Lab3 - Spark Streaming, Kafka and Cassandra

Group: Sandra_Anna

Sandra Pico Oristrell <sandrapo@kth.se>, Anna Hedström <annaheds@kth.se>

October 7, 2018

In this assignment, we have implemented a Spark Streaming application where the general idea is to have a Spark Streaming application reading streaming data from Kafka, and thereafter store the results in Cassandra.

1. Implementation

The following section walk-through the important parts of the code. In the 1.1 subsection, we elaborate on the main function and configuration code. In the 1.2 subsection, we comment on the implementation for the supporting function, i.e., mappingFunc. Please find the source code (in full and commented) in the Appendix 1: Source code.

1.1 Main

Below one can find the main function and configuration code for the KafkaSpark object.

- First, we need our Spark application to connect to Cassandra. We implement the integration between Spark-Cassandra by executing commands; Cluster.builder() and Cluster.connect().
- Next, we want to build a keyspace and a table in Cassandra. For this we use session.execute. Please note that keyspace is called avg_space and that table is called avg. The table has two columns, one of text type (the key) and the other of float type (the average value we wish to compute).
- For the Spark-Kafka integration, we need to define the Kafka parameters by means of a Map configuration.

Code 1: Main

```
object KafkaSpark {

  def main(args: Array[String]) {

    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);");

    val kafkaConf = Map(
      "metadata.broker.list" -> "localhost:9092",
      "zookeeper.connect" -> "localhost:2181",
      "group.id" -> "kafka-spark-streaming",
      "zookeeper.connection.timeout.ms" -> "1000"
    )
    val conf = new SparkConf().setMaster("local[*]").setAppName("KafkaAverageApplication")

    val ssc = new StreamingContext(conf, Seconds(1))
    ssc.checkpoint("file:///tmp/spark/checkpoint")

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

    val pairs = messages.map(x => (x._2).split(",")).map(r =>(r(0),r(1).toDouble))

    def mappingFunc(key: String, value: Option[Double], state: {
     // See next section.
    }

    val stateDstream = pairs.mapWithState(StateSpec.function(mappingFunc _))
    stateDstream.saveToCassandra("avg_space", "avg", SomeColumns("word", "count"))
    ssc.start()
    ssc.awaitTermination()
  }
}
```

- To configure `Spark`, we create a new `SparkConf()`, where we set the master to use as many threads as necessary (i.e., `setMaster("local[*]").`) to ensure best use of parallelism. Note that application name is mandatory in this stage, so we set it to `"KafkaAverageApplication"`. Next, we set the streaming context for `Spark`, using `StreamingContext`, where we pass the `Spark` configuration `conf` and the unit of time `Seconds(1)`. We also set a `checkpoint` for this.
- Now we focus on connecting the `KafkaUtils` to process the data. We have a `[key class], [value class], [key decoder class], [value decoder class]`. Using a receiver-less direct approach, to make `Spark Streaming` receive data from `Kafka`, we use `KafkaUtils.createDirectStream` and pass our `Spark` context i.e., `ssc`, our `Kafka` configuration i.e., `kafkaConf` as well as our parsed `topicsSet`. By using this approach - rather than letttig receivers get our data, we make `Spark` periodically query `Kafka` for the most recent offsets in the ”topic+partition” to get the ranges in each batch.
- The input streaming data is parsed in the format `(key, value)` as `String` and `Int`. Yet in our case, we note that each value contains both the letter and the count e.g., `<key,"z,15">`. Thus, for this reason, before we pass it to the supporting function, `mappingFunc`, we first need to ignore the incoming key and instead extract the letter as key and the associated number as value.
- Lastly, to then calculate the average value of each key in a stateful manner, we create `stateDstream` where we call our supporting function `mappingFunc` using `mapWithState`. Finally, to store the data from an `RDD` abstraction to `Cassandra` table (that is keyspace, table, columns), we thereafter use the `saveToCassandra` command.

1.2 `mappingFunc`

Below one can find the `mappingFunc`. The idea here is to calculate the average value per each key (i.e., the letter) in a stateful manner. While new pairs arrive, we want to continuously up date them and store the results in `Cassandra`.

- While we note that the streaming data comes in `(key, value)` pairs, here we want to calculate the average using just `State[Double]` because we only need to save the current average per each key in order to update the average every time. Nevertheless, another way to do it could be to have the state as: `State[Tuple2[Int,Double]]`, which saves the frequency of each letter and the average obtained in that moment. However, we decided to proceed with the first approach.
- First, we are using the function: `state.getOption.getOrElse(0.0)` to be able to catch the information from the previous state. At the same time, in case it is first time that the letter appears, the average value is set to 0.

- Next, we update the average into the `newState` using the previous average and the new value using the formula: `newState = (oldState + value.getOrElse(0.0))*0.5`
- Finally, we update the new average into the state using `state.update(newState)` and then, as requested in the assignment, we return `(key,newState)`.

Code 2: mappingFunc

```
def mappingFunc(key: String, value: Option[Double], state: State[Double]):
(String, Double) = {

    val oldState = state.getOption.getOrElse(0.0)
    val newState = (oldState + value.getOrElse(0.0))*0.5
    state.update(newState)

    return (key,newState)
}
```

2. Results

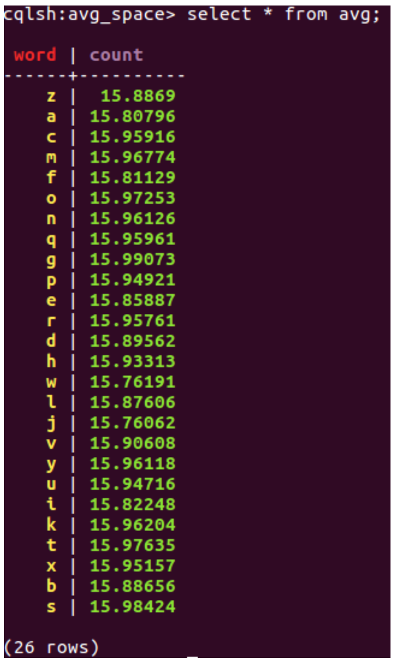
In the figure 1, one can view the final results as generated by the final query in the `cqlsh` prompt (as seen in the code snippet).

Code 3: Final `cqlsh` query

```
> $CASSANDRA_HOME/bin/cqlsh
> use avg_space;
> select * from avg;
```

In the result, we have row instances representing the letter, i.e., `word` as well as associated average we have calculated, i.e., `count`.

Figure 1: Final Output: Average number per each letter



```
cqlsh:avg_space> select * from avg;

 word | count
------+----------
    z |  15.8869
    a | 15.80796
    c | 15.95916
    m | 15.96774
    f | 15.81129
    o | 15.97253
    n | 15.96126
    q | 15.95961
    g | 15.99073
    p | 15.94921
    e | 15.85887
    r | 15.95761
    d | 15.89562
    h | 15.93313
    w | 15.76191
    l | 15.87606
    j | 15.76062
    v | 15.90608
    y | 15.96118
    u | 15.94716
    i | 15.82248
    k | 15.96204
    t | 15.97635
    x | 15.95157
    b | 15.88656
    s | 15.98424

(26 rows)
```

3. Getting Started

This section is dedicated to describe the steps needed to run the program. We assume a sucessful installation of Spark, Kafka and Cassandra.

Code 4: Run the program

```
    // Create environmental variables for Kafka and Cassandra.

    > export KAFKA_HOME="path/to/the/kafka/folder"
    > export PATH=$KAFKA_HOME/bin:$PATH
    > export CASSANDRA_HOME="/path/to/the/cassandra/folder"
    > export PYTHONPATH="/path/to/the/python/folder"
    > export PATH=$PYTHONPATH/bin:$CASSANDRA_HOME/bin:$PATH

    //Start Zookeper and Kafka in two different terminal tabs:

    > zookeeper-server-start.sh $KAFKA_HOME/config/zookeeper.properties
    > kafka-server-start.sh $KAFKA_HOME/config/server.properties

    //Create the avg topic:

    > kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
    --partitions 1 --topic avg

    //Start Cassandra in another terminal tab:

    > cassandra -f

    //Execute the KafkaSpark.scala file:

    > cd ./lab3/src/sparkstreaming
    > sbt run build.sbt

    //Generate the random data pairs using the generator file in another terminal tab:

    > cd ./lab3/src/generator
    > sbt run build.sbt

    //Once everything is running, you can check the results in Cassandra:

    > $CASSANDRA_HOME/bin/cqlsh
    > use avg_space;
    > select * from avg;
```

4. Appendix: Code

Code 5: Source Code

```
package sparkstreaming

import java.util.HashMap
import org.apache.kafka.clients.consumer.ConsumerConfig
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.streaming.kafka._
import kafka.serializer.{DefaultDecoder, StringDecoder}
import org.apache.spark.SparkConf
import org.apache.spark.streaming._
import org.apache.spark.streaming.kafka._
import org.apache.spark.storage.StorageLevel
import java.util.{Date, Properties}
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord, ProducerConfig}
import scala.util.Random
import org.apache.spark.streaming.{Milliseconds, StreamingContext, Time}

import org.apache.spark.sql.cassandra._
import com.datastax.spark.connector._
import com.datastax.driver.core.{Session, Cluster, Host, Metadata}
import com.datastax.spark.connector.streaming._

object KafkaSpark {

  def main(args: Array[String]) {

    //Connect to Cassandra and make a keyspace and table as explained
    //in assignment description.
    val cluster = Cluster.builder().addContactPoint("127.0.0.1").build()
    val session = cluster.connect()

    //Build a keyspace nad a Table in Cassandra.
    //KeySpace must be called: avg_space and Table must be called: avg
    session.execute("CREATE KEYSPACE IF NOT EXISTS avg_space WITH REPLICATION =" +
                    "{'class': 'SimpleStrategy', 'replication_factor': 1};")
    //Table has two columns: text(being the key) and float(being the average value).
    session.execute("CREATE TABLE IF NOT EXISTS avg_space.avg (word text PRIMARY KEY,
    count float);");

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

    //Spark configuration for streaming:
    val conf = new SparkConf().setMaster("local[*]").setAppName("KafkaAverageApplication")
    //Application name is mandatory in the configuration stage.

    //Streaming context from Spark.
    val ssc = new StreamingContext(conf, Seconds(1))
    ssc.checkpoint("file:///tmp/spark/checkpoint")

    //KafkaStream: keyclass, value class, key decoder class, value decoder class.
    val topicsSet = "avg".split(",").toSet
    val messages = KafkaUtils.createDirectStream[String,String,StringDecoder,StringDecoder]
    (ssc,kafkaConf,topicsSet)

    /*Messages are using the format: key,value. Each value contains the word and the count
    Example: <key,"z,15">. For this reason, we need to first ignore the key
    and just extract the word, being the letter, and the number, from the value. */
    val pairs = messages.map(x => (x._2).split(",")).map(x =>(x(0),x(1).toDouble))

    //Calculate the average value per each key in a Statful manner:
    def mappingFunc(key: String, value: Option[Double], state: State[Double]):
    (String, Double) = {
        val oldState = state.getOption.getOrElse(0.0)
        //Way to calculate the average using just State[Double]
        val newState = (oldState + value.getOrElse(0.0))*0.5
        state.update(newState)
        return (key,newState)
    }

    //Save Cassandra:
    val stateDstream = pairs.mapWithState(StateSpec.function(mappingFunc _))
    //Save the result in Cassandra Table: keyspace, table, columns:
    stateDstream.saveToCassandra("avg_space", "avg", SomeColumns("word", "count"))
    ssc.start()
    ssc.awaitTermination()
  }
}
```