DS 503 - Project 2

By :
Abdulaziz Alajaji - asalajaji@wpi.edu
Yousef Fadila - yousef@fadila.net

Question1 - Step1)

The main class to create the Datasets is DataGeneratorMain.java.
The code is very short and straightforward

```
public class DataGeneratorMain {

    public static void main(String[] args) {

        DataGenerator[] dataGenerators = new DataGenerator[]{ new PSetGenerator(15000000), new RSetGenerator(15000000)};

        for (DataGenerator dataGenerator : dataGenerators) {
            dataGenerator.generate();
        }
    }
}
```

Both PSetGenerator and RSetGenerator classes extend DataGenerator abstract class which has abstract method named: “generate()”.
The Generator constructor receive number of records as an argument in order to allow creating small samples.

When generating the rectangles , we first generate the width, height so we could take them into account when generating x,y to prevent rectangles to go over boundaries

```
StringBuilder sb = new StringBuilder();
int width = randomInt(MIN, MAX_WIDTH);
int height = randomInt(MIN, MAX_HEIGHT);
// choose the top-left so that the whole rectangle is in the boundary of 0,max
int x1= randomInt(MIN, MAX-width);
int y1 = randomInt(MIN + height ,MAX);
```

Question1 - Step2)

Run commands:
hadoop jar out/artifacts/project2.jar edu.wpi.ds503.SpatialJoin /user/hadoop/p_dataset /user/hadoop/r_dataset /user/hadoop/spatialjoin1

Or
hadoop jar out/artifacts/project2.jar edu.wpi.ds503.q1.SpatialJoin /user/hadoop/p_dataset /user/hadoop/r_dataset /user/hadoop/spatialjoin1 **"W(9900,1700,9999,1800)"**

"W(9900,1700,9999,1800)" is optional as required

We divide the whole space into grids, each grid width.height is the max allowed width,height for rectangle respectively.

The points mapper is simply output the point with its grid key.

```
    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        String[] fields = value.toString().split(",");
        int x=Integer.parseInt(fields[0]);
        int y=Integer.parseInt(fields[1]);
        SpatialJoinData point = new SpatialJoinData().setPoint(x, y);
        if (window != null && !window.includePoint(x,y)) {
            return;
        }
        context.write(new Grid().setGridByPoint(x, y), point);
    }
}
```

The function setGridByPoint is implemented as:

```
public Grid setGridByPoint(int x, int y)
{
   int gridX = x/GRID_WIDTH;
   int gridY = y/GRID_HEIGHT;

   this.x.set(gridX);
   this.y.set(gridY);
   return this;
}
```

That means each grid is identified with its **bottom-left corner.**

In the case of rectangles, each rectangles could maximum overlap 4 grids (because we choose the grid width,height to be the maximum values. In the mapper we will associate each rectangle to all its overlapped grids, up to 4;

```
context.write(new Grid().setGridByPoint(x,y), rectangle);

if (x / Grid.GRID_WIDTH != (x + width) / Grid.GRID_WIDTH)
   context.write(new Grid().setGridByPoint(x + width, y), rectangle);
```

```
if (y / Grid.GRID_HEIGHT != (y - height) / Grid.GRID_HEIGHT)
   context.write(new Grid().setGridByPoint(x, y - height), rectangle);

if ((y / Grid.GRID_HEIGHT != (y - height) / Grid.GRID_HEIGHT) && (x / Grid.GRID_WIDTH
!= (x + width) / Grid.GRID_WIDTH))
   context.write(new Grid().setGridByPoint(x + width, y - height), rectangle);
```

In the reducer, all rectangle and points that fall in the same grid with arrive together with the same key. Therefore all what we need to do is go over all rectangles, for each one, check all points to find which one is contained in it.

```
for (SpatialJoinData val : values) {
    if (val.isRectangle()) {
        rects.add(new SpatialJoinData(val));
    } else {
        points.add(new SpatialJoinData(val));
    }
}

for (int i=0; i< rects.size(); i++) {
    for (int j=0; j< points.size(); j++){
        if (rects.get(i).includePoint(points.get(j).x.get(), points.get(j).y.get()))
            context.write(NullWritable.get(), new Text("<" + rects.get(i).id.toStrinc
        }
    }
}
```

To efficiently support the option input parameter **"W(9900,1700,9999,1800)"** both the points mapper and rectangles mapper will filter out any point, rectangles doesn't intersect with the window.

For a point, the check is very simple. If W contain the point, it will be sent to reducer otherwise ignored.

For a rectangle; the check is if W overlap the rectangle or not

```
if (window != null && !window.overlapRect(rectangle)) {
    return;
}
```

Overlapping is false if one rectangle is on left side of other or if one rectangle is above other

```java
public boolean overlapRect(SpatialJoinData rect) {

    // If one rectangle is on left side of other
    if (this.x.get() > rect.x.get() + rect.width.get() || rect.x.get() > this.x.get() + this.width.get())
        return false;

    // If one rectangle is above other
    if (this.y.get() < rect.y.get() - rect.height.get() || rect.y.get() < this.y.get() - this.height.get())
        return false;

    return true;
}
```

Question 2)

Step 1)

This is a straightforward step, which is uploading the dataset into HDFS.

Step 2)

In order to create a custom file input format, we had to create two new classes (“JSONFileInputFormat” and (“JSONLineRecordReader”), by extending Hadoop’s FileInputFormat and LineRecordReader. The JSONFileInputFormat is only an interface to the mapper, and all the logic is coded into JSONLineRecordReader.

The idea is basically instead of mapping each line as one record, we could determine how exactly each record is created from the file and passed as (key, value) to the mapper.

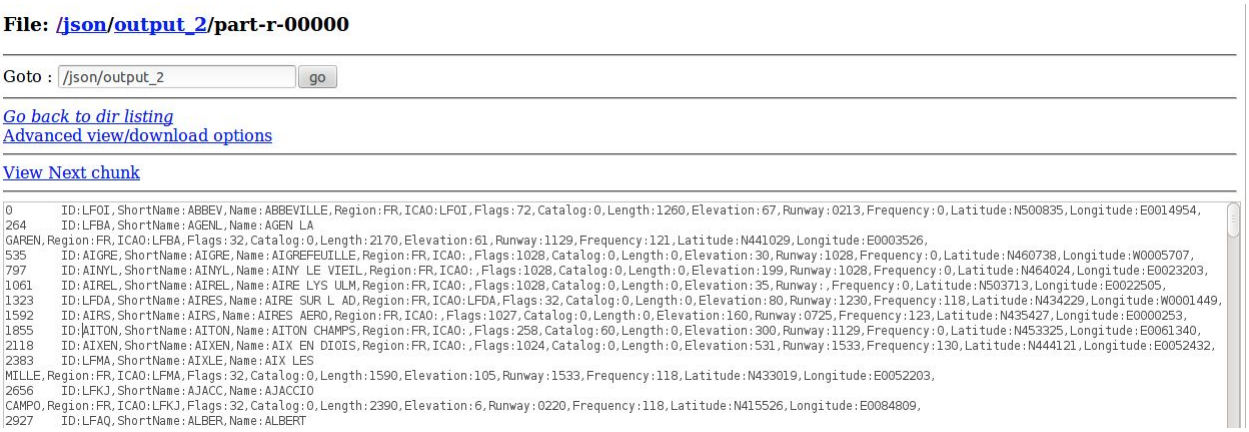
We have here a file that has many records, written in JSON format. For each record, it starts with a line that has an opening bracket (“{“) and ends with a line that has an enclosing bracket (“}). Our method for constructing each record is by reading multiple lines until a stopping criteria (reaching a line that contains “}” , or exceeding 30 lines). Even though all records has fixed size of 15 lines, we wrote that code this way for better generalization.

Also, the code has the ability to detect if the record spans multiple splits or not by checking the current position of the record in the split and and the split size. For example, if the record position is one line before the split ends, it will keep reading from the next split until it reaches the stopping criteria (finding the enclosing bracket).

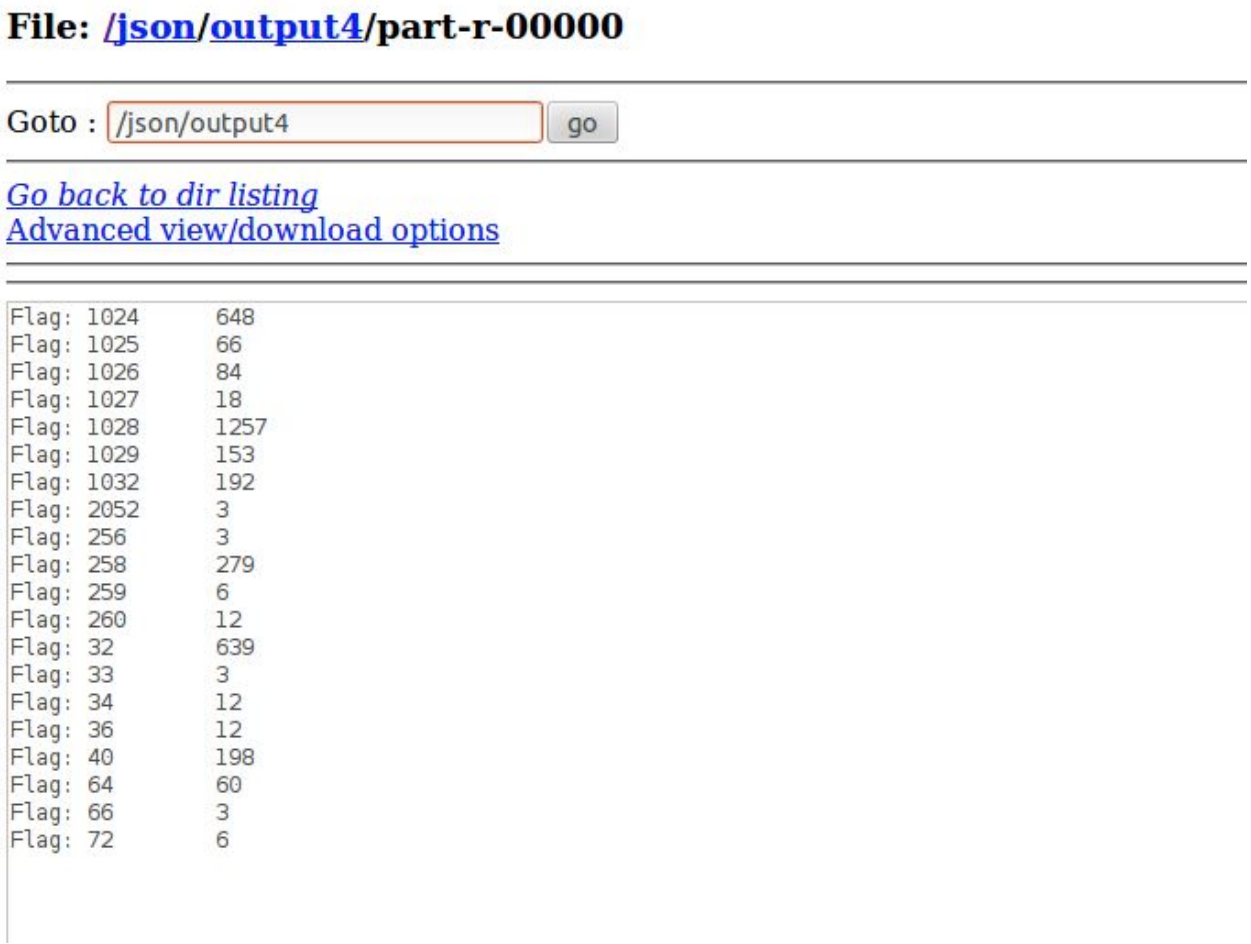
For converting the multiple lines and constructing a comma-separated string: We have used a regular expression to match JSON key/value pairs and combine all the results in one line:

```
Pattern pattern = Pattern.compile("\"(\\w+)\"\\s?:\\s?\"?([\\w\\d\\s]+)\"?,?");
```

In the below figure, it shows how mapper receives the (key, value) from from the splits using the custom file input format. It receives each record in a separate (converted) line.



And here is the result of the required task (Counts records for each Flag value)



Question 3)

Step 1)

Creating the dataset:

We generate two files:

Dataset: The file that contains random points (x,y), and we scaled it to ~100MB (98MB) with 10000000 points .

Then we upload the data to hdfs.

Random Seed file:

This file is used as an input when running K-Means clustering algorithm. We selected a file of 1000 points.

Later in the execution, the program will take this file as an input and it will take Randomly K-points from the random seed file.

Step 2)

In order to make of use of the points dataset in a better way, we created custom file input format called (PointWritable) which reads two numbers stored the file.
E.g. “123 433” -> x=123, y = 433.

Some highlight and description of the workflow:

- We start the program by initialization parameters (set up unique working directory) e.g. kmeans_runs/TIMESTAMP.
- This way we make sure each run will have separate folder to access them easily later.
- In the first iteration “Zero” , the program will read the input path for KSEED_FILE, and K.
- It will also print them just for information at the beginning.
- Then, it will store the path using the shared configuration context (which the mapper can access).
    - **conf.set ("cluster_input", path_clusters_input.toUri().getPath());**
- Mappers can read that path in their (setup) method, so it can initialize an array of k centroids and use them in the map method.
- In order to map each point to the right K centroid:
  We calculate the euclidean distance between the point and each k centroid, and the point goes to the centroid which has the smallest distance.

    - **int distance = (int) (Math.pow(x-centroid_x,2)+ Math.pow(y-centroid_y,2));**

- Mappers produces (PointWritable, List of (PointWritable)), the key here is basically one of the K inputs.
- For the Combiner phase: It will receive a K centroid, and a list of points assigned to that centroid from the current node mappers,
  It will then calculate the partial average for both x and y coordinates based on the points value,
- For the reducers phase: It will receive a K centroid, and a list of partial averages (the ones calculated in the combiner) assigned to that centroid,
  It will then calculate the new centroid based on the points value, and store that as a text file.
- For iterations > 0:
  It will start another map-reduce job, the same way except that instead of using the random seed file as an input for k centroids, It will take centroids inputs from the last produced output (the previous job).
- Because we used the a unique working directory, and used a naming conventions for the input files (clusters_input_0 , clusters_input_1) which is (clusters_input_# of iteration), it made it easier for us to pass inputs for the next iteration.
- For checking whither centroids has converged or not:
    - We have a used a counter inside the reducer. It will check inside cleanup() method, by reading the previous output file and match it with current calculated centroids, if they match it will increment the counter for that reducer.

```
if (count == matches) {
        context.write( NullWritable.get(), new Text("---NOTHING HAS CHANGED----"));
        context.getCounter("STATUS", "unchanged").increment(1);


  }
```

    - The job tracker, will monitor and check -when the job is completed- for the counter inside the reducer. If it sees results have not changed for two consecutive iterations, it will produce a message and stop the iteration.

```
if (counter_value != null){
                counter_global+=counter_value;
                if (counter_global > 1) {
                        LOG.info("NOTHING HAS CHANGED > 2 times... ");
                        LOG.info("Terminating Iteration...");
                        break;
                }
        }
```