2020-10-12

ID2221 – Data Intensive Computing

Lab 2 – Spark Streaming, Apache Kafka, Apache Cassandra

Nagasudeep Vemula<vemula@kth.se >

Akhil Yerrapragada<akhily@kth.se >

Overview

Here we would like to explain design decisions taken when implementing **KafkaSpark.scala** and output generated when executing the file. Here, we read streaming data from Kafka, calculate the average, and store the calculated results in Cassandra. We now delve into the implementation in detail.

Producer:

Producer is responsible for generating keyValue pairs which serves as an input to Kafka. When initiated using **sbt run,** we see the below output:

```
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=z,5, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=s,7, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=q,3, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=q,3, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=n,19, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=v,21, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=u,12, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=q,7, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=w,23, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=x,5, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=f,8, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=p,2, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=t,18, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=w,17, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=f,10, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=b,19, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=i,17, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=a,4, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=k,14, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=z,4, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=u,15, timestamp=null)
ProducerRecord(topic=avg, partition=null, headers=RecordHeaders(headers = [], isReadOnly = true), key=null, value=a,14, timestamp=null)
```

KafkaSpark:

In KafkaSpark, we first establish connection to Cassandra and create a keyspace named **avg_space** and a table named **avg** in it. We now use **createDirectStream** which is a receiver less approach to allow spark to query latest offsets from Kafka. In **kafkaStream** we receive 1 Second microbatches from the topic **avg** when queried periodically.

```
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);")


// make a connection to Kafka and read (key, value) pairs from it
// connect Spark to Kafka in the receiver-less direct approach where Spark periodically queries Kafka for the latest offsets in each topic + partition

val sparkConf = new SparkConf().setAppName("KafkaSparkAverageValue").setMaster("local[2]")
val ssc = new StreamingContext(sparkConf, Seconds(1))

ssc.checkpoint(".")

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

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

As we observe in the producer output image, key is null and the value is in string format. Therefore, we first split the value from string to (String, Double) pairs and use **mapWithState** on them to calculate average in a stateful way.

```
val allValues = kafkaStream.map(_._2)

val splitEach = allValues.map(_.split(","))

val keyValue = splitEach.map(eachPair => (eachPair(0), eachPair(1).toDouble))
```

Now, to calculate average, we pass a function **mappingFunc** _ as an input to **mapWithState**. Below is the implementation to calculate average. We declare the state used for persistence as (Int, Double, Double) type, where Int is a counter, Double is the sum and Double is the average. Each time we receive a value for key, we increment the counter, add the received value to sum and perform the average calculation by dividing sum by counter.

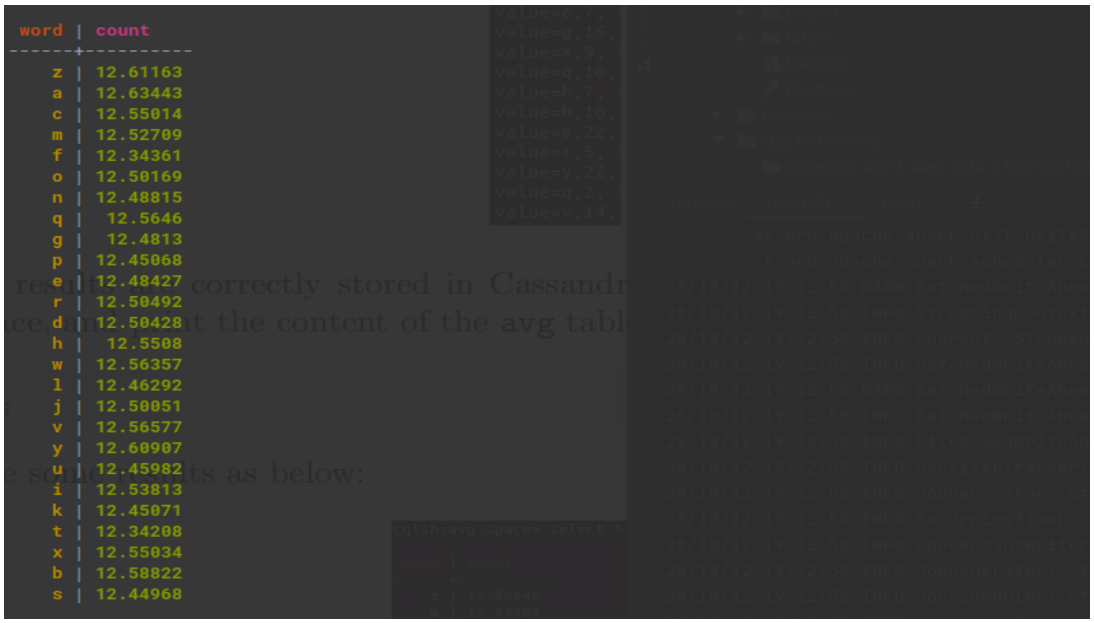
```
def mappingFunc(key: String, value: Option[Double], state: State[(Int, Double, Double)]): (String, Double) = {
  val newData = value.getOrElse(0.0);
  var (count, sum, average) = state.getOption.getOrElse((0, 0.0, 0.0))

  count = count + 1;
  sum = sum + newData;
  average = sum / count;

  state.update((count, sum, average))
  (key, average)
}
```

We finally update the state with the current count, sum and average and return the key and average which will be saved to Cassandra.

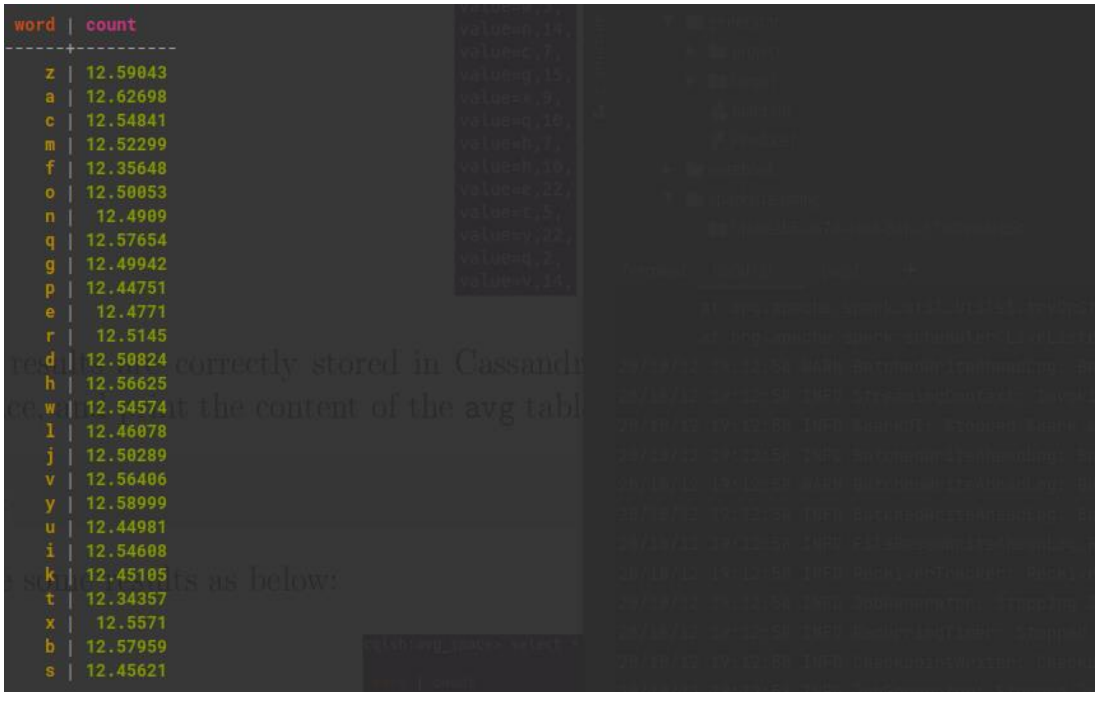
```
val stateDstream = keyValue.mapWithState(StateSpec.function(mappingFunc _))

// store the result in Cassandra
stateDstream.saveToCassandra("avg_space", "avg", SomeColumns("word", "count"))
```

Output:

Below pictures depicts the average being constantly updated to Cassandra:



word	count
z	12.61163
a	12.63443
c	12.55014
m	12.52709
f	12.34361
o	12.50169
n	12.48815
q	12.5646
g	12.4813
p	12.45068
e	12.48427
r	12.50492
d	12.50428
h	12.5508
w	12.56357
l	12.46292
j	12.50051
v	12.56577
y	12.60907
u	12.45982
i	12.53813
k	12.45071
t	12.34208
x	12.55034
b	12.58822
s	12.44968



word	count
z	12.59043
a	12.62698
c	12.54841
m	12.52299
f	12.35648
o	12.50053
n	12.4909
q	12.57654
g	12.49942
p	12.44751
e	12.4771
r	12.5145
d	12.50824
h	12.56625
w	12.54574
l	12.46078
j	12.50289
v	12.56406
y	12.58999
u	12.44981
i	12.54608
k	12.45105
t	12.34357
x	12.5571
b	12.57959
s	12.45621

```
 word | count
------+----------
    z | 12.58426
    a | 12.62902
    c | 12.54889
    m | 12.52118
    f | 12.36002
    o | 12.49493
    n | 12.48814
    q | 12.57079
    g | 12.49937
    p | 12.46272
    e |  12.4726
    r | 12.51837
    d | 12.51151
    h | 12.56126
    w | 12.53269
    l | 12.46397
    j | 12.49548
    v | 12.56502
    y | 12.57336
    u | 12.45146
    i | 12.54158
    k | 12.45153
    t | 12.34362
    x | 12.55901
    b |   12.5694
    s | 12.46097
```