ID2221 : Data-Intensive Computing

Lab 3 - Building Data Stream Applications with Flink

Note: **Imported necessary packages which are not included below to save space. We will use the timestamp and watermarks assigned in the TaxiRideSource however, have included our version of the timestamp and watermarks logic in the code below for reference. Have adapted the format to use code with brief explanation as we go along.**

TASK 1

```
public class AirportTrends {

    public enum JFKTerminal {
        TERMINAL_1(71436),
        TERMINAL_2(71688),
        TERMINAL_3(71191),
        TERMINAL_4(70945),
        TERMINAL_5(70190),
        TERMINAL_6(70686),
        NOT_A_TERMINAL(-1);

        int mapGrid;

        private JFKTerminal(int grid){
            this.mapGrid = grid;
        }

        public static JFKTerminal gridToTerminal(int grid){
            for(JFKTerminal terminal : values()){
                if(terminal.mapGrid == grid) return terminal;
            }
            return NOT_A_TERMINAL;
        }

        public static int getHour(long timestamp) {
            Calendar calendar = Calendar.getInstance();
            calendar.setTimeZone(TimeZone.getTimeZone("America/New_York"));
            calendar.setTimeInMillis(timestamp);
            return calendar.get(Calendar.HOUR_OF_DAY);
        }
    }

    public static void main(String[] args) throws Exception {


        final String input = "/Users/prashant/flink-java-project/nycTaxiRides.gz";

        final int maxEventDelay = 60;               // events are out of order by max 60 seconds
        final int servingSpeedFactor = 600;         // events of 10 minutes are served in 1 second

        // set up streaming execution environment
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // We will use here event time.
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        // initiate the data generator
        DataStream<TaxiRide> rides = env.addSource(
                new TaxiRideSource(input, maxEventDelay, servingSpeedFactor));
```

// We now build a pipeline to get the top terminals of the airport per hour.

```
SingleOutputStreamOperator<Tuple3<JFKTerminal, Integer, Integer>> filteredRides = rides
```

// Use Geoutils.mapToGridCell to locate the airport terminal in that grid

Here in this transformation we have to follow steps:

1. Use Geoutils.mapToGridCell to map coordinates to grid cells and on that we could apply the gridToTerminal transformation to map them to grid.

```
.map(new MapFunction<TaxiRide, Tuple2<JFKTerminal, DateTime>>() {
   @Override
   public Tuple2<JFKTerminal, DateTime> map(TaxiRide value) throws Exception {
      if (value.isStart) {
         return new Tuple2<>(
               JFKTerminal.gridToTerminal(GeoUtils.mapToGridCell(value.startLon, value.startLat)),
               value.startTime
         );
      } else {
         return new Tuple2<>(
               JFKTerminal.gridToTerminal(GeoUtils.mapToGridCell(value.endLon, value.endLat)),
               value.endTime
         );
      }
   }
})
```

2. Once we have the terminals, we filter out that terminals that are not in the airport.

```
.filter(new FilterFunction<Tuple2<JFKTerminal, DateTime>>() {
   @Override
   public boolean filter(Tuple2<JFKTerminal, DateTime> taxiRide) throws Exception {
      return (taxiRide.f0 != JFKTerminal.NOT_A_TERMINAL);
```

// value that is not a terminal using the NOT_A_TERMINAL declaration made in the class earlier.

```
   }
})
```

3. Now we have to figure out that time of that event at the airport using the function provided in the instructions document. We apply it after the filter transformation to save computations.

```
.map(new MapFunction<Tuple2<JFKTerminal, DateTime>, Tuple2<JFKTerminal, Integer>>() {
   @Override
   public Tuple2<JFKTerminal, Integer> map(Tuple2<JFKTerminal, DateTime> value) throws
Exception {
      Calendar calendar = Calendar.getInstance();
      calendar.setTimeZone(TimeZone.getTimeZone("America/New_York"));
      calendar.setTimeInMillis(value.f1.getMillis());
      return new Tuple2<>(value.f0, calendar.get(Calendar.HOUR_OF_DAY));
   }
```

```
})
//
```

4. Now, in the tuple, we key by the terminal value which is at 0 index for further operations.

```
.keyBy(0)
```

5. **The parameters of the problem stated to use a time window of an hour so we can use the TumblingEventTimeWindows method and fix it to provide a window of 1 hour. Since, we already have watermarks assigned in the TaxiRIdeSource which we already imported earlier, we don't have assign any more watermarks.**

```
.window(TumblingEventTimeWindows.of(Time.hours(1)))
```

6. Now, apply the count on the filtered stream of data so far.

```
.apply(new WindowFunction<Tuple2<JFKTerminal, Integer>, Tuple3<JFKTerminal, Integer, Integer>,
Tuple, TimeWindow>() {

    @Override
    public void apply(Tuple tuple, TimeWindow window, Iterable<Tuple2<JFKTerminal, Integer>> input,
Collector<Tuple3<JFKTerminal, Integer, Integer>> out) throws Exception {
        JFKTerminal terminal = tuple.getField(0); //terminal field
```

7. Now we get the starting timestamp of the window and convert into the hour format.This is the first timestamp that belongs to this window and then we count the visits made within this timewindow and print it.

```
        int rec_hour = JFKTerminal.getHour(window.getStart());
```

8. Counter of the visits

```
        int counter = 0;
        for (Iterator x = input.iterator(); x.hasNext(); x.next()) {
            counter++;
        }
```

9. Collect the tuple in the format of terminal, count and the hour of data collection.

```
        out.collect(new Tuple3<>(terminal, counter, rec_hour));
    }
})
```

10. **TimeStamp and Watermarks we didn't have to use as it was already assigned in the TaxiRideSource.**

Description:

Here we assign timestamps and watermarks. The Timestamp assigned by us will be just the timestamp of the taxi ride which includes its arrival and departure time event. We believe that the datastream might be not in

order therefore we will also use the AssignerwithPeriodicWatermarks and OutofOrderness with 1 minute

```
.assignTimestampsAndWatermarks(new AssignerWithPeriodicWatermarks<Tuple2<JFKTerminal, DateTime>>() {
  private final long maxOutOfOrderness = 60000;
  private long new_max_stamp;
@Override
  public Watermark getCurrentWatermark() {
    return new Watermark(new_max_stamp - maxOutOfOrderness);
  }
  @Override
  public long extractTimestamp(Tuple2<JFKTerminal, DateTime> element, long  previousElementTimestamp) {
    long timestamp = element.f1.getMillis();
    new_max_stamp = Math.max(timestamp, new_max_stamp);
    return timestamp;
  }
})
```

11. Output of the Task 1:

```
1> (TERMINAL_4,30,19)
2> (TERMINAL_3,39,20)
3> (TERMINAL_3,16,21)
4> (TERMINAL_3,3,22)
1> (TERMINAL_4,14,23)
2> (TERMINAL_4,45,0)
```

****
*

TASK 2

Aim: Using the result of task 1, we got unique terminals, their counters for every hour. Now, timeWindowAll together and select with maximum count the

1. **We again use TumblingEventTimeWindows for one hour window.**

```
.windowAll(TumblingEventTimeWindows.of(Time.hours(1)))
.apply(new AllWindowFunction<Tuple3<JFKTerminal, Integer, Integer>, Tuple3<JFKTerminal,
Integer, Integer>, TimeWindow>() {
    @Override
  public void apply(TimeWindow window, Iterable<Tuple3<JFKTerminal, Integer, Integer>>
values, Collector<Tuple3<JFKTerminal, Integer, Integer>> out) throws Exception {
```

2. **Now we get the first timestamp of the window and count visits that are made in this window.**

```
                int rec_hour = JFKTerminal.getHour(window.getStart());
```

3. Do the maximum counter of visits.

```
                int max_count = 0;
                JFKTerminal popular_terminal = JFKTerminal.NOT_A_TERMINAL;
```

4.swap values for sorting in order of maximum visits.

```
                for(Tuple3<JFKTerminal, Integer, Integer> value : values) {
                   if (max_count < value.f1) {
                      max_count = value.f1;
                      popular_terminal = value.f0;
                   }
                }
```

5. **Single most popular terminal per hour along with the number of ideas and the time of the day..**

```
                out.collect(new Tuple3<>(popular_terminal, highestmax_countCount, rec_hour));
             }
          });
```

6. Print the filtered stream of data.

```
      filteredRides.print();

      env.execute();

   }
}
```

6. Output:

```
 1>       (TERMINAL_3,3,22)
 1>       (TERMINAL_4,14,23)
 2>       (TERMINAL_4,45,0)
 3>       (TERMINAL_4,70,1)
 4>       (TERMINAL_3,48,2)
 1>       (TERMINAL_4,42,3)
```

********** END **********