Day 3 – Lab2:

IoT

Introduction

In this example, we process real-world vehicle IoT data. Our data is in file `vehicle_1_1000.tsv`. This file contains first 1000 rows of the vehicle sensor data representing car movements.

The data stored is in a tab-separated file. The values represent observed position of the tracked vehicles

Data Schema

The data fields are:

- Col 1: Device ID (unique for each vehicle)
- Col 2: Time of the observation (in UTC)
- Col 3: Speed of the vehicle
- Col 4: The compass direction of the vehicle
- Col 5: The longitude of the GPS coordinates
- Col 6: The latitude of the GPS coordinates

Accuracy of GPS Coordinates

For GPS coordinates, about 100 meters accuracy is roughly coordinates rounded to 3 decimal places.

See GIS Accuracy for a detailed explanation.

The Goal

The goal is to count the number of times a car is observed parked at the same location with accuracy of about 100 meters.

The Solution

Let's get started! We will begin with the Spark Streaming program.

Spark Streaming Program

Our program will read the data from a Kafka stream and print the number of times a vehicle has been observed parked at the same location (within 100 meters).

Define Vehicle data

```
case class VehicleStr(id: String, timeUtc: String, speed: String, compassDir: String,
longitude: String, latitude: String)
```

Rounding Function for GPS Coordinates

```
  def roundAccuracy(coordinate: String, decimalPoints: Int): String = {
    val Array(intPart, decimalPart) = coordinate.split("\\.")
    intPart + "." + decimalPart.substring(0, decimalPoints)
  }
```

Function for Updating State

This is for the sate update at the end of each interval.

```
  /*
   * State update function. The state is a collection of keys and values.
   * Key: (id, long, lat)
   * Value: number of times parked
   */
  def updateParkedVehicles(batchTime: Time, key: (String, String, String),
      value: Option[Long], state: State[Long]):
    Option[((String, String, String), Long)] = {
    val noOfTimesParked = value.getOrElse(0L) + state.getOption.getOrElse(0L)
    val output = (key, noOfTimesParked)
    state.update(noOfTimesParked)
    Some(output)
  }
```

Key Parts of the `main` Function

See the comments in the code for explanations.

Configuration

```
    // Show only errors in console
    val rootLogger = Logger.getRootLogger()
    rootLogger.setLevel(Level.ERROR)

    // Consume command line parameters
    val Array(brokers, topics, interval) = args

    // Create Spark configuration
    val sparkConf = new SparkConf().setAppName("SparkKafkaIoT")

    // Create streaming context, with batch duration in ms
    val ssc = new StreamingContext(sparkConf, Duration(interval.toLong))
    ssc.checkpoint("./output")

    // Managing state
    val stateSpec = StateSpec.function(updateParkedVehicles _)

    // Create a set of topics from a string
    val topicsSet = topics.split(",").toSet

    // Define Kafka parameters
    val kafkaParams = Map[String, Object](
      "bootstrap.servers" -> brokers,
      "key.deserializer" -> classOf[StringDeserializer],
      "value.deserializer" -> classOf[StringDeserializer],
      "group.id" -> "use_a_separate_group_id_for_each_stream",
      "auto.offset.reset" -> "latest",
      "enable.auto.commit" -> (false: java.lang.Boolean))
```

Creating and Processing the Stream

```
    // Create a Kafka stream
    val stream = KafkaUtils.createDirectStream[String, String](
      ssc, PreferConsistent, Subscribe[String, String](topicsSet,kafkaParams))

    // Get messages - lines of text from Kafka
    val lines = stream.map(consumerRecord => consumerRecord.value)

    // Accuracy for GPS coordinates
    // https://gis.stackexchange.com/questions/8650/measuring-accuracy-of-latitude-
and-longitude
    val hundredMeters = 3

    // We round the observations within 100 meters
    val vehicleObservations = lines.map(_.split("\t")).
      map(strings => VehicleStr(strings(0), strings(1), strings(2), strings(3),
        roundAccuracy(strings(4), hundredMeters), roundAccuracy(strings(5),
hundredMeters)))

    val parkedVehicles = vehicleObservations.filter(_.speed == "0")

    val parkingCountsRaw = parkedVehicles.map(v => ((v.id, v.longitude, v.latitude),
1)).countByValue()

    // parkingCountsRaw are tuples with the structure:
    // (((id, long, lat), 1), timesParked)


    // Drop the "1" field
    // The structure is now a KV tuple ((id, long, lat), timesParked)
    val parkingCounts = parkingCountsRaw.map(pcr => ((pcr._1._1._1, pcr._1._1._2,
pcr._1._1._3), pcr._2))
```

Updating the State

```
    // Applying the state update
    val parkingCountsStream = parkingCounts.mapWithState(stateSpec)

    // The state represented as a DStream - we will use it to check the updates
    val parkingSnaphotsStream = parkingCountsStream.stateSnapshots()

    // This will print 10 values from the state DStream
    // Notice how the timesParked for a vehicle at a location accumulate as we
process more data
    parkingSnaphotsStream.print()

    // A real app may update a database which feeds a dashboard...
```

Starting the Stream Processing

```
    // Start stream processing
    ssc.start()
    ssc.awaitTermination()
```

Build and Package

In the terminal in the Spark program directory, run `sbt assembly`. This will create the jar file, which we will later deploy to Spark.

Run docker-compose

Run `docker-compose up` from the `docker` directory.

Create Topic

Create topic `vehicle-observations`

```
$ docker-compose exec kafka /opt/kafka/bin/kafka-topics.sh --create --zookeeper
zookeeper:2181 --replication-factor 1 --partitions 1 --topic vehicle-observations
```

Producer

Run the producer.

```
$ docker-compose exec kafka /opt/kafka/bin/kafka-console-producer.sh --broker-list
kafka:9092 --topic vehicle-observations
```

Deploy

```
$ mv target/scala-2.11/spark-kafka-iot.jar ../docker/spark/
```

Deploy into Spark. We will use 10 seconds as our interval.

```
$ docker-compose exec master spark-submit \
  --master spark://master:7077 \
  /app/spark-kafka-iot.jar \
  kafka:9092 vehicle-observations 10000
```

Run and Observe

Open the file `vehicle_1_1000.tsv`. Copy some rows from the file. Paste copied rows from the file in the producer. Observe the output in the Spark terminal. Do it a couple of times.
Our programs prints the state of the system (by default print will produce 10 lines).

You will see the output like this:

```
-------------------------------------------
Time: 1497802830000 ms
-------------------------------------------
((88,-97.409,27.765),8)
((111,-97.831,30.445),2)
((111,-97.832,30.445),2)
((111,-97.835,30.444),4)
((111,-97.834,30.445),2)
((120,-97.831,30.445),4)
```

```
((120,-97.831,30.446),2)
((88,-97.403,27.764),2)
((120,-97.830,30.446),58)
((88,-97.402,27.765),80)
```

This is stateful stream processing - notice how the state of the system gets updated with more data consumed. The last field in the tuple is the number of times we observed the vehicle parked at this location.

Done

Congratulations! You now have a Spark Streaming IoT program.