Federico Baldassarre - fedbal@kth.se 10 Oct 2017

Lab 3 - Data Intensive Computing

Stream Processing with Flink

Windows and Watermarks

In an ideal stream, every time the end of a window is hit, one can assume that all the records belonging to that window have arrived and hence the processing of that window can be started immediately. However, due to issues such as network latency and processing power, it can happen that records arrive past the closing time of a window. To solve this issue, the sources of these events can emit watermarks: when a watermark is received we can assume that all the records before the watermark have been observed. With this indication, it is enough to wait for the right watermark before processing the contents of a window.

Preprocessing

With this preprocessing, the latitude and longitude of each ride are converted into cell ids, then only the rides that started or ended in an terminal of the airport are kept.

```
private val extractLatLon = (ride: TaxiRide) =>
 if (ride.isStart)
   (ride.startLon, ride.startLat)
 else
   (ride.endLon, ride.endLat)

private val extractTerminal = (lon: Float, lat: Float) =>
 TerminalUtils.gridToTerminal(GeoUtils.mapToGridCell(lon, lat))

val maxDelay = 60 // events are out of order by max 60 seconds
val speed = 600 // events of 10 minutes are served in 1 second

val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
val rides = env.addSource(new TaxiRideSource("nycTaxiRides.gz", maxDelay,
speed))

val jfkRides = rides
 .map(extractLatLon)
 .map(extractTerminal.tupled)
 .filter(_ != Terminal_404)
```

Problem 1 - Number of taxis in each terminal by hour

The stream of ride events is grouped by terminal and windowed with a tumbling window of one hour. Then, every terminal-window pair is processed to output a tuple with the terminal, the ride events count and the start of the window in hours.

```
val windowedCountsByTerminal = jfkRides
 .keyBy(terminal => terminal)
 .window(TumblingEventTimeWindows.of(Time.hours(1)))
 .apply((terminal: Terminal, window, records, out: Collector[(Terminal, Int, Int)]) => {
   val cal: Calendar = Calendar.getInstance()
   cal.setTimeZone(TimeZone.getTimeZone("America/New_York"))
   cal.setTimeInMillis(window.getStart)
   val hour = cal.get(Calendar.HOUR_OF_DAY)
   out.collect((terminal, records.size, hour))
 })
windowedCountsByTerminal.print()
```

Problem 2 - Busiest terminal by hour

In order to find the busiest terminal for every hour, the stream of rides is windowed without grouping on the terminal id. Then, for every window a count of rides per terminal is computed. Finally, only the terminal with the highest rides count is emitted.

```
val windowedBusiestTerminal = jfkRides
 .timeWindowAll(Time.hours(1))
 .apply((window, records, out: Collector[(Terminal, Int, Int)]) => {
   val cal: Calendar = Calendar.getInstance()
   cal.setTimeZone(TimeZone.getTimeZone("America/New_York"))
   cal.setTimeInMillis(window.getStart)
   val hour = cal.get(Calendar.HOUR_OF_DAY)
   val counts = records
     .foldLeft(HashMap.empty[Terminal, Int].withDefaultValue(0))((acc, terminal) => {
       acc(terminal) += 1
       acc
     })
   val busiestTerminal = counts.maxBy(_._2)
   out.collect((busiestTerminal._1, busiestTerminal._2, hour))
 })
windowedBusiestTerminal.print()
```

JfkTerminals.scala

```scala
package lab3.problems

import java.util.{Calendar, TimeZone}

import com.dataartisans.flinktraining.exercises.datastream_java.datatypes.TaxiRide
import com.dataartisans.flinktraining.exercises.datastream_java.sources.TaxiRideSource
import com.dataartisans.flinktraining.exercises.datastream_java.utils.GeoUtils
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.util.Collector

import scala.collection.mutable.HashMap

object JfkTerminals {
  private val extractLatLon = (ride: TaxiRide) =>
    if (ride.isStart)
      (ride.startLon, ride.startLat)
    else
      (ride.endLon, ride.endLat)

  private val extractTerminal = (lon: Float, lat: Float) =>
    TerminalUtils.gridToTerminal(GeoUtils.mapToGridCell(lon, lat))

  def main(args: Array[String]) {
    val maxDelay = 60 // events are out of order by max 60 seconds
    val speed = 600 // events of 10 minutes are served in 1 second

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    val rides = env.addSource(new TaxiRideSource("nycTaxiRides.gz", maxDelay, speed))

    val jfkRides = rides
      .map(extractLatLon)
      .map(extractTerminal.tupled)
      .filter(_ != Terminal_404)

    val windowedCountsByTerminal = jfkRides
      .keyBy(terminal => terminal)
      .window(TumblingEventTimeWindows.of(Time.hours(1)))
      .apply((terminal: Terminal, window, records, out: Collector[(Terminal, Int, Int)]) => {
        val cal: Calendar = Calendar.getInstance()
        cal.setTimeZone(TimeZone.getTimeZone("America/New_York"))
        cal.setTimeInMillis(window.getStart)
        val hour = cal.get(Calendar.HOUR_OF_DAY)
        out.collect((terminal, records.size, hour))
      })
    windowedCountsByTerminal.print()

    val windowedBusiestTerminal = jfkRides
      .timeWindowAll(Time.hours(1))
      .apply((window, records, out: Collector[(Terminal, Int, Int)]) => {
        val cal: Calendar = Calendar.getInstance()
        cal.setTimeZone(TimeZone.getTimeZone("America/New_York"))
        cal.setTimeInMillis(window.getStart)
        val hour = cal.get(Calendar.HOUR_OF_DAY)

        val counts = records
          .foldLeft(HashMap.empty[Terminal, Int].withDefaultValue(0))((acc, terminal) => {
            acc(terminal) += 1
            acc
          })
        val busiestTerminal = counts.maxBy(_._2)
        out.collect((busiestTerminal._1, busiestTerminal._2, hour))
      })
    windowedBusiestTerminal.print()

    // execute program
    env.execute("Airport Terminals")
  }
}
```

TerminalUtils.scala

```scala
package lab3.problems

sealed trait Terminal{def grid: Int}
case object Terminal_1 extends Terminal {val grid = 71436}
case object Terminal_2 extends Terminal {val grid = 71688}
case object Terminal_3 extends Terminal {val grid = 71191}
case object Terminal_4 extends Terminal {val grid = 70945}
case object Terminal_5 extends Terminal {val grid = 70190}
case object Terminal_6 extends Terminal {val grid = 70686}
case object Terminal_404 extends Terminal {val grid = -1}

object TerminalUtils {
  val terminals: Set[Terminal] = Set(Terminal_1, Terminal_2, Terminal_3, Terminal_4, Terminal_5, Terminal_6)

  def gridToTerminal(gridCell: Int): Terminal = {
    terminals.find(t => t.grid == gridCell) match {
      case Some(terminal) => terminal;
      case None => Terminal_404;
    }
  }

}
```