Lab1 Code Report

Adrian Ackva Giorgio Ruffa

Deliverables

The archive contains the following files

NoNameNoGrade_akva_ruffa_lab3/	Root folder
NoNameNoGrade_akva_ruffa_lab3/build.sbt	Sbt build script
NoNameNoGrade_akva_ruffa_lab3/KafkaSpark.scala	Source
NoNameNoGrade_akva_ruffa_lab3/CodeReport_Lab3_NoNameNoGrade_Ackva_Ruffa.pdf	This report

Compiling

Setup Compilation and Run

We consider kafka, cassandra and spark to be installed and running with default local configuration.

Compliation and execution of the program is handled using SBT with the commands:

```
$ sbt compile
$ sbt run
```

Issued in the folder containing the *KafkaSpark.scala* file.

Design Decisions

Methods signatures, the average problem

In contrast to what we did in lab1, we decided to fill in the code more freely and change some methods signatures to achieve the result.
In particular, we changed the "`mappingFunc`" signature in order to update a "`State[(Double,Double)]`" instead of a "`State[Double]`". Why? Because we thought that in order to keep calculating the average for each key it was needed to propagate both the count of all the elements processed and their sum, then the computation of the average becomes a trivial division.
We checked that the function was working correctly with some debugging printouts:

```
if (key == "t") // debugging on one key;
{
  println(s"Key: $key; Value: $sanitizedValue; ValueCount: $valueCount; ValueSum: $valueSum; NVC: $newValueCount; NVS: $newValueSum; avg: $avg")
}
```

```
Key: t; Value: 8.0; ValueCount: 0.0; ValueSum: 0.0; NVC: 1.0; NVS: 8.0; avg: 8.0
Key: t; Value: 5.0; ValueCount: 1.0; ValueSum: 8.0; NVC: 2.0; NVS: 13.0; avg: 6.5
Key: t; Value: 8.0; ValueCount: 2.0; ValueSum: 13.0; NVC: 3.0; NVS: 21.0; avg: 7.0
Key: t; Value: 13.0; ValueCount: 3.0; ValueSum: 21.0; NVC: 4.0; NVS: 34.0; avg: 8.5
Key: t; Value: 11.0; ValueCount: 4.0; ValueSum: 34.0; NVC: 5.0; NVS: 45.0; avg: 9.0
Key: t; Value: 18.0; ValueCount: 5.0; ValueSum: 45.0; NVC: 6.0; NVS: 63.0; avg: 10.5
Key: t; Value: 21.0; ValueCount: 6.0; ValueSum: 63.0; NVC: 7.0; NVS: 84.0; avg: 12.0
Key: t; Value: 19.0; ValueCount: 7.0; ValueSum: 84.0; NVC: 8.0; NVS: 103.0; avg: 12.875
Key: t; Value: 15.0; ValueCount: 8.0; ValueSum: 103.0; NVC: 9.0; NVS: 118.0; avg: 13.111111111111111
Key: t; Value: 23.0; ValueCount: 9.0; ValueSum: 118.0; NVC: 10.0; NVS: 141.0; avg: 14.1
Key: t; Value: 2.0; ValueCount: 10.0; ValueSum: 141.0; NVC: 11.0; NVS: 143.0; avg: 13.0
Key: t; Value: 21.0; ValueCount: 11.0; ValueSum: 143.0; NVC: 12.0; NVS: 164.0; avg: 13.666666666666666
Key: t; Value: 10.0; ValueCount: 12.0; ValueSum: 164.0; NVC: 13.0; NVS: 174.0; avg: 13.384615384615385
Key: t; Value: 7.0; ValueCount: 13.0; ValueSum: 174.0; NVC: 14.0; NVS: 181.0; avg: 12.928571428571429
Key: t; Value: 4.0; ValueCount: 14.0; ValueSum: 181.0; NVC: 15.0; NVS: 185.0; avg: 12.333333333333334
Key: t; Value: 20.0; ValueCount: 15.0; ValueSum: 185.0; NVC: 16.0; NVS: 205.0; avg: 12.8125
Key: t; Value: 4.0; ValueCount: 16.0; ValueSum: 205.0; NVC: 17.0; NVS: 209.0; avg: 12.294117647058824
```

As you can see, using this method the average converges quickly to a value around 12.5, which is exactly half of the alphabet size used by the generator, considering that zero is an emitted value.
In fact here are the results stored in cassandra:

word	count
z	12.53108
a	12.55414
c	12.53112
m	12.49513
f	12.52584
o	12.49067
n	12.57549
q	12.47512
g	12.43507
p	12.43247
e	12.57984
r	12.65608
d	12.54889
h	12.4541
w	12.46856
l	12.55136
j	12.52109
v	12.52391
y	12.60231
u	12.55327
i	12.47648
k	12.4061
t	12.46779
x	12.50393
b	12.51959
s	12.55352

(26 rows)

Miscellaneous

Direct connection

In this example we used a direct connection to kafka which has stronger guarantees compared to the receiver one. In addition to that we are not forced to allocate one thread only for the receiver itself. Compilation errors can be failrly kriptic so remember to set the required decoder for key and values (StringDecoder in this case).

Kafka key values

We were a little bit lost in the beginning because we failed to consider that kafka messages are key-value pairs themselves, but in this particular case the generator was emitting a every message with a null key. So we had to extract the value of the kafka message first and then split it in key-value pairs using the "`getTheTuple`" function defined in the code.

Received from kafka	Map to extract the value	Maps to generate the KV (getTheTuple) function
(null, "a,1")	"a,1"	(a, 1)

```
def getTheTuple(x:String) : (String, Double) = {
  val splitted = x.split(",")
  (splitted(0), splitted(1).toDouble)
}
val pairs = messages.map(x => x._2).map(getTheTuple)
```

At the end of this process we obtain a JavaPairdDStream which has the mapWithState function we can use.

State may not exists

Inside the mapping function with state, remember to check if the state exists! Because, as expected, the first value associated with a key does not have a starting state yet. The documentation of the State class gives some inspiration (please see Future Development section).

Checkpoint folder

As explained in the spark streaming programming guide, in order to store the state for each key, spark needs to have a checkpoint directory set. For convenience (laziness) we used the "/tmp/" folder.

Cassandra integration works by magic

It is kind of weird that Cassandra infers what to use as key and column when storing the DStream. It is true though that the DStream is made of K-V pairs after the map operation, but nevertheless he decided by itself where to put the value.

Future Developments

We are not very proud of this piece of code in the mappingFunc:

```
var valueCount = 0D
var valueSum = 0D
if (state.exists){ //Obviously,
  valueCount = state.get()._1
  valueSum = state.get()._2
}
```

Which is somehow “old style” and not very “scala-ish”. This, for sure, can be implemented using pattern matching and the “`getOption`” method of the State class, but our proficiency in scala is not there yet.

Conclusions

Having had some experience with Flink, we thought that the operation would have been much easier. The documentation (of Spark in general) is there to help you, but it is scattered around various sources and this time we found the official examples to be very useful. Understanding the usage of mapWithState was quite challenging at the beginning because the function is doing two things: returning a K-V pair (with the average as value in this case), and altering the state. It is kind of counterintuitive considering that, for what we have seen so far, the design of spark is to avoid inner function alters external objects as everything should be immutable.

Very Good Readings

https://spark.apache.org/docs/2.2.0/streaming-kafka-0-8-integration.html

https://spark.apache.org/docs/2.2.0/api/scala/index.html#org.apache.spark.streaming.api.java.JavaPairInputDStream

https://spark.apache.org/docs/2.2.0/api/scala/index.html#org.apache.spark.streaming.api.java.JavaMapWithStateDStream

https://github.com/apache/spark/blob/v2.2.0/examples/src/main/scala/org/apache/spark/examples/streaming/DirectKafkaWordCount.scala

https://spark.apache.org/docs/latest/streaming-programming-guide.html

https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/streaming/StateSpec.html

http://www.waitingforcode.com/apache-spark-streaming/stateful-transformations-mapwithstate/read

https://databricks.com/blog/2016/02/01/faster-stateful-stream-processing-in-apache-spark-streaming.html

https://spark.apache.org/docs/1.6.0/api/java/org/apache/spark/streaming/State.html