Práctica 1 - Hadoop

Francisco Javier Sáez Maldonado, José Antonio Álvarez Ocete

Parte 1

Proceso

Describiremos a continuación los pasos que se han ido siguiendo para obtener los resultados finales de la práctica.

Instalamos java

```
yum install java-1.8.0-openjdk-devel
```

Cambiamos la versión por defecto que se usaba en las transparencias Incluso si instalamos la versión 1.7 se instalará la versión 1.8. Hemos de cambiar los paths a las versión 1.8 para que funciona correctamente:

```
export JAVA_HOME=/usr/lib/jvm/jre-1.8.0-openjdk
```

Editamos el archivo `/opt/hadoop/etc/hadoop/hadoop.env.sh`:

```
export JAVA_HOME=/usr/lib/jvm/jre-1.8.0-openjdk
```

Compilación

Tomamos el código que se ha proporcionado y lo compilamos utilizando la orden

```
bin/hadoop fs -cat /user/bigdata/compilar.bash | exec bash -s WordCount
```

Ejecución

Primero hay que iniciar el NameNode y el DataNode:

```
sbin/start-dfs.sh
```

Iniciar el ResourceManager y el NodeManager:

```
sbin/start-yarn.sh
```

Recuerda que hemos de subir el archivo utilizando:

```
/opt/hadoop/bin/hdfs dfs -put Quijote.txt /user/root
```

Lanzamos nuestro trabajo de MapReduce:

```
sudo /opt/hadoop/bin/hadoop jar WordCount.jar uam.WordCount Quijote.txt output/
```

Obtenemos en el directorio output la salida. En concreto, dos archivos:

- Un archivo SUCCESS indicando que la tarea ha sido exitosa.
- Un archivo `part-r-00000` que tiene la salida del programa que queríamos ejecutar.

Mostramos una parte del fichero para mostrar parte de la salida (la salida completa se puede encontrar en este archivo).

```
"Tablante", 1
"dichosa    1
"el 8
"y  1
"¡Oh,   1
(Y  1
(a  1
(al 1
(como   1
(creyendo   1
(de 2
(habiéndose 1
(por    2
(porque 2
(pues   1
(que    21
```

Como podemos comprobar, las palabras quedan con ciertos símbolos de puntuación que no nos interesa que estén para realizar el conteo correcto de palabras.

Modificación del programa

Para arreglar esto, solo debemos cambiar una linea en la función `Map` del archivo `WordCount.java`. En concreto, la dejamos de la siguiente forma:

```
StringTokenizer itr = new StringTokenizer(value.toString().
        toLowerCase().replaceAll("[^a-z ]", ""));
```

como vemos, hemos pasado las palabras a minúsculas usando `toLowerCase` y luego hemos eliminado todo aquello que no sean letras usando `replaceAll`.

Una vez realizada esta modificación, debemos compilar de nuevo el programa como lo hemos hecho anteriormente y, seguidamente, ejecutar de nuevo el programa java para que realice el conteo de palabras. En este caso, indicamos que la salida la realice sobre la carpeta `output2/` para tener las dos salidas por separado y poder compararlas. Obtenemos los dos mismos ficheros que anteriormente.

Hecho esto, utilizamos el archivo `script.py` que hemos creado usando `Python`, que toma todas las palabras que hay en los ficheros de salida y el número de veces que aparece cada una, las ordena y toma las 10 primeras para mostrarlas por pantalla.

El resultado que se obtiene es que las palabras son las mismas y prácticamente en el mismo orden, salvo una pequeña variación entre dos de ellas. Podemos verlo gráficamente en la siguiente figura:

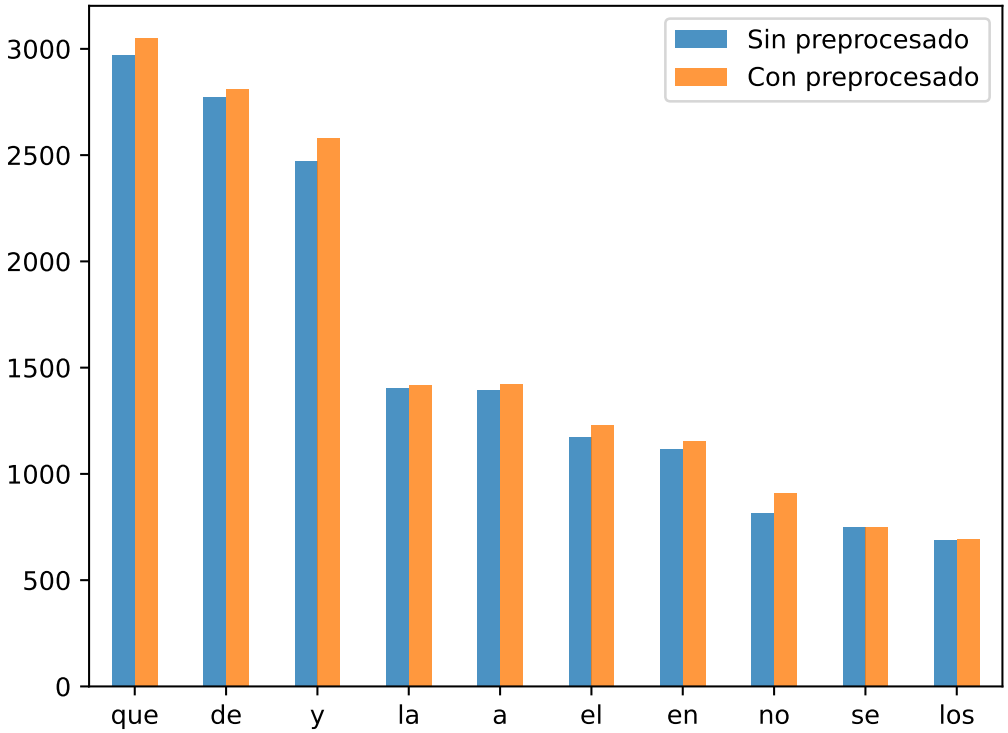

Figure 1: Comparación de palabras

Se puede apreciar que, al aplicar el preprocesado, la palabra a aparece un número de veces ligeramente superior al que lo hace cuando no tenemos el preprocesador previo del texto.

Comparación de resultados con otros ejemplos

TODO

Cuestiones planteadas

Pregunta 1.1. ¿ Qué ficheros ha modificado para activar la configuración del HDFS? ¿ Qué líneas ha sido necesario modificar?

Hemos modificado el fichero `/opt/hadoop-2.8.1/etc/hadoop/hadoop-env.sh` añadiendo la línea `export JAVA_HOME= /usr/lib/jvm/jre-1.7.0-openjdk` para especificar la instalación de `Java` que queremos utilizar

Como se explica en https://stackoverflow.com/questions/17569423/what-is-best-way-to-start-and-stop-hadoop-ecosystem-with-command-line, el script `stop-all.sh` detiene todos los daemons de Hadoop a la vez, pero está obsoleto. En lugar de eso es recomendable parar los daemons de HDFS y YARN por separado en todas las máquinas utilizando `stop-dfs.sh` y `stop-yarn.sh`.

A continuación, para instalar Hadoop pseudo-distribuido, hemos modificado el fichero `/opt/hadoop/etc/hadoop/core-site.xml`:

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

Y añadimos al fichero `/opt/hadoop/etc/hadoop/hdfs-site.xml` lo siguiente:

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

A continuación, hemos realizado la instalación del sistema pseudo-distribuido usando YARN, así que hemos modificado los siguientes ficheros **para configurar el uso de YARN (no de HDFS)**. `etc/hadoop/mapred-site.xml`:

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

Y también `etc/hadoop/yarn-site.xml`:

```
<configuration>
    <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>
</configuration>
```

Ejercicio 1.2: Para pasar a la ejecución de Hadoop sin HDFS, ¿ es suficiente con parar el servicio con `stop-dfs.sh`? ¿ Cómo se consigue ?

Pregunta 3.1: ¿ Dónde se crea hdfs ? ¿ Cómo se puede decidir su localización ?

El sistema de archivos HDFS se crea donde la variable `dfs.datanode.data.dir` indique. Esta variable se puede modificar en el archivo `hdfs-site.xml`. (Su valor por defecto)[https://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml] es

```
file://${hadoop.tmp.dir}/dfs/data
```

y, podemos ver que la variable `hadoo.tmp.dir` tiene el valor `/tmp/hadoop-${user.name}`.

Pregunta 3.2: ¿ Cómo se puede borrar todo el contenido del HDFS, incluido su estructura ?

Pregunta 3.3: Si estás usando hdfs, ¿ cómo puedes volver a ejecutar WordCount como si fuese single.node ?

Recordamos que hemos hecho una serie de cambios en archivos `xml` para configurar Hadood para que funcionase de forma pseudo-distribuida. Para ejecutarlo como si fuese *single-node*, deberíamos eliminar los cambios que hemos hecho en estos ficheros `xml`: `core-site.xml` y `hdfs-site.xml`. Así, volveríamos al modo por defecto de Hadoop y conseguiríamos que funcionase como si fuese `single-node`.

Pregunta 3.4: ¿ Cuáles son las 10 palabras más utilizadas ?

Esta pregunta ya ha sido mostrada anteriormente en la Figura 1. Las palabras más utilizadas son:

`que,de,y,la,a,el,en,no,se,los`

Pregunta 3.5: Cuántas veces aparecen las siguientes palabras: el, dijo

Para esta pregunta, volvemos a usar nuestro fichero `script.py` que nos imprimirá cuántas veces aparece cada una de ellas en el texto, primero usando preprocesado y a continuación sin utilizarlo.

```
Número de veces que aparece la palabra
     el
         - Sin preprocesado 1173
         - Con preprocesado 1228
     dijo
         - Sin preprocesado 196
         - Con preprocesado 271
```

Puede comprobarse ejecutando el script indicado.

Parte 2 : Tutorial de Spark

Seguimos el tutorial de spark y contestamos a las preguntas que se nos piden

Pregunta TS1.1 ¿Cómo hacer para obtener una lista de los elementos al cuadrado?

Bastará en este caso usar sobre el RDD la función map pasándole como parámetro la función `lambda x: x*x` que eleva el número al cuadrado. El código es este, con el resultado debajo:

```
numeros = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
cuadrados = numeros.map(lambda x : x*x)
print(cuadrados.collect())
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Pregunta TS1.2 ¿Cómo filtrar los impares?

El código se nos entrega resuelto en este caso. Es bastante sencillo, si usamos sobre el RDD la función `filter` dándole como parámetro la función que calcula si cada elemento es impar, esto es, si su módulo 2 es 1 : `lambda x : x%2 == 1`, nos da todos los impares. El código completo es:

```
rddi = numeros.filter (lambda e: e%2==1)
print (rddi.collect())
```

```
[1, 3, 5, 7, 9]
```

Pregunta TS1.3 ¿Tiene sentido esta operación? ¿Si se repite se obtiene siempre el mismo resultado?

Nos encontramos con el siguiente código:

```
#Tiene sentido esta operación?
numeros = sc.parallelize([1,2,3,4,5])

print (numeros.reduce(lambda elem1,elem2: elem1-elem2))
```

Esta operacion no tiene sentido, pues para hacer el reduce necesitamos que la operación sea conmutativa, es decir, que $a - b = b - a$, lo cual **NO** es cierto en todos los casos. Por eso además no producirá siempre los mismos resultados. Si ejecutamos en múltiples ocasiones veremos que los resultados no son los mismos siempre.

Pregunta TS1.4 ¿Cómo lo ordenarías para que primero aparezcan los impares y luego los pares?

Lo haremos para el caso general. Nos damos cuenta primero de que si el vector tiene n posiciones y tratamos de tomar $m > n$ posiciones, Spark se quedará con todas las posibles. Por tanto, lo primero que hacemos es obtener un entero con el máximo de elementos

```
numeros = sc.parallelize([3,2,1,4,5])

n = numeros.count()
```

Por último, basta ahora tomar elementos ordenados especificando cuál es el criterio de orden. En este caso, queremos que los elementos pares sean los más grandes, por lo que debemos pasarle como argumento la función `lambda x: x%2 == 0`. De esta manera, escribiendo la línea siguiente, obtenemos el resultado.

```
print(numeros.takeOrdered(n,lambda elem: elem%2== 0))

[3, 1, 5, 2, 4]
```

Pregunta TS1.5 ¿Cuántos elementos tiene cada rdd? ¿Cuál tiene más?

Nos encontramos con el siguiente código

```
lineas = sc.parallelize(['', 'a', 'a b', 'a b c'])

palabras_flat = lineas.flatMap(lambda elemento: elemento.split())
palabras_map = lineas.map(lambda elemento: elemento.split())

print (palabras_flat.collect())
print (palabras_map.collect())
```

Cuya salida es:

```
['a', 'a', 'b', 'a', 'b', 'c']
[[], ['a'], ['a', 'b'], ['a', 'b', 'c']]
```

Para ver cuántos elementos tiene cada uno, podemos hacer

```
print(palabras_flat.count())
print(palabras_map.count())

6
4
```

Claramente, `palabras_flat` tiene más. Esto ocurre porque en `flatMap`, cada elemento de entrada al que se le aplica la función `lambda`, puede tener como salida 0 o más (un vector) de elementos. Por tanto:

- Con `flatMap` para cada elemento de `lineas` obtenemos un vector que luego separaremos por elementos e ignoraremos los elementos que estén vacíos. Por ello, de 4 elementos iniciales pasamos a 6 que es el total de letras que tenemos en el RDD.
- Con `map`, a cada uno de los vectores iniciales se le aplica la función indicada y se introduce **tal cual** en el RDD resultante, no se separan los elementos de los vectores obtenidos como ocurría en el caso anterior.

Pregunta TS1.6 ¿De qué tipo son los elementos del rdd palabras_map? ¿Por qué palabras_map tiene el primer elemento vacío?

Siguiendo la respuesta de la pregunta anterior, en `palabras_map` los elementos obtenidos tras aplicar a cada elemento de `líneas` la función, son vectores, por lo que los elementos de `palabras_map` son vectores. Lo podemos ver en la salida del código

```
print (palabras_map.collect())
```

```
[[], ['a'], ['a', 'b'], ['a', 'b', 'c']]
```

Además, tiene el primer **elemento vacío** porque la función split aplicada sobre el elemento `''` devuelve un vector vacío pues no hay nada que separar.

Pregunta TS1.7. Prueba la transformación distinct si lo aplicamos a cadenas.

Para probarlo, realizamos el siguiente código en el que ponemos en un RDD ejemplos de prueba con las mismas palabras en mayúscula y minúscula, con espacios o signos de puntuación:

```
test = sc.parallelize(["abcd", "abcd","dcba","abCd","a bcd","hola","HOLA","hola!","HoLa","ho la"])

dis = test.distinct()

print(dis.collect())
```

La salida que obtenemos es:

```
['abcd', 'abCd', 'a bcd', 'hola', 'hola!', 'dcba', 'HOLA', 'HoLa', 'ho la']
```

Como podemos ver, teníamos un único ejemplo duplicado (el primero) que es el único que se ha eliminado. Esto nos indica que el comparador de elementos de los rdd comparan los strings *caracter a caracter*.

Pregunta TS1.8 ¿Cómo se podría obtener la misma salida pero utilizando una sola transformación y sin realizar la unión?

Se podría hacer que el filtro se quede con los elementos que empiecen por I o por E, haciendo el código del siguiente modo:

```
log = sc.parallelize(['E: e21', 'I: i11', 'W: w12', 'I: i11', 'W: w13', 'E: e45'])

inferr = log.filter(lambda elem: elem[0] in ['I','E'])


print(inferr.collect())
```

```
'E: e21', 'I: i11', 'I: i11', 'E: e45']
```

Pregunta TS1.9 ¿Cómo explica el funcionamiento de las celdas anteriores?

Comentamos una a una las celdas anteriores:

```
numeros = sc.parallelize([1,2,3,4,5])

print (numeros.reduce(lambda elem1,elem2: elem2+elem1))
```

Esta primera celda solo crea un RDD y luego hace la operación reduce, que combina los valores usando una función que le indicamos, en este caso la suma de los valores. Así, devolverá la suma de todos los valores ya que esta operación es conmutativa y asociativa.

```
#Tiene sentido esta operación?
numeros = sc.parallelize([1,2,3,4,5])

print (numeros.reduce(lambda elem1,elem2: elem1-elem2))
```

Esta celda hace lo mismo que la anterior salvo que la operación que realiza es la resta. Como ya hemos visto en una pregunta anterior, esta operación no tiene sentido pues la resta no es una operación conmutativa por lo que el resultado no siempre será el mismo.

```
palabras = sc.parallelize(['HOLA', 'Que', 'TAL', 'Bien'])

pal_minus = palabras.map(lambda elemento: elemento.lower())

print (pal_minus.reduce(lambda elem1,elem2: elem1+"-"+elem2))
#y esta tiene sentido esta operación?
# Qué pasa si ponemos elem2+"-"+elem1
```

En esta celda se crea un RDD que tiene cadenas de caracteres, primero se pasan a minúsculas aplicándoles la funcion `.lower` y luego se concatenan todas las palabras a un solo scring uniéndolas por guiones, obteniendo como salida:

```
hola-que-tal-bien
```

Si cambiásemos el orden en elq ue se concatenan los elementos como se nos indica en el comentario, la salida cambia y las palabras se van uniendo en el orden inverso, debido a cómo se realiza la operación reduce. La salida es:

```
bien-tal-que-hola
```

```
r = sc.parallelize([('A', 1),('C', 4),('A', 1),('B', 1),('B', 4)])
rr = r.reduceByKey(lambda v1,v2:v1+v2)
print (rr.collect())
```

En esta celda se crea un RDD que tiene como elementos Tuplas `(clave,valor)`. A continuación, usando `reduceByKey` pasándole como argumento una función suma, lo que se hace es sumar los valores de las tuplas cuya clave sea la misma, obteniendo la salida esperada:

```
[('C', 4), ('A', 2), ('B', 5)]
```

La última celda es la siguiente:

```
r = sc.parallelize([('A', 1),('C', 4),('A', 1),('B', 1),('B', 4)])
rr1 = r.reduceByKey(lambda v1,v2:v1+v2)
print (rr1.collect())
rr2 = rr1.reduceByKey(lambda v1,v2:v1)
print (rr2.collect())
```

De nuevo, se crea un RDD cuyos elementos son tuplas `(clave,valor)` , primero se realiza lo mismo que en la celda anterior, y luego se vuelve a aplicar sobre el RDD obtenido la función `reduceByKey`, esta vez pasándole como función simplemente mantenerla clave que tiene. Sin embargo, esto no parece producir ningún efecto sobre el RDD, pues cuando se realiza la operación `collect` para ver los elementos del mismo, la salida es la misma que se produce en la celda anterior.

Como **conclusión** a esta pregunta, podemos decir que es importante cómo se aplica la función `reduce` sobre los RDD y que hay que tener cuidado con las operaciones que indicamos a esta función pues podrían no producir los resultados que se desean.

Pregunta `groupByKey`

Dada la siguiente celda

```
r = sc.parallelize([('A', 1),('C', 2),('A', 3),('B', 4),('B', 5)])
rr = r.groupByKey()
res= rr.collect()

print(rr.collect())


# Que operación realizar al RDD rr para que la operacion sea como un reduceByKey
```

Lo priemro que vemos es que la operación `rr.collect` devuelve una lista con tuplas `(clave, pyspark Result Iterable)`. Estos iterables debemos pasarlos primero a una lista, y luego realizar sobre ellos la operación que quisiésemos hacer con el `reduceByKey`. Vemos que los pasamos a una lista utilizando `mapValues(list)` que convierte los valores a un tipo:

```
rrf = rr.mapValues(list).collect()
print(rrf)
```

```
[('C', [2]), ('A', [1, 3]), ('B', [4, 5])]
```

Y, a continuación, podemos realizar la operación que haríamos con el reduce. Por ejemplo, si queremos hacer la suma de los vectores utilizando `map`, podríamos hacer todo en una linea de la siguiente manera:

```
rrf = rr.mapValues(list).map(lambda x : (x[0],sum(x[1])))
print(rrf.collect())
```

```
[('C', 2), ('A', 4), ('B', 9)]
```

Ahora, se nos pide simular el `groupByKey` usando `reduceByKey` y `map`. Para ello, usando `reduceByKey` podemos obtener las listas que obtendríamos tras aplicar el `mapValues(list)` que hemos obtenido anteriormente.

```
simul_group = r.reduceByKey(lambda v1,v2: [v1,v2])
print(simul_group.collect())
```

```
[('C', 2), ('A', [1, 3]), ('B', [4, 5])]
```

Pregunta sobre `join`

```
rdd1 = sc.parallelize([('A',1),('B',2),('C',3)])
rdd2 = sc.parallelize([('A',4),('B',5),('C',6)])

rddjoin = rdd1.join(rdd2)
```

El resultado de esto es

```
[('A', (1, 4)), ('B', (2, 5)), ('C', (3, 6))]
```

Es decir, un nuevo RDD en el que los valores son tuplas con los valores que hay en cada uno de los RDD. Se nos pide que, dado ese código, cambiemos las claves de los dos RDDs iniciales para ver qué RDD se crea finalmente. Si lo hacemos, cambiando por ejemplo los RDD del siguiente modo:

```
rdd1 = sc.parallelize([('A',1),('B',2),('D',3)])
rdd2 = sc.parallelize([('A',4),('B',5),('E',6)])

rddjoin = rdd1.join(rdd2)
```

El resultado obtenido es el siguiente:

```
[('A', (1, 4)), ('B', (2, 5))]
```

Es decir, que el `join` está haciendo la unión de los elementos cuyas claves están en la intersección del conjunto de claves. Como D y E no están en ambos RDD, no están en la intersección del conjunto de claves y por tanto no se obtienen en el RDD final.

Tipos de Join

Se nos pregunta que qué ocurre cuando sustituimos `join` por `leftOuter/rightOuter/fullOuter` join. El resultado es que estos tipos de `join` crean elementos en el nuevo RDD aunque sus claves no estén en ambos RDD iniciales. En concreto:

- `leftOuter` añade al nuevo RDD también las tuplas `(clave,valor)` cuya clave esté en el RDD sobre el que se llama la función, pero no estén en el RDD que se pasa como parámetro. Se añade un `None` en la tupla conjunta del RDD final:

```
rdd1 = sc.parallelize([('A',1),('B',2),('C',3)])
rdd2 = sc.parallelize([('A',4),('A',5),('B',6),('D',7)])
rddjoin = rdd1.leftOuterJoin(rdd2)

[('A', (1, 4)), ('A', (1, 5)), ('B', (2, 6)), ('C', (3, None))]
```

- `rightOuter` hace lo mismo que el anterior pero en el sentido opuesto, es decir, usando el segundo RDD.
- `fullOuter` combina los dos anteriores

Pregunta TS1.10 Borra la salida y cambia las particiones en parallelize. ¿ Qué sucede ?

Lo que ocurre si ponemos 10 particiones es lo siguiente (borrando anteriormente el contenido del directorio donde tenemos la salida):

```
numeros = sc.parallelize(range(0,1000),10)
numeros.saveAsTextFile('salida')

%ls -la salida/*
```

```
-rw-r--r-- 1 root root 290 Oct  4 10:42 salida/part-00000
-rw-r--r-- 1 root root 400 Oct  4 10:42 salida/part-00001
-rw-r--r-- 1 root root 400 Oct  4 10:42 salida/part-00002
-rw-r--r-- 1 root root 400 Oct  4 10:42 salida/part-00003
-rw-r--r-- 1 root root 400 Oct  4 10:42 salida/part-00004
-rw-r--r-- 1 root root 400 Oct  4 10:42 salida/part-00005
-rw-r--r-- 1 root root 400 Oct  4 10:42 salida/part-00006
-rw-r--r-- 1 root root 400 Oct  4 10:42 salida/part-00007
-rw-r--r-- 1 root root 400 Oct  4 10:42 salida/part-00008
-rw-r--r-- 1 root root 400 Oct  4 10:42 salida/part-00009
-rw-r--r-- 1 root root   0 Oct  4 10:42 salida/_SUCCESS
```

Como se podía esperar, se crea un archivo para cada una de las particiones de salida, según el número de particiones que le hayamos indicado.

Procesamiento El Quijote

Pregunta TS2.1 Explica la utilidad de cada transformación y detalle para cada una de ellas si cambia el número de elementos en el RDD resultante. Es decir si el RDD de partida tiene N elementos, y el de salida M elementos, indica si N>M, N=M o N<M.

Dado el siguiente código:

```
charsPerLine = quijote.map(lambda s: len(s))
allWords = quijote.flatMap(lambda s: s.split())
allWordsNoArticles = allWords.filter(lambda a: a.lower() not in ["el", "la"])
allWordsUnique = allWords.map(lambda s: s.lower()).distinct()
sampleWords = allWords.sample(withReplacement=True, fraction=0.2, seed=666)
weirdSampling = sampleWords.union(allWordsNoArticles.sample(False, fraction=0.3))
```

Se nos pide indicar qué hace cada una de las transformaciones. Procedemos línea a línea:

1. `charsPerLine = quijote.map(lambda s: len(s))` esta línea obtiene un nuevo RDD en el que se cambian las líneas del quijote por la longitud de cada línea. No se modifica el número de elementos, se tiene $N = M$.
2. `allWords = quijote.flatMap(lambda s: s.split())` obtiene un nuevo RDD en el que se separa cada uno de los strings iniciales en cada una de sus palabras y luego cada palabra se convierte en un elemento del RDD. Por tanto, el RDD de salida tiene muchos más elementos que el de entrada, podemos decir que $M >> N$.
3. `allWordsNoArticles = allWords.filter(lambda a: a.lower() not in ["el", "la"])` se toma el RDD de las palabras y se pasa un filtro que elimina todos los artículos *el* y *la* (sin distinción de mayúsculas/minúsculas) del documento. Al eliminar elementos, el número de elementos en la salida será menor que en la entrada por lo que $M < N $.
4. `allWordsUnique = allWords.map(lambda s: s.lower()).distinct()` se crea un RDD que tiene un elemento por cada palabra distinta (sin distinción mayúsculas/minúsculas) que haya en el RDD `allWords`. De nuevo, se eliminan elementos repetidos por lo que el número de elementos en la salida será menor que en la entrada, $M < N$.
5. `sampleWords = allWords.sample(withReplacement=True, fraction=0.2, seed=666)` se extrae aleatoriamente con reemplazamiento un 20% de las palabras que hay en `allWords` para crear un nuevo RDD. Por supuesto, se tendrá que $M = 0.2N$ ($M < N$).
6. `weirdSampling = sampleWords.union(allWordsNoArticles.sample(False, fraction=0.3))` priemro extrae sin reemplazamiento un 30% de elementos que tenemos en el RDD que no tiene artículos (en este caso, tendríamos $M < N$ pues reducimos elementos) y a continuación une estos elementos extraídos al RDD que tiene una muestra del 20% que hemos obtenido en el paso anterior. En esta unión aumenta el tamaño del RDD por lo que en este paso $M > N$.