Ejercicios

Introducción

Para la ejecución de los ejercicios asociados a este curso, vamos a usar una máquina virtual proporcionada por Cloudera (cdh 5.5) genérica descargada directamente de su web "http://www.cloudera.com/downloads/quickstart_vms/5-5.html". Las características de la misma se pueden ver en la web. Si no estuviera disponible en la web a la hora de realizar estos ejercicios, será proporcionada por el instructor.

Para la resolución de los ejercicios podéis ayudaros de la documentación online de Spark https://spark.apache.org/docs/1.5.1/ (menú "Programming Guides", selecciona "Spark Programming Guide") y preguntar a vuestros compañeros.

Los ejercicios están prespaparados para ser resueltos en Scala, aunque es posible hacerlo también en Java y Python (bajo criterio personal de cada alumno).

Ejercicio: Usando el Shell de Spark (modulo 1)

El propósito de este ejercicio es trabajar con el Shell de Spark en Scala para leer un fichero en un RDD.

Tareas a realizar

1. Arrancar el Shell de Spark para scala y familiarizarse con la información que aparece por pantalla (Infos, Warnings, versión de Scala y Spark, etc...). Tarda un poco en arrancar.
   a. `spark-shell`
2. Comprobar que se ha creado un contexto "sc" tal y como vimos en la documentación
   a. `Escribir "sc"`
   b. Deberéis ver por pantalla algo de este tipo:

   res0:org.apache.spark.SparkContext= org.apache.spark.SparkContext@"*alfanum*"

3. Usando el comando de autocompletado sobre el SparkContext, podéis ver los métodos disponibles. La función de autocompletado consiste en presionar el tabulador después de escribir el objeto SparkContext seguido de un punto
4. Para salir del Shell, se puede escribir "exit" o presionar Cntrl+C

Ejercicio: Comenzando con los RDDs (módulo 2)

El objetivo de este ejercicio es practicar con los RDDs a través del Shell de Spark, para lo que usaremos ficheros externos.
Una práctica habitual en el análisis de datos en su etapa de pruebas es analizar ficheros pequeños que son subconjuntos similares a los datasets que se usarán en producción. En ocasiones, estos ficheros no se encuentran físicamente en ninguno de los nodos del cluster, por lo que es necesario importarlos de alguna manera.
Una forma sencilla de hacer estas transferencias entre nuestro Host y la MV/Cluster es a través de herramientas como Winscp https://winscp.net/eng/download.php
Otra opción es hacerlo como en los ejercicios pasados, a través de una carpeta compartida con la MV.

A- Exploración de fichero plano 1

Tareas a realizar

1. Inicia el Shell de Spark si te saliste en el paso anterior.
2. Para este ejercicio vamos a trabajar con datos en local
    a. Para acceder al fichero en local, se coloca delante de la ruta la palabra “file:”
3. Crea una carpeta llamada BIT en “/home” de forma que se cree la ruta “/home/BIT” y copia dentro de ella todos los ficheros de datos necesarios para el curso:
    a. Copiar la carpeta data_spark a la máquina virtual como en otras ocasiones y familiarizaos con su contenido
4. Dentro de data_spark se encuentra el fichero “relato.txt”. Copia este fichero en la máquina virtual en la siguiente ruta:
    a. `“/home/BIT/data/relato.txt”`
5. Visualiza el fichero con un editor de texto como “gedit” o “vi” o a través de la Shell con el comando “cat”
6. Crea un RDD llamado “relato” que contenga el contenido del fichero utilizando el método “textFile”
    a.
    ```
    val relato=sc.textFile("file:/home/BIT/data/relato.txt")
    ```
7. Una vez hecho esto, observa que aún no se ha creado el RDD. Esto ocurrirá cuando ejecutemos una acción sobre el RDD
8. Cuenta el número de líneas del RDD y observa el resultado. Si el resultado es 23 es correcto.
    a.
    ```
    relato.count()
    ```
9. Ejecuta el método “collect()” sobre el RDD y observa el resultado. Recuerda lo que comentamos durante el curso sobre cuándo es recomendable el uso de este método.

```
a. relato.collect()
```

10. Observa el resto de métodos aplicables sobre el RDD como vimos en el ejercicio anterior.
11. Si tienes tiempo, investiga cómo usar la función “foreach” para visualizar el contenido del RDD de una forma más cómoda de entender

```
a. relato.foreach(println)
```

A- Exploración de fichero plano 2

Tareas a realizar

1. Copia la carpeta weblogs contenida en la carpeta de ejercicios de Spark a “/home/BIT/data/weblogs/” y revisa su contenido.
2. Escoge uno de los ficheros, ábrelo, y estudia cómo está estructurada cada una de sus líneas (datos que contiene, separadores (espacio), etc)

```
- 128 [15/Sep/2013:23:59:53 +0100] "GET /KBDOC-00031.html HTTP/1.0" 200 1388 "http://www.loudacre.com"  "Loudacre CSR 116.180.70.237 Browser"
```

3. **116.180.70.237** es la IP, **128** el número de usuario y **GET /KBDOC-00031.html HTTP/1.0** el artículo sobre el que recae la acción.
4. Crea una variable que contenga la ruta del fichero, por ejemplo file:/home/BIT/data/weblogs/2013-09-15.log

```
a. val log= "file:/home/BIT/data/weblogs/2013-09-15.log"
```

5. Crea un RDD con el contenido del fichero llamada logs

```
a. val logs= sc.textFile(log)
```

6. Crea un nuevo RDD, jpglogs, que contenga solo las líneas del RDD que contienen la cadena de caracteres “.jpg”

```
a. val jpglogs=logs.filter(x=>x.contains(".jpg"))lo
```

7. Imprime en pantalla las 5 primeras líneas de jpglogs

```
a. jpglogs.take(5)s
```

8. Es posible anidar varios métodos en la misma línea. Crea una variable jpglogs2 que devuelva el número de líneas que contienen la cadena de caracteres “.jpg”
9. Ahora vamos a comenzar a usar una de las funciones más importantes de Spark, la función “map()”. Para ello, coge el RDD logs y calcula la longitud de las 5 primeras líneas. Puedes usar la función “*size()*” o “*length()*” Recordad que la función map ejecuta una función sobre cada línea del RDD, no sobre el conjunto total del RDD.

```
a. logs.map(x=>x.length).take(5)
```

10. Imprime por pantalla cada una de las palabras que contiene cada una de las 5 primeras líneas del RDD logs. Puedes usar la función “split()”

```
a. sc.textFile(logs).filter(line=>
   line.contains(".jpg")).count()
```

11. Mapea el contenido de logs a un RDD “logwords” de arrays de palabras de cada líneap
    a. `var logwords = logs.map(line => line.split(' '))`
12. Crea un nuevo RDD llamado “ips” a partir del RDD logs que contenga solamente las ips de cada línea (primer elemento de cada fila)
    a. `var ips = logs.map(line => line.split(' ')(0))`
13. Imprime por pantalla las 5 primeras líneas de ips
    a. `ips.take(5)`
14. Visualiza el contenido de ips con la función “collect()”. Verás que no es demasiado intuitivo. Prueba a usar el comando “foreach”
15. Crea un bucle “for” para visualizar el contenido de las 10 primeras líneas de ips. Ayuda: un bucle for tiene la siguiente sintaxis:

```
scala> for (x <- ips.take(10)
) { print(x) }
```

16. Guarda el resultado del bucle anterior en un fichero de texto usando el método saveAsTextFile en la ruta “/home/cloudera/iplist” y observa su contenido.
    a. `ips.saveAsTextFile("file:/home/cloudera/iplist")`

A- Exploración de un conjunto de ficheros planos en una carpeta

Tareas a realizar

1. Crea un RDD que contenga solo las ips de todos los documentos contenidos en la ruta “/home/BIT/data/weblogs”. Guarda su contenido en la ruta “/home/cloudera/iplistw” y observa su contenido.
    a. `sc.textFile("file:/home/BIT/data/weblogs/*").map(line => line.split(' ')(0))saveAsTextFile("file:/home/cloudera/iplistw")`
2. A partir del RDD logs, crea un RDD llamado “htmllogs” que contenga solo la ip seguida de cada ID de usuario de cada fichero html. El ID de usuario es el tercer campo de cada línea de cada log. Después imprime las 5 primeras líneas. Un ejemplo sería este:

    a. `var htmllogs=logs.filter(_.contains(".html")).map(line => (line.split(' ')(0),line.split(' ')(2)))`
    b. `htmllogs.take(5).foreach(t => println(t._1 + "/" + t._2))`

Ejercicio: Trabajando con PairRDDs (módulo 4)

El objetivo de este ejercicio es familiarizarnos con el trabajo con pares RDD.

A- Trabajo con todos los datos de la carpeta de logs: "/home/BIT/data/weblogs/*"

Tareas a realizar

a) Usando MapReduce, cuenta el número de peticiones de cada usuario, es decir, las veces que cada usuario aparece en una línea de un log. Para ello
   a. Usa un Map para crear un RDD que contenga el par (ID, 1), siendo la clave el ID y el Value el número 1. Recordad que el campo ID es el tercer elemento de cada línea. Los datos obtenidos tendrían que quedar de la siguiente manera

```
(userida, 1)
(userida, 1)
(useridb, 1)
```

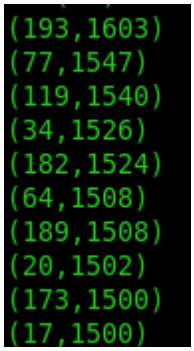
   b. Usa un Reduce para sumar los valores correspondientes a cada userid. Los datos tendría que mostrarse de la siguiente manera:

```
a)var logs=sc.textFile("file:/home/BIT/data/weblogs/*")
b)var userreqs = logs.map(line => line.split(' ')).map(words =>
(words(2),1)).reduceByKey((v1,v2) => v1 + v2)
```

b) Muestra los id de usuario y el número de accesos para los 10 usuarios con mayor número de accesos. Para ello:
   a. Utiliza un map() para intercambiar la Clave por el Valor, de forma que quede algo así: (Si no se te ocurre cómo hacerlo, investiga la función "swap()")

```
val swapped=userreqs.map(field => field.swap)
```

   b. Utiliza la función vista en teoría para ordenar un RDD. Ten en cuenta que queremos mostrar los datos en orden descendiente (De mayor a menor número de peticiones). Recuerda que el RDD debe estar en la misma forma que al inicio, es decir, con clave: userid y valor: nº de peticiones. El resultado debe ser:




```
swapped.sortByKey(false).map(field =>
field.swap).take(10).foreach(println)
```

c) Crea un RDD donde la clave sea el userid y el valor sea una lista de ips a las que el userid se ha conectado (es decir, agrupar las IPs por userID). Ayúdate de la función groupByKey() para conseguirlo, de manera que el resultado final sea algo así:

```
a. var userips = logs.map(line => line.split('
   ')).map(words => (words(2),words(0))).groupByKey()
        userips.take(10)
```

Si te sobra tiempo prueba a mostrar el RDD resultante por pantalla de forma que tenga una estructura como la siguiente:

```
ID:79844
IPS:
136.132.254.160
136.132.254.160
53.251.68.51
53.251.68.51
ID:16669
IPS:
23.137.191.64
23.137.191.64
ID:99640
IPS:
207.61.107.245
207.61.107.245
17.159.12.204
17.159.12.204
96.24.214.109
96.24.214.109
123.79.96.8
123.79.96.8
20.117.86.221
20.117.86.221
```

A- Trabajo con todos los datos de la carpeta de logs: "/home/BIT/data/accounts.cvs"

Tareas a realizar

1. Abre el fichero accounts.cvs con el editor de texto que prefieras y estudia su contenido. Verás que el primer campo es el id del usuario, que corresponde con el id del usua1rio de los logs del servidor web. El resto de campos corresponden con fecha, nombre, apellido, dirección, etc.
2. Haz un JOIN entre los datos de logs del ejercicio pasado y los datos de accounts.csv, de manera que se obtenga un conjunto de datos en el que la clave sea el userid y como valor tenga la información del usuario seguido del número de visitas de cada usuario. Los pasos a ejecutar son:
    a. Haz un map() de los datos de accounts.cvs de forma que la Clave sea el userid y el Valor sea toda la línea, incluido el userid. Obtendríamos algo de este tipo
        i.
```
var accounts =
sc.textFile("file:/home/BIT/data/accounts.csv").
map(line => line.split(',')).map(account =>
(account(0),account))
```

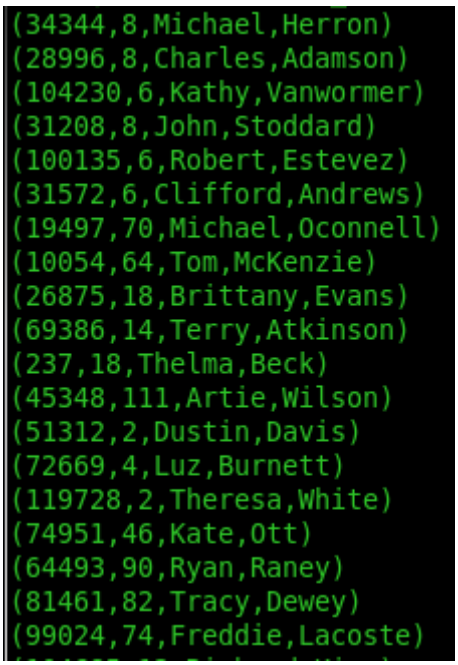
```
(userid1, [userid1, 2008-11-24 10:04:08, \N, Cheryl, West,
4905 Olive Street, San Francisco, CA, …])

(userid2, [userid2, 2008-11-23 14:05:07, \N, Elizabeth,
Kerns, 4703 Eva Pearl Street, Richmond, CA, …])

(userid3, [userid3, 2008-11-02 17:12:12, 2013-07-18
16:42:36, Melissa, Roman, 3539 James Martin Circle,
Oakland, CA, ...])
```

b. Haz un JOIN del RDD que acabas de crear con el que creaste en el paso anterior que contenía (userid, nº visitas), de manera que quede algo como esto:
   i. `var accounthits = accounts.join(userreqs)`

c. Crea un RDD a partir del RDD anterior, que contenga el userid, número de visitas, nombre y apellido de las 5 primeras líneas, para obtener una estructura como la siguiente
   i. `for (pair <- accounthits.take(10)) {println(pair._1,pair._2._2, pair._2._1(3),pair._2._1(4))}`



A- Trabajo con más métodos sobre pares RDD

Tareas a realizar

1. Usa keyBy para crear un RDD con los datos de las cuentas, pero con el código postal como clave (noveno campo del fichero accounts.CSV). Puedes buscar información sobre este método en la API online de Spark
   a. `var accountsByPCode = sc.textFile("file:/home/BIT/data/accounts.csv").map(_.split(',')).keyBy(_(8))`
2. Crea un RDD de pares con el código postal como la clave y una lista de nombres (Apellido, Nombre) de ese código postal como el valor. Sus lugares son el 5º y el 4º respectivamente.v=
   a. Si tienes tiempo, estudia la función “mapValues()” e intenta utilizarla para cumplir con el propósito de este ejercicio.
   b. `var namesByPCode = accountsByPCode.mapValues(values => values(4) + ',' + values(3)).groupByKey()`
3. Ordena los datos por código postal y luego, para los primeros 5 códigos postales, muestra el código y la lista de nombres cuyas cuentas están en ese código postal. La salida sería parecida a esta:

```
namesByPCode.sortByKey().take(10).foreach{
    | case(x,y) => println ("---" + x)
    | y.foreach(println)};

namesByPCode.sortByKey().take(10).foreach{
    | x => println ("---" + x._1)
    | x._2.foreach(println)};
```

```
--- 85003
Jenkins,Thad
Rick,Edward
Lindsay,Ivy
...
--- 85004
Morris,Eric
Reiser,Hazel
Gregg,Alicia
Preston,Elizabeth
...
```

Ejercicios opcionales: Trabajando con PairRDDs (módulo 4.1)

El objetivo de estos ejercicios es afianzar los conocimientos obtenidos sobre PairRDDs con ejercicios de una dificultad un poco mayor. Estos ejercicios vienen a reflejar un poco mejor los usos que podemos darle al Big Data.

EJ1: Tareas a realizar

1. Toma el dataset 'shakespeare' proporcionado por el profesor. Cópialo a la máquina virtual arrastrando y soltando la carpeta.
2. Utilizando la terminal, introduce dicho dataset en el HDFS (La ruta por defecto del HDFS es: hdfs://quickstart.cloudera:8020/user/cloudera, copia el dataset en dicha ruta en la carpeta 'shakespeare')
    1. hdfs dfs -put “rutadelacarpeta” user/cloudera
3. Ahora vamos a realizar un análisis al estilo de escritura de Shakespeare, para ello nos interesa conocer **las palabras más repetidas en sus obras.** Muestra por pantalla las 100 palabras que más veces aparecen en las obras de Shakespeare, junto con

la frecuencia de aparición de cada una, ordenadas descendientemente (de mayor a menor frecuencia de aparición).

```
val logs=sc.textFile("shakespeare/*")
Val logs2=logs.flatMap(line => line.split(" "))
Val logs3=logs2.map(word => (word,1)).reduceByKey(_+_)
Val logs4=logs3.map(word =>
word.swap).sortByKey(false).map(value.swap)
```

4. El análisis anterior no es todo lo completo que una situación real nos puede requerir, pues entre las palabras más repetidas encontramos: pronombres, artículos, proposiciones...; es decir, palabras que no nos aportan información alguna. A este tipo de palabras se les conoce como palabras vacías (StopWords), y nos interesa eliminarlas de nuestro análisis.

5. Para la realización de este ejercicio vamos a necesitar un dataset que contenga "stop words" en inglés (pues, como es lógico, los libros de Shakespeare están en este idioma). Copia el archivo "stop-word-list.csv" en la máquina virtual y cópialo al HDFS (en la ruta por defecto).

6. También tenemos que tener en cuenta que, para hacer nuestro análisis, no nos interesan ni las líneas que solo contengan espacios en blanco ni las palabras que estén formadas por una única letra. Tampoco haremos distinción entre mayúsculas ni minúsculas, por lo que, por ejemplo, "Hello" y "hello" deberán ser tomadas como la misma palabra en el análisis.

    1. El método para cambiar una línea a minúsculas es: toLowerCase
    2. La expresión regular que toma únicamente caracteres alfabéticos es: "[^a-zA-Z]+"
    3. Un archivo CSV está delimitado por comas.
    4. No nos interesa mostrar las ocurrencias de una única letra. Por lo que deberemos filtrar el RDD final.

```
val logs=sc.textFile("shakespeare/*")
Val  logs2=logs.map(line  =>  line.replaceAll("[^a-zA-Z]+","
"))
Val logs3=logs2.flatMap(line => line.split(" "))
Val logs4= logs3.map(word => word.toLowerCase)
Val stopwords =sc.textFile("stop-word-list.csv")
Val stopwords2=stopwords.flatMap(line => line.split(","))
Val stopwords3=stopwords2.map(word => word.replace(" ","")
Val logs5=logs4.substract(sc.parallelize(Seq(" ")))
Val logs6=logs5.substract(stopwords3)
Val logs7=logs6.map(word => (word,1)).reduceByKey(_+_)
Val logs8=logs7.map(word =>
word.swap).sortByKey(false).map(value.swap)
Val logs9=logs8.filter(word =>word._1.size !=1)
Logs9.take(20).foreach(println)
```

```
(shall,3738)
(king,3488)
(lord,3391)
(thee,3384)
(sir,3032)
(now,2947)
(good,2929)
(come,2600)
(ll,2490)
(here,2433)
(enter,2427)
(more,2425)
(well,2396)
(love,2376)
(man,2076)
(hath,2034)
(one,1934)
(upon,1843)
(know,1812)
(go,1769)
(make,1743)
(see,1541)
(such,1481)
```

EJ2: Tareas a realizar

1. Toma el dataset 'counts' proporcionado por el profesor. Cópialo a la máquina virtual arrastrando y soltando la carpeta.

2. Utilizando la terminal, introduce dicho dataset en el HDFS (La ruta por defecto del HDFS es: hdfs://quickstart.cloudera:8020/user/cloudera, copia el dataset en dicha ruta en la carpeta 'counts')

    1. hdfs dfs -put “rutaenlocal” user/cloudera/counts

3. El dataset que encontramos en la carpeta 'counts' contiene líneas en las que encontramos una palabra y un número separados por un espacio de tabulador. Este dataset viene a representar un archivo de frecuencias de palabras. A partir de estos datos: muestra por pantalla un RDD que se componga de pares clave-valor, siendo la clave una letra, símbolo o dígito y siendo el valor el nº de palabras cuya primera letra es la clave. PISTAS:

    1. La expresión regular que representa una expresión regular es: "\t".

2. No debemos contar como letras diferentes a mayúsculas y minúsculas.

Ejercicio: SparkSQL (JSON) (modulo 5.1)

EL objetivo de este ejercicio es familiarizarnos con el uso de la herramienta SQL de Spark.

Tareas a realizar

1. Crea un nuevo contexto SQLContext
   a. `var ssc = new org.apache.spark.sql.SQLContext(sc)`
2. Importa los implicits que permiten convertir RDDs en DataFrames
   a. `import sqlContext.implicits._`
3. Carga el dataset “zips.json” que se encuentra en la carpeta de ejercicios de Spark y que contiene datos de códigos postales de Estados Unidos. Puedes usar el comando “ssc.load(“ruta_fichero”, “formato”)”. Tiene este aspecto:

   a. `var zips = ssc.load("file:/home/BIT/data/zips.json", "json")`
4. Visualiza los datos con el comando “show()”. Tienes que ver una tabla con 5 columnas con un subconjunto de los datos del fichero. Puedes ver que el código postal es “_id”, la ciudad es “city”, la ubicación “loc”, la población “pop” y el estado “state”.
   a. `zips.show()`
5. Obtén las filas cuyos códigos postales cuya población es superior a 10000 usando el api de DataFrames
   a. `zips.filter(zips("pop") > 10000).collect()`

6. Guarda esta tabla en un fichero temporal para poder ejecutar SQL contra ella.
   a. `zips.registerTempTable("zips")`
7. Realiza la misma consulta que en el punto 5, pero esta vez usando SQL
   a. `ssc.sql("select * from zips where pop > 10000").collect()`
8. Usando SQL, obtén la ciudad con más de 100 códigos postales

ssc.sql("select city from zips group by city having count(*)>100 ").show()

9. Usando SQL, obtén la población del estado de Wisconsin (WI)

ssc.sql("select SUM(pop) as POPULATION from zips where state='WI'").show()

10. Usando SQL, obtén los 5 estados más poblados
    a. `ssc.sql("select * from zips where pop > 10000").collect()`

ssc.sql("select state, SUM(pop) as POPULATION from zips group by state order by SUM(pop) DESC ").show()

Ejercicio: SparkSQL (hive) (módulo 5.2)

El objetivo de este ejercicio es familiarizarnos con el uso de SparkSQL para acceder a tablas en Hive, dado que es una herramienta ampliamente extendida en entornos analíticos.

Tareas a realizar

1. Abrir una terminal y ejecutad este comando: “sudo cp /usr/lib/hive/conf/hive-site.xml /usr/lib/spark/conf/”. Para más información podéis consultar esta web “ https://community.cloudera.com/t5/Advanced-Analytics-Apache-Spark/how-to-access-the-hive-tables-from-spark-shell/td-p/36609 ”
2. Reiniciar el Shell de Spark.
3. En un terminal, arrancar el Shell de **Hive** y echar un vistazo a las bases de datos y tablas que hay en cada una de ellas.
4. En otro terminal aparte, arrancar el Shell de Spark, y a través de SparkSQL crear una base de datos y una tabla con dos o tres columnas. Si no creamocon Spark a través de Hive.
   bbdd: hivespark
   tabla: empleados
   columnas: id INT, name STRING, age INT
   config table: FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'

a. `val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)`
b. `sqlContext.sql("CREATE DATABASE IF NOT EXISTS hivespark")`
c. `sqlContext.sql("CREATE TABLE IF hivespark.empleados(id INT, name STRING, age INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")`

5. Crear un fichero "/home/cloudera/empleado.txt" que contenga los siguientes datos de esta manera dispuestos

```
1201, nombre1, 25
1202, nombre2, 28
1203, nombre3, 39
1204, nombre4, 23
1205, nombre5, 23
```

6. Aprovechando la estructura de la tabla que hemos creado antes, usando SparkSQL, subid los *datos del fichero "/home/cloudera/empleado.txt" a la tabla* hive, usando como la sintaxis de HiveQL como vimos en el curso de Hive (LOAD DATA LOCAL INPATH).
   a. `sqlContext.sql("LOAD DATA LOCAL INPATH '/home/cloudera/empleado.txt' INTO TABLE hivespark.empleados")`

7. Ejecutad cualquier consulta en los terminales de Hive y Spark para comprobar que todo funciona y se devuelven los mismos datos. En el terminal de Spark usad el comando "show()" para mostrar los datos.
   a. `Hive: select * from empleados;`
   b. `Spark: var query1=sqlContext.sql("SELECT * FROM hivespark.empleados")`
   c. `Spark: query1.show()`

Ejercicio: SparkSQL (DataFrames) (módulo 5.3)

El objetivo de este ejercicio es familiarizarnos un poco más con la API de DataFrames.

1. Creamos un contexto SQL
   a. `var ssc = new org.apache.spark.sql.SQLContext(sc)`
2. Importa los implicits que permiten convertir RDDs en DataFrames y Row
   a. `import sqlContext.implicits._`
   b. `import org.apache.spark.sql.Row`
   c. `import org.apache.spark.sql.types.{StructType,StructField,StringType}`
3. Creamos una variable con la ruta al fichero "/home/cloudera/Desktop/DataSetPartidos.txt". Será necesario copiar el dataset "DataSetPartidos.txt" al escritorio de la máquina virtual. Las líneas tienen el siguiente formato:
   `#idPartido::temporada::jornada::EquipoLocal::EquipoVisitante::golesLocal::golesVisitante::fecha::timestamp`
   a. `var ruta_datos="file:/home/cloudera/Desktop/DataSetPartidos.txt"`
4. Leemos el contenido del archivo en una variable
   a. `var datos =sc.textFile(ruta_datos)`
5. Creamos una variable que contenga el esquema de los datos
   a. `val schemaString = "idPartido::temporada::jornada::EquipoLocal::EquipoVisitante::golesLocal::golesVisitante::fecha::timestamp"`
6. Generamos el esquema basado en la variable que contiene el esquema de los datos que acabamos de crear
   a. `val schema = StructType(schemaString.split("::").map(fieldName => StructField(fieldName, StringType, true)))`
7. Convertimos las filas de nuestro RDD a Rows
   a. `val rowRDD = datos.map(_.split("::")).map(p => Row(p(0), p(1) , p(2) , p(3) , p(4) , p(5) , p(6) , p(7) , p(8).trim))`
8. Aplicamos el Schema al RDD
   a. `val partidosDataFrame = sqlContext.createDataFrame(rowRDD, schema)`
9. Registramos el DataFrame como una Tabla
   a. partidosDataFrame.registerTempTable("partidos")
10. Ya estamos listos para hacer consultas sobre el DF con el siguiente formato
    a. val results = sqlContext.sql("SELECT temporada, jornada FROM partidos")
    b. results.show()

11. los resulados de las queries son DF y soportan las operaciones como los RDDs normales. Las columnas en el Row de resultados son accesibles por índice o nombre de campo
    a. results.map(t => "Name: " + t(0)).collect().foreach(println)
12. Ejercicio: ¿Cuál es el record de goles como visitante en una temporada del Oviedo?
    a. Val recordOviedo = sqlContext.sql("select sum(golesVisitante) as goles, temporada from partidos where equipoVisitante='Real Oviedo' group by temporada order by goles desc")
    b. recordOviedo.take(1)
13. ¿Quién ha estado más temporadas en 1 Division Sporting u Oviedo?
    a. val temporadasOviedo = sqlContext.sql("select count(distinct(temporada)) from partidos where equipoLocal='Real Oviedo' or equipoVisitante='Real Oviedo' ")
    b. val temporadasSporting = sqlContext.sql("select count(distinct(temporada)) from partidos where equipoLocal='Sporting de Gijon' or equipoVisitante='Sporting de Gijon' ")

Ejercicios opcionales: Trabajando con SparkSQL (módulo 5.4)

El objetivo de este ejercicio es afianzar los conocimientos adquiridos con SparkSQL en un entorno real. Para ello tenemos un dataset con todos los episodios de los Simpsons, sus temporadas y su calificación en IMDB (entre otros parámetros).

EJ1: Tareas a realizar

1. Toma el dataset 'simpsons_episodes.csv' proporcionado por el profesor. Cópialo a la máquina virtual arrastrando y soltando el archivo.
2. Utilizando la terminal, introduce dicho dataset en el HDFS (La ruta por defecto del HDFS es: hdfs://quickstart.cloudera:8020/user/cloudera, copia el dataset en dicha ruta.)
    1. hdfs dfs -put “rutaenlocal” user/cloudera/counts
3. El dataset que tenemos es estructurado, es decir, sus columnas tienen título (como en una base de datos). Invierte un rato en familiarizarte con el *dataset* y sus respectivos campos.
4. El objetivo de este ejercicio es conseguir un diagrama lineal en el que podamos apreciar, gráficamente, la puntuación media de los capítulos de los Simpsons durante

todas sus temporadas. Este ejercicio constituye un posible caso real en el que podremos obtener información y realizar conclusiones mediante el análisis de multitud de datos.

5. Para cargar archivos 'csv' en Spark necesitamos utilizar esta librería, mientras que para crear el diagrama lineal a partir de los datos que obtengamos necesitamos esta otra. Para ejecutar Spark desde la terminal e indicarle las librería que queremos utilizar deberemos ejecutar el comando (IMPORTANTE EJECUTARLO SIN ESPACIOS):
    1. spark-shell --packages com.databricks:spark-csv_2.10:1.5.0,org.sameersingh.scalaplot:scalaplot:0.0.4
    2. Notar que al ejecutar el comando anterior, Spark cargará automáticamente las librerías, pues las descargará del repositorio Maven Central, donde ambas están alojadas.
6. Mientras carga la consola de Spark, notarás que le cuesta más tiempo y que ves más información por pantalla. Esto es debido a que Spark está descargando y añadiendo las librerías que le hemos indicado en el paso anterior. Una vez que Spark haya cargado, copia estos IMPORTS en la consola:
    1. import org.sameersingh.scalaplot.Implicits._

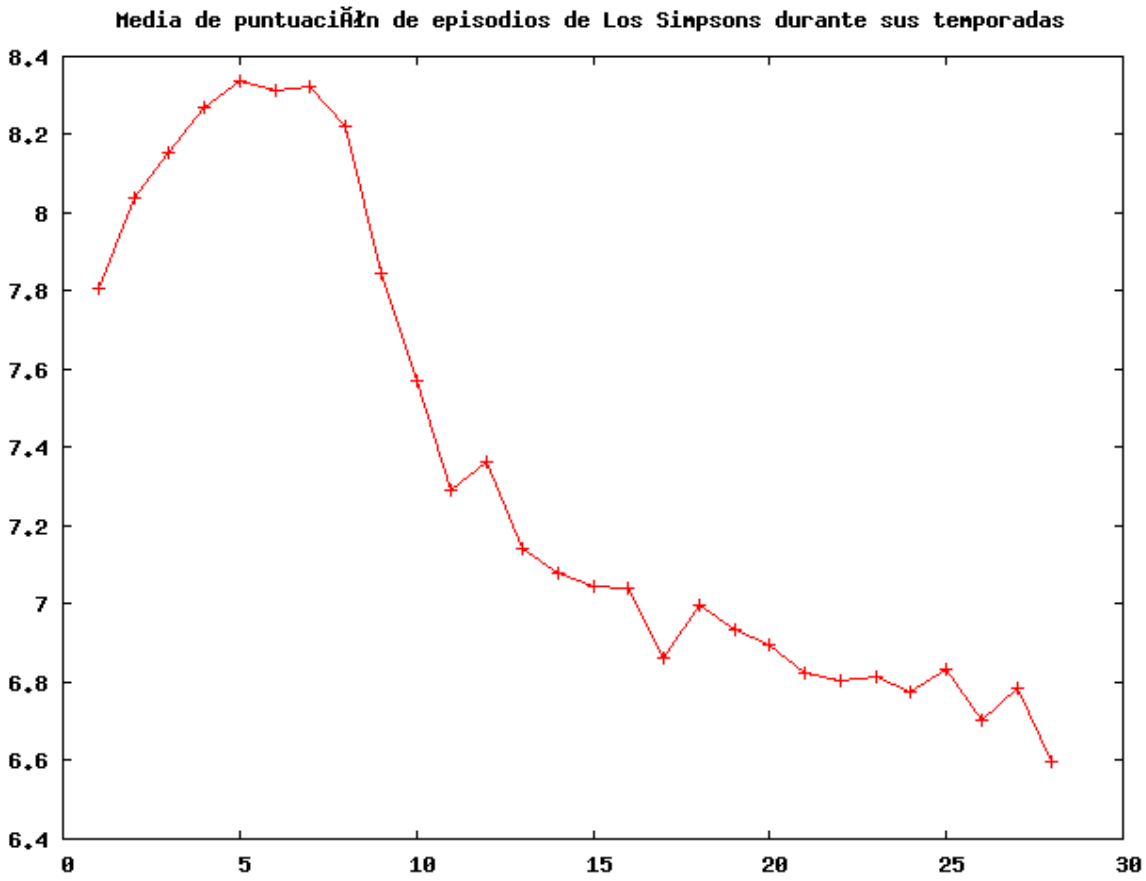
```
import org.sameersingh.scalaplot.MemXYSeries
import org.sameersingh.scalaplot.XYData
import org.sameersingh.scalaplot.XYChart
import org.sameersingh.scalaplot.gnuplot.GnuplotPlotter
```

7. Investiga en internet como transformar un archivo CSV en un Dataframe en Spark 1.6 y realiza la transformación.
    1. PISTA: Como has podido observar en el punto 5, para realizar la transformación hemos importado la librería "com.databricks.spark.csv"
        1. val df = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").option("inferSchema", "true").load("simpsons.csv")
8. Como los datos son estructurados, vamos a utilizar SparkSQL. El desarrollo a partir de aquí se puede realizar de diferentes formas, pero, como es lógico, la explicación del ejercicio se va a centrar en uno de ellos. En la teoría hemos visto un método para que Spark pueda ejecutar comandos SQL sobre el dataset, utilizalo para crear una tabla temporal:
    1. df.registerTempTable("NombreDeLaTablaVirtual")
9. Una vez que tenemos la tabla virtual creada, tendremos que crear un DataFrame con los datos: temporada y media de la puntuación de los episodios para esa temporada. Busca en los apuntes el comando para ejecutar código SQL en Spark y que el resultado sea un DataFrame. Una vez que tienes el comando anterior, ejecuta el código SQL necesario para obtener los datos buscados. Pistas:
        1. El campo "season" está en formato STRING, será necesario cambiarlo a INT.

2. El DataFrame resultante deberá estar ordenado de forma ASCENDENTE por número de temporada.
3. Te serán útiles las funciones predefinidas de SQL: CAST y MEAN.
4. val datos = sqlContext.sql("SELECT CAST(season as INT), MEAN(imdb_rating) FROM episodios GROUP BY season ORDER BY CAST(season as INT) ASC"

10. Una vez que ejecutes lo requerido en el punto anterior, muestra por pantalla el Dataframe resultante para asegurarte de que el resultado es el esperado.
    1. df.show() Siendo df el Dataframe resultante de la ejecución del punto anterior.
11. Como estamos realizando un caso real, hemos podido apreciar que los datos no están siempre en el formato que más nos interesa a nosotros (hemos tenido que cambiar el tipo de STRING a INT). Cambia el formato del Dataframe anterior a un PairRDD (RDD formado por una clave y un valor, siendo la clave el número de temporada y el valor la media de puntuación para esa temporada). Te serán útiles las funciones vistas en la teoría.
    1. val rdd=datos.rdd.map(line =>(line.getDouble(0),line.getDouble(1)))
    2. val rdd=datos.rdd.map(line =>(line.getInt(0),line.getDouble(1)))
12. Ya entramos en la fase final del ejercicio y queremos crear, con nuestros resultados, un diagrama lineal que represente la información obtenida, de forma que nos sea más sencillo extraer conclusiones. Ahora necesitamos dividir el RDD anterior, de forma que almacenaremos en una variable Y las diferentes temporadas que hemos analizado, mientras que almacenaremos en una variable X las puntuaciones medias de dichas temporadas. Notar que la variable 'rdd' es el nombre de la variable que contiene al PairRDD obtenido en el paso anterior.
    1. val x = rdd.map({case (key,value) => key.toDouble})
    2. val y = rdd.map({case (key,value) => value})
13. Ahora abre una terminal nueva. Asegurate que estás en la ruta: "/home/cloudera" y crea una carpeta nueva llamada "docs". Utiliza para ello el comando "mkdir".
14. En esa misma terminal, ejecuta el comando: *sudo yum install gnuplot* . Este comando instalará una utilidad necesaria para realizar el diagrama lineal.
15. De nuevo en la terminal de Scala, ejecuta el siguiente código para crear el diagrama lineal. Introduce cada línea de una en una para evitar problemas. Una vez ejecutado y, si todo ha ido bien, busca el resultado en la carpeta creada en el paso 13.

```
val series = new MemXYSeries(x.collect(), y.collect(), "puntuacion")
val data = new XYData(series)
val chart = new XYChart("Media de puntuación de episodios de Los Simpsons durante sus temporadas", data)
output(PNG("docs/", "test"), chart)
```


Media de puntuaciÃ³n de episodios de Los Simpsons durante sus temporadas
8.4
8.2
8
7.8
7.6
7.4
7.2
7
6.8
6.6
6.4
0
5
10
15
20
25
30

Ejercicio: Spark Streaming I (modulo 6.1)

El objetivo de este ejercicio es el de iniciarnos en el uso de Spark Streaming y observar sus cualidades. Para ello generaremos un script en un terminal que contará las palabras que introduzcamos en otra terminal a modo de streaming simulado.

Tareas a realizar

1. Visita en la web la documentación de Spark https://spark.apache.org/docs/1.5.2/streaming-programming-guide.html y familiarízate con el ejercicio. El objetivo es hacer lo mismo que pone en la web en el apartado "A Quick Example"
2. Tomate un tiempo para navegar por la web y explorar todo lo que puede ofrecerte. Cuando lo consideres, comienza el ejercicio:
3. Abre un terminal nuevo y escribe el siguiente comando: "nc -lkv 4444", que hace que todo lo que escribas se envíe al puerto 4444
4. Inicia un nuevo terminal y arranca el Shell de Spark en modo local con al menos 2 threads, necesarios para ejecutar este ejercicio: "spark-shell --master local[2]"
5. Por otro lado, accede al fichero "/usr/lib/spark/conf/log4j.properties", y edítalo para poner el nivel de log a ERROR, de modo que en tu Shell puedas ver con claridad el streaming de palabras contadas devuelto por tu script.
6. Importa las clases necesarias para trabajar con Spark Streaming
   a. `import org.apache.spark.streaming.StreamingContext`
   b. `import org.apache.spark.streaming.StreamingContext._`
   c. `import org.apache.spark.streaming.Seconds`
7. Crea un SparkContext con una duración de 5 segundos
   a. `var ssc = new StreamingContext(sc,Seconds(5))`
8. Crea un DStream para leer texto del puerto que pusiste en el comando "nc", especificando el hostname de nuestra máquina, que es "quickstart.cloudera"
   a. `var mystream = ssc.socketTextStream("localhost",4444)`
9. Crea un MapReduce, como vimos en los apuntes, para contar el número de palabras que aparecen en cada Stream
   a. `var words = mystream.flatMap(line => line.split("\\W"))`
   b. `var wordCounts = words.map(x => (x, 1)).reduceByKey((x,y) => x+y)`
10. Imprime por pantalla los resultados de cada batch
    a. `wordCounts.print()`
11. Arranca el Streaming Context y llama a awaitTermination para esperar a que la tarea termine
    a. `ssc.start()`
    b. `ssc.awaitTermination()`

```
spark-shell --master local[2]

import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.Seconds
val ssc = new StreamingContext(sc, Seconds(5))
val mystream = ssc.socketTextStream("quickstart.cloudera",4444)
val words = mystream.flatMap(line => line.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey((x,y) => x+y)
wordCounts.print()
ssc.start()
ssc.awaitTermination()
```

12. Una vez hayas acabado, sal del Shell y del terminal donde ejecutaste el comando “nc” haciendo un CNTRL+C
13. Para ejecutarlo desde un script:

```
spark-shell --master local[2] -i prueba.scala
```

Ejercicio: Spark Streaming II (modulo 6.2)

El objetivo de este ejercicio es seguir familiarizándonos con el concepto de DStream. Para ello vamos a simular accesos a web (los de la carpeta weblogs) y vamos a trabajar con ellos como si estuvieran ocurriendo en tiempo real, creando una aplicación Spark Streaming que capture aquellos que cumplan una característica específica que indicaremos más abajo.
Para ayudarnos de ello, se ha creado un script en Python que podéis encontrar en la carpeta de ejercicios de Spark. La primera tarea es copiar este script en la ruta "/home/BIT/examples/streamtest.py" (o en otra que tú elijas) que lo que hace es leer los ficheros contenidos en weblogs y simular un streaming de ellos (parecido a lo que hicimos con el comando "nc" en el ejercicio pasado, pero de manera automática y sobre un conjunto de ficheros)

Tareas a realizar

1. Abre un nuevo terminal, sitúate en la ruta "/home/BIT/examples" y ejecuta el script Python mencionado arriba de la siguiente manera

   ```
   python streamtest.py quickstart.cloudera 4444 5
   /home/BIT/data/weblogs/*
   ```

2. Copia el fichero "StreamingLogs.scalaspark" situado en la carpeta de ejercicios de Spark en "/home/BIT/stubs/StreamingLogs.scalaspark" y familiarízate con el contenido. El objetivo es ejecutar en el Shell cada una de las líneas que contiene más las que vamos a programar para este ejercicio.
3. Abre un nuevo terminal y arranca el Shell de Spark con al menos dos threads, como en el caso anterior (tal y como pone en el fichero). A continuación ejecuta los imports y crea un nuevo StreamingContext con intervalos de un segundo.
   a. Spark-shell –master local[2]
   b. `import org.apache.spark.streaming.StreamingContext`
   c. `import org.apache.spark.streaming.StreamingContext._`
   d. `import org.apache.spark.streaming.Seconds`
   e. `var ssc=new StreamingContext(sc, Seconds(5))`

4. Crea un DStream de logs cuyo host es "quickstart.cloudera" (también podéis usar "localhost") y cuyo puerto es "4444" (lo mismo que hemos indicado por parámetro al script Python anterior)
   a. `Var dstream=ssc.socketTextStream("quickstart.cloudera",4444)`
5. Filtra las líneas del Stream que contengan la cadena de caracteres "KBDOC"
   a. `val lineas =dstream.filter(x=>x.contains("KBDOC"))`

1. Para cada RDD, imprime el número de líneas que contienen la cadena de caracteres indicada. Para ello, puedes usar la función "*foreachRDD()*".
   a. `Var numero=foreachRDD(líneas,1)`
2. Guarda el resultado del filtrado en un fichero de texto en sistema de archivos local (créate una ruta en /home/cloudera/…)de la máquina virtual, no en hdfs y revisa el resultado al acabar el ejercicio.
   a. `Numero=ssc.saveAsTextFiles("/home/Cloudera/…")`
3. Para arrancar el ejercicio ejecuta los comandos start() y awaitTermination()
   a. `ssc.start()`
   b. `ssc.awaitTermination()`

EXTRA

4. Si te ha sobrado tiempo, prueba a ampliar el ejercicio de la siguiente manera
5. Cada dos segundos, muestra el número de peticiones de KBDOC tomando ventanas de 10 segundos. Ayúdate de la función "countByWindow" y recuerda el uso del checkpoint necesario para utilizar funciones de ventana. La idea es replicar el código hecho hasta ahora y añadir las líneas que hacen falta para cumplir con el objetivo marcado en este punto.
   a.
```
Val
countsByWindow=líneas.map(1).reduceByKeyAndWindow(x:i
nt, y:int) =>x+y, Seconds(2))
```
6. Inicia el StreamingContext y espera a que termine.
   a. `Ssc.start()`
   b. `Ssc.awaitTermination()`