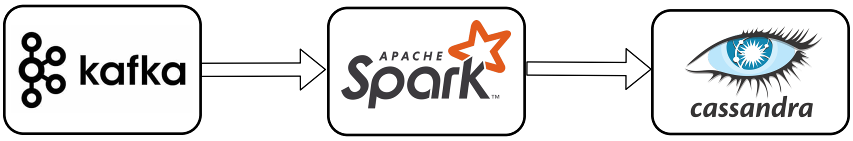
Data Intensive Computing – Lab Assignment 2

Adriano Mundo (mundo@kth.se), Riccardo Colella (colella@kth.se) - Group 29

October 12, 2020

1. Introduction

The purpose of the first part for this lab assignment is to practice with streaming processing using the tools studied during the course. Part 1 objective is to implement a *Spark Streaming* application which reads streaming data from *Kafka* and stores the results in *Cassandra*.
Spark Streaming works on top of `DStream` (sequence of `RDDs`) which represent a stream of data that can used as input data directly from *Kafka*.



2. How to Run

This section describes the steps needed to successfully run the `KafkaSpark.scala` application.

```
// environmental variable for Kafka
export KAFKA_HOME="/path/to/kafka/folder"
export PATH=$KAFKA_HOME/bin:$PATH

// environmental variable for Cassandra
export CASSANDRA_HOME="/path/to/cassandra/folder"
export PYTHONPATH="/path/to/python/path"
export PATH=$PYTHONPATH_HOME/bin:$CASSANDRA_HOME/bin:$PATH

// start ZooKeeper server and Kafka server
$KAFKA_HOME/bin/zookeeper-server-start.sh $KAFKA_HOME/config/zookeeper.properties
$KAFKA_HOME/bin/kafka-server-start.sh $KAFKA_HOME/config/server.properties

// create the topic avg with a single partition and one replica
$KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
--partitions 1 --topic avg

// start Cassandra in the foreground
$CASSANDRA_HOME/bin/cassandra -f

// go to the path and execute KakfaSpark.scala
cd ./path/to/the/folder/sparkstreaming
sbt run

// go to the path and execute Producer.scala
cd ./path/to/the/folder/generator
sbt run

// start Cassandra prompt and see the results
$CASSANDRA_HOME/bin/cqlsh
USE avg_space;
SELECT * FROM avg;
```

3. Implementation

The application goal is to read streaming data in the form of (key, value) pairs, calculate the average value of each key, continuously update and store the results while new pairs arrive. In this section we provide a detailed walk-through of the implemented code.

The code for the application resides in the `main` function of the object `KafkaSpark`. We assume that the all the import for the right classes have been done correctly.
First, we need to configure *Spark* to work. Thus, we create a new `SparkConf()`, set the master with two cores and the application name since it is mandatory. Next, we set the `StreamingContext` which is the main entry point for all *Spark Streaming* functionality, giving as parameter `Seconds(3)`. It represents the batches time interval of 3 seconds. We also set the checkpoint.

Next, we need to connect our application to the *Cassandra* datastore by using the commands `cluster.builder()` and `cluster.connect()`. Once we have the connection we can execute commands on the *Cassandra* instance with `session.execute()`. It has been used to create a keyspace called `avg_space` and a table called `avg` with two columns, text type for the primary key and float type for the average value.

The last part of this code snippet is related to the integration between *Kafka* and *Spark*. Here, we need to define the configuration parameters by using the `Map()` function. Then, we define the `Set` of the topic we created to run the application, called `avg`. It receives the data randomly generated from the Producer.

Finally, through the `createDirectStream` functionwe connect to *Kafka* with a receiver-less approach. It means that instead of using receivers, *Spark* periodically queries *Kafka* for the latest offset in each topic and partition defined as parameter. It is important to properly set the key and value classes and the key and value decoder classes. Moreover, we need to pass the context `ssc`, the configuration parameters `kafkaConf` and the `topicsSet` to the function.

```
object KafkaSpark {
  def main(args: Array[String]) {

  val conf = new SparkConf().setMaster("local[2]").setAppName("KafkaSparkCassandraAverage")
  val ssc = new StreamingContext(conf, Seconds(3))
  ssc.checkpoint("file:///tmp/spark/checkpoint")

  val cluster = Cluster.builder().addContactPoint("127.0.0.1").build()
  val session = cluster.connect()

  session.execute("CREATE KEYSPACE IF NOT EXISTS avg_space WITH REPLICATION = {'class' :
'SimpleStrategy', 'replication_factor': 1};")
  session.execute("CREATE TABLE IF NOT EXISTS avg_space.avg (word text PRIMARY KEY, count
float);")
  session.close()

  val kafkaConf = Map(
       "metadata.broker.list" -> "localhost:9092",
       "zookeeper.connect" -> "localhost:2181",
       "group.id" -> "kafka-spark-streaming",
       "zookeeper.connection.timeout.ms" -> "1000")

  val topicsSet = Set("avg")
  val messages = KafkaUtils.createDirectStream[String, String, StringDecoder,
StringDecoder](ssc, kafkaConf, topicsSet)
```

This second code snippet shows the pairs manipulation and the mappingFunc code used within the mapWithState.
In the first part we need to manipulate the (key, value) pairs generated by the Producer in the form of <key, “letter, number”>. It is important to retrieve the letter, number pairs by using a first map function which select the second element of the tuple and split the elements with “,”. Then, we used another map function to properly create the pairs from the array of strings generated previously.

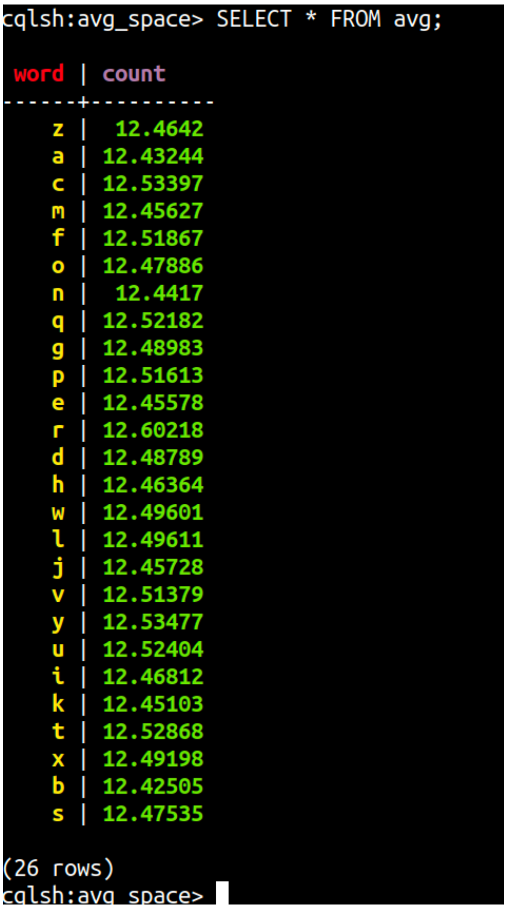
To calculate the average value of each key in a stateful manner we use the mapWithState function hving mappingFunc as supporting function. To store data in the table it can be used the saveToCassandra command which stores the word and average count in the avg table of the related avg_space keyspace. The last two lines start the computation and wait for the termination.

A more detailed explanation is deserved by the mappingFunc, which calculates the average value for each key. The data comes in (key, value) pairs; hence we decided to use a State(Double, Int) to save the state which is useful for storing the sum of all the values and the frequency for each key.
At the beginning we get the information from the previous state by using state.getOption.getElse((0.0, 0)). This retrieves the total sum and the count from the previous state.
Then, we need to compute the updated sum. In order to that, we add to the the previous state sum with the new value, retrieved with value.getOrElse(0.0). Now we update the frequency by adding one to the count and we store it in the newCount variable.
So, we have the variable newSum which stores the total sum for the key and the variable newCount which stores the related frequency. Now, we have to update the state by calling state.update and then we return, the key (letter) and the average computed by the division between the total sum and the frequency newSum/newCount.

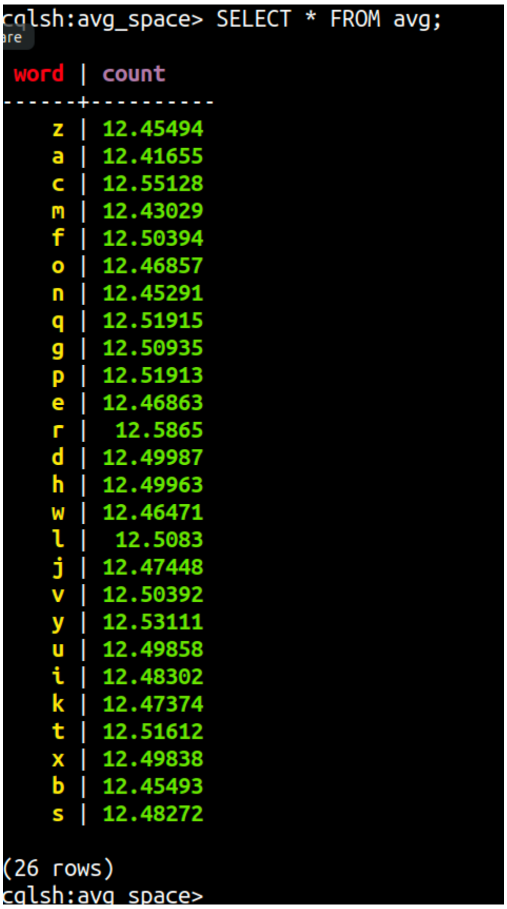
```
  val values = messages.map(message => message._2.split(","))
  val pairs = values.map(v => (v(0), v(1).toDouble))

  def mappingFunc(key: String, value: Option[Double], state: State[(Double, Int)]):
(String, Double) = {
        val (sum, count) = state.getOption.getOrElse((0.0, 0))
        val newSum = value.getOrElse(0.0) + sum
        val newCount = count + 1
        state.update((newSum, newCount))
        (key, newSum/newCount)
    }

   val stateDstream = pairs.mapWithState(StateSpec.function(mappingFunc _))

   stateDstream.saveToCassandra("avg_space", "avg", SomeColumns("word", "count"))

   ssc.start()
   ssc.awaitTermination()
  }
}
```

4. Results

In the pictures below, it is possible to see the final results generated by the query specified in the How to Run section within the Cassandra shell. The result shows the letter and the associated average. The results are continuously updated as the screenshots show, and the average value are correct since the function used by the Producer generates uniformly random value for the 26 alphabet values.



```
cqlsh:avg_space> SELECT * FROM avg;

 word | count
------+----------
    z |  12.4642
    a | 12.43244
    c | 12.53397
    m | 12.45627
    f | 12.51867
    o | 12.47886
    n |  12.4417
    q | 12.52182
    g | 12.48983
    p | 12.51613
    e | 12.45578
    r | 12.60218
    d | 12.48789
    h | 12.46364
    w | 12.49601
    l | 12.49611
    j | 12.45728
    v | 12.51379
    y | 12.53477
    u | 12.52404
    i | 12.46812
    k | 12.45103
    t | 12.52868
    x | 12.49198
    b | 12.42505
    s | 12.47535

(26 rows)
cqlsh:avg_space>
```



```
cqlsh:avg_space> SELECT * FROM avg;

 word | count
------+----------
    z | 12.45494
    a | 12.41655
    c | 12.55128
    m | 12.43029
    f | 12.50394
    o | 12.46857
    n | 12.45291
    q | 12.51915
    g | 12.50935
    p | 12.51913
    e | 12.46863
    r |  12.5865
    d | 12.49987
    h | 12.49963
    w | 12.46471
    l |  12.5083
    j | 12.47448
    v | 12.50392
    y | 12.53111
    u | 12.49858
    i | 12.48302
    k | 12.47374
    t | 12.51612
    x | 12.49838
    b | 12.45493
    s | 12.48272

(26 rows)
cqlsh:avg_space>
```