READING STREAMING TWITTER FEEDS INTO APACHE SPARK

In part 1 of this blog post we explained how to read Tweets streaming off Twitter into Apache Kafka. Here we explain how to read that data from Kafka into Apache Spark. We broke this document into two pieces, because this second piece is considerably more complicated.

(This tutorial is part of our Apache Spark Guide. Use the right-hand menu to navigate.)

Prerequisites

First install Kafka as shown in Part 1 to verify that you can retrieve tweets from Twitter. Then start Kafka and run that Python program documented there. It will still be downloading Tweets into Kafka. Keep it running until you run the code below. Or just remember to start it before you run the code below.

It is difficult to get this code working mainly because you have to get the right versions of the jar files that will work with the version of Spark and Kafka you are using. Getting those mismatched will generate all kinds of hours you will spend hours debugging. It is not always clear what version to use. And sowing other confusing then there are sample code versions on the Apache Spark website that conflict with other versions.

So if you want to avoid those kinds of errors, use the exact versions that we have here, which are shown below. Or forge ahead with the new versions that will be delivered after this blog post is written, taking care to get compatible versions of each.

Now, install Kafka and Spark:

- Kafka 2.12-0.10.2.1

- Kafka-clients-0.10.2.1.jar (Gets installed when you install Kafka.)
- Spark-streaming-kafka-0-10_2.11-2.1.1.jar (You need to download this as explained below.)
- Apache Spark 2.1.1
- Scala 2.11.8 (You do not need to install Scala. It is installed when you install Spark.)

We will also be using sbt, which is a tool for compiling Scala code and resolving all dependencies, so you need to install that. Sbt is very complicated, but you do not have to master that. Just know that you need to put the code into this folder structure:

> /ProjectHome: build.sbt (explained below)
> /ProjectHome/src/main/scala/scala code (explained below).

Copy jar files

Now, the build.sbt file we write below will resolve all the import statements to make your code compile. But it will not install the Kafka and Spark-streaming-kafka jar files to where you need them. The easier way to fix that is copy them to $SPARK_HOME/jars as Spark can obviously find them there, in its own folder.

One of these files you copy from the Kafka lib folder:

> cp /usr/share/kafka/kafka_2.12-0.10.2.1/libs/kafka-clients-0.10.2.1.jar
> $SPARK_HOME/jars

The second one you download from the internet as it is neither part of Spark nor Kafka:

> wget https://spark.apache.org/docs/2.0.0-preview/streaming-kafka-integration.html

Now, in /ProjectHome create the file build.sbt and then copy the text shown below. It is not necessary to understand what it all means. But you are free to investigate that. Basically it indicates the external dependencies required and their version number. If you are a Scala or Java programmer you will know that those files are available on the internet at, for example here at Maven Central. There is where you get the syntax for the SBT dependencies which look something like this:

> libraryDependencies += "org.apache.spark" % "spark-streaming-kafka-0-10_2.11" % "2.1.1"

```
import AssemblyKeys._
name := "TwitterStream"
version := "1.0"
libraryDependencies ++= {
val sparkVer = "2.1.1"
Seq(
"org.apache.spark" %% "spark-core" % sparkVer,
"org.apache.spark" %% "spark-mllib" % sparkVer,
"org.apache.spark" %% "spark-streaming" % sparkVer,
"org.apache.spark" % "spark-streaming-kafka-0-10_2.11" % sparkVer,
"org.apache.kafka" % "kafka-clients" % "0.10.2.1"
)
}
scalaVersion := "2.11.8"
assemblySettings
```

```
mergeStrategy in assembly := {
case m if m.toLowerCase.endsWith("manifest.mf")      => MergeStrategy.discard
case m if m.toLowerCase.matches("meta-inf.*\\.sf$")  => MergeStrategy.discard
case "log4j.properties"                  => MergeStrategy.discard
case m if m.toLowerCase.startsWith("meta-inf/services/") =>
MergeStrategy.filterDistinctLines
case "reference.conf"                    => MergeStrategy.concat
case _                                   => MergeStrategy.first
}
```

The Scala code

Here we explain each section of the code, which we store in /ProjectHome/src/main/scala/TwitterStream.scala.

The first part are the imports. Again it is important to use these so they match the files imported with build.sbt to avoid compile errors.

```
import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.streaming.kafka010._
import
org.apache.spark.streaming.kafka010.LocationStrategies.PreferConsistent
import org.apache.spark.streaming.kafka010.ConsumerStrategies.Subscribe
import org.apache.spark.SparkConf
import org.apache.spark.streaming._
```

Here we create an **object** called **TwitterStream**. Sbt will create jar file that we will run with spark-submit. We need to give that class name to use, which in this case is **TwitterStream**.

kafkaParams gives the name or names of the Kafka server(s) where Kafka is running and which port. **Group id** is supposed to have some value, so we just take the value from an Apache example. The other parts tell it what objects to use to convert the data in Kafka, which in this case is **org.apache.kafka.common.serialization.StringDeserializer**.

The first values we create as a **SparkConf** and **StreamingContext**. We don't need to create a **SparkContext** as spark-submit will do that for us.

```
object TwitterStream {
def main(args: Array) {
val kafkaParams = Map(
"bootstrap.servers" -> "localhost:9092",
"key.deserializer" -> classOf,
"value.deserializer" -> classOf,
"group.id" -> "use_a_separate_group_id_for_each_stream",
"auto.offset.reset" -> "latest",
"enable.auto.commit" -> (false: java.lang.Boolean)
)
val sparkConf = new SparkConf().setAppName("tweeter")
val ssc = new StreamingContext(sparkConf, Seconds(2))
```

Now, in part 1 we decided to get Tweets that had the topic **trump**, as there are obviously plenty of those. Those have to be an **Array** to pass to the **KafkaUtils.createDirectStream** function.

That functions creates a **DStream**. For Spark Streaming this is designed to contained streaming data. You cannot actually do much with a DStream until you turn it into an RDD, which we do below. So this points to the irony that while these types products, like Spark, are called streaming, they obviously have to take some discrete, fixed chunk of data to work on it. That is what a RDD is. In fact, the DStream will create multiple RDDs.

```
val topics = Array("trump")
val stream = KafkaUtils.createDirectStream(
ssc,
PreferConsistent,
Subscribe(topics, kafkaParams)
)
```

KafkaUtils.createDirectStream creates a **org.apache.spark.streaming.dstream.DStream**.

To work with that data we use the **Dstream.foreach()** method to pluck off RDDs as they are created. Then we use **RDD.foreach()** to get each object in the RDD. Those objects will be Kafka **ConsumerRecords**, i.e., **org.apache.kafka.clients.consumer.ConsumerRecord**. You use the **ConsumerRecord.value()** method to read the message retrieved from the Kafka topic.

The Twitter messages are in JSON format, whose contents are described here. If you are doing analytics on this data you would not print it out. But we do that below to show that our sample is working.

ssc.start() ssc.awaitTermination() will start the streaming jon, which will continually run it until you kill it with (control-)(-c-).

```
stream.foreachRDD { rdd =>
rdd.foreach { record =>
val value = record.value()
val tweet = scala.util.parsing.json.JSON.parseFull(value)
val map:Map = tweet.get.asInstanceOf]
println(map.get("text"))
}
}
ssc.start()
ssc.awaitTermination()
}
}
```

To compile this code go to /ProjectHome and run:

```
sbt package
```

The first time you do that is will take quite a long time as it will download many items from the internet.

Run the job in Spark

Now you run the job using **$SPARK_HOME/bin/spark-submit** passing is the class name and the name of the jar file that sbt created, which will be **/ProjectHome/arget/scala-2.11/twitterstream_2.11-1.0.jar . local** means use x number of cpus. * means use all of them.

spark-submit \
--class TwitterStream \
--master **local** \
/home/walker/Documents/target/scala-2.11/**twitterstream_2.11-1.0.jar**

Now the program will run and output the Tweet text in a continuous stream. As you can see below it is creating RDDs as it runs too:

Some(RT @Cernovich: That baby ran away crying when @CassandraRules and I put cameras on him during the DNC. Totally irrelevant and has…)
Some(Trump Approval Rating Tanks: Majority Says He's 'Too Friendly With Russia' https://t.co/gUSH4oI2Rc via Christine Beswick)
17/06/10 12:08:39 INFO Executor: Finished task 0.0 in stage 3.0 (TID 3). 998 bytes result sent to driver
17/06/10 12:08:39 INFO TaskSetManager: Finished task 0.0 in stage 3.0 (TID 3) in 1344 ms on localhost (executor driver) (1/1)
17/06/10 12:12:59 INFO TaskSchedulerImpl: Removed TaskSet 14.0, whose tasks have all completed, from pool
17/06/10 12:12:59 INFO DAGScheduler: ResultStage 14 (foreach at TwitterStream.scala:32) finished in 1.147 s
17/06/10 12:12:59 INFO DAGScheduler: Job 14 finished: foreach at TwitterStream.scala:32, took 1.156825 s
17/06/10 12:12:59 INFO JobScheduler: Finished job streaming job 1497111178000 ms.0 from job set of time 1497111178000 ms
17/06/10 12:12:59 INFO JobScheduler: Total delay: 1.172 s for time 1497111178000 ms (execution: 1.166 s)
17/06/10 12:12:59 INFO KafkaRDD: Removing RDD 13 from persistence list