mystreamRDD = ssc.socketTextStream(hostname, 9999)
```

```
        val words = mystreamRDD.flatMap(_.split(" "))

        val output = words.filter { word => word.startsWith("l") } // filter the words
starts with letter“s”

        output.print()


   ssc.start() // Start the computation

        ssc.awaitTermination() // Wait for termination



 }
```

• Nota: en el modo de salida podemos elegir 3 formatos :

• Modo Completo : se escribirá toda la tabla de resultados.

• Modo Anexar: solo se escribirán nuevas filas anexas. (Supongamos que las filas existentes no cambian.)

• Modo de actualización: se escribirán las filas actualizadas de la tabla de resultados.

EJERCICIO 4. Tratar ficheros CSV guardados en un directorio

```
import org.apache.spark.sql.SparkSession

import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}

object ejemplo {


  def main(args: Array[String]): Unit = {


    val sparkSession = SparkSession.builder

      .master("local")

      .appName("example")

      .getOrCreate()
```

```
  val schema = StructType(
    Array(StructField("nombre", StringType),
      StructField("edad", StringType),
      StructField("sexo", StringType)))

  //create stream from folder
  val fileStreamDf = sparkSession.readStream
    .schema(schema)
    .csv("/tmp/input")

  val query = fileStreamDf.writeStream
    .format("console")
    .outputMode("append")
    .start()
    .awaitTermination()
 }
}Los ficheros deben tener nombres distintos ¡!!
```

EJERCICIO5

Transmitir o leer continuamente un origen de archivo JSON desde una carpeta, procesarlo y escribir los datos en otro origen. Spark Streaming utiliza readStream para supervisar los archivos de carpeta y proceso que llegan al directorio en tiempo real y utiliza writeStream para escribir DataFrame o Dataset.

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}
object ejemplo {

 def main(args: Array[String]): Unit = {
```

```
  val spark:SparkSession = SparkSession.builder()
    .master("local[2]")
    .appName("ejemplo stream")
    .getOrCreate()

  import spark.implicits._
  val schema = new StructType().add("nombre","string").add("edad","long")
  val personasDF = spark.readStream
    .schema(schema)
    .json("/tmp/input")

  //se crea una vista para hacer alguna consulta a los datos en streaming
  personasDF.createOrReplaceTempView("personas")

  //se selecciona sexo y la media de edad por cada sexo.
  val mediaEdad = spark.sql("SELECT nombre, avg(edad) FROM personas GROUP BY
nombre")

  //la salida se realizará completa y por consola.

  val query = mediaEdad.writeStream
    .outputMode("complete")
    .format("console")
    .start()


 }

}
```

Ventanas Y WATERMARK

ventanas de tiempo: Tratamiento de los datos por un espacio temporal concreto. Las ventanas de tiempo se repiten por espacio de tiempo mayor o menor al tamaño de la ventana originando solapamiento de datos o perdidas.

//ejemplo del procesamiento para una ventana de 10 segundos procesada cada 5 seguntos.

val numPalabras = palabras.groupBy(window($"timestamp","10 seconds","5 seconds",$"value").count()

EJERCICIO 6 EJEMPLO DE VENTANA DE TIEMPO PROCESAR LOS DATOS para una ventana de 10 segundos procesada cada 5 seguntos.

```
root@debian:/home/keepcoding/IdeaProjects/TestScalaNuevo# cat file | nc -l local
host.localdomain -p 9999
```

```
import org.apache.spark.sql.SparkSession
import java.sql.Timestamp
import org.apache.spark.sql.functions._

object ejemplo {

  def main(args: Array[String]): Unit = {

    val spark = SparkSession.builder
      .master("local")
      .appName("example")
      .getOrCreate()
    import spark.implicits._

    val lines = spark.readStream
      .format("socket")
      .option("host", "localhost")
```

```
      .option("includeTimestamp", true)
      .option("port", 9999)
      .load()
    val palabras = lines.as[(String, Timestamp)].flatMap(line =>
      line._1.split(" ").map(word => (word, line._2)))
      .toDF("palabra", "timestamp")




// AGRUPAMOS LAS PALABRAS Y CREAMOS LA VENTANA DE TIEMPO
val windowCount = palabras.groupBy(
      window($"timestamp", "10 seconds", "5 seconds"),
      $"palabra") .count()
    //se realiza la query y se añade el parametro truncate para que no muestre
timestamps partidos
    val query = windowCount.writeStream.outputMode("update")
      .format("console")
      .option("truncate", false)
      .start()


    query.awaitTermination(90000)
query.stop()




  }
      }
```

Solución

Fichero hola mundo

Que tal

DEBEMOS VER QUE HOLA MUNDO QUE TAL SALE A 15 Y A 20 VUELVE A APARECER

```
vindow                                    |palabra|count|
-----------------------------------------+-------+-----+
[2020-01-25 16:19:05, 2020-01-25 16:19:15]|tal    |1    |
[2020-01-25 16:19:10, 2020-01-25 16:19:20]|que    |1    |
[2020-01-25 16:19:10, 2020-01-25 16:19:20]|hola   |1    |
[2020-01-25 16:19:05, 2020-01-25 16:19:15]|mundo  |1    |
[2020-01-25 16:19:05, 2020-01-25 16:19:15]|hola   |1    |
[2020-01-25 16:19:10, 2020-01-25 16:19:20]|tal    |1    |
[2020-01-25 16:19:10, 2020-01-25 16:19:20]|mundo  |1    |
[2020-01-25 16:19:05, 2020-01-25 16:19:15]|que    |1    |
-----------------------------------------+-------+-----+
```

EJERCICIO 7

Al ejemplo anterior añadimos WATERMARK →

watermark: Las marcas de agua se pensaron para esperar por los datos que llegan con retraso.

Los datos recuperados por las watermark se posicionan en el lugar que les correspondería de no haberse ocurrido el retraso. Se ha de definir un tiempo de actuación de watermark.

EJERCICIO En nuestro ejemplo, especificamos la marca de agua como 15 segundos. Así que spark esperará ese tiempo para eventos tardíos

withWatermark("timestamp", "15 seconds")

```
root@debian:/home/keepcoding/IdeaProjects/TestScalaNuevo# cat file | nc -l local
host.localdomain -p 9999
```

```
import org.apache.spark.sql.SparkSession

import java.sql.Timestamp

import org.apache.spark.sql.functions._


object ejemplo {
```

```
  def main(args: Array[String]): Unit = {


    val spark = SparkSession.builder
      .master("local")
      .appName("example")
      .getOrCreate()
    import spark.implicits._

    val lines = spark.readStream
      .format("socket")
      .option("host", "localhost")
      .option("includeTimestamp", true)
      .option("port", 9999)
      .load()
    val palabras = lines.as[(String, Timestamp)].flatMap(line =>
      line._1.split(" ").map(word => (word, line._2)))
      .toDF("palabra", "timestamp")

    //ahora se agrupa las palabras, se cuentan y se crean las ventanas.
    //añadiendo marca de agua
    val windowCount = palabras
                .withWatermark("timestamp", "15 seconds")
                .groupBy(
                window($"timestamp", "10 seconds", "5 seconds"),
                $"palabra")
                .count()


    //se realiza la query y se añade el parametro truncate para que no muestre
timestamps partidos
    val query = windowCount.writeStream.outputMode("update")
```

```
    .format("console")
    .option("truncate", false)
    .start()

  query.awaitTermination(90000)
}
    }
```