PRÁCTICA HADOOP (BIG DATA)

Álvaro Pérez García

MASTER EN INGENIERÍA INFORMÁTICA Universidad de Huelva

Diseño del sistema

Nuestro sistema toma como patrón de diseño un sistema equivalente al redactado en la teoría sobre Hadoop. De esta manera podemos dividir nuestro sistema en 3 partes principales bien diferenciadas:

Driver:

Es la parte encargada de especificar los tipos de datos con los que se va a trabajar, tanto de entrada como de salida, además de lanzar los parámetros necesarios de la aplicación (job) y lanzar la misma al entorno para ser ejecutada.

Mapper:

Es la parte encargada de procesar cada línea de nuestro conjunto de datos, una vez procesada, se encarga de extraer la información realmente necesaria y proporcionar una salida conjunta con los datos extraidos.

Reducer:

Es la parte encargada de aglutinar la información recibida desde el Mapper y realizar la lógica necesaria con los datos agrupados que se reciben en la entrada. Su objetivo es proporcionar una salida con los datos presentados de manera conveniente.

Para representar este sistema se ha realizado un proyecto Java utilizando Hadoop, compuesto por tres clases, las cuales serán descritas a continuación:

Clase AverageRadiation:

```
public class AverageRadiation {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(AverageRadiation.class);
    job.setJobName("Average radiation");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(AverageRadiationMapper.class);
    job.setReducerClass(AverageRadiationReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

En esta clase, como podemos comprobar se definen los parámetros de entrada y salida de nuestra aplicación (la ruta del fichero de entrada y el directorio donde creara los archivos con los resultados).

Además, define cuáles serán las clases encargadas de implementar los métodos Map y Reduce para llevar a cabo la lógica necesaria con nuestros datos.

Clase AverageRadiationMapper

```
public class AverageRadiationMapper
  extends Mapper<LongWritable, Text, Text, IntWritable> {

  private static final String[] estaciones_permitidas = {"2","3","4","5","6","7","8","9","10"};

  private boolean isCorrect(String radiacion) {
      return radiacion.matches("[-+]?\\d*\\.?\\d+");
  }

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    String line = value.toString();

    String[] linea = line.split(";",-1);

    String id_estacion = linea[2].replace("\"", "");
    String nombre_estacion = linea[3].replace("\"", "");

    if(Arrays.asList(estaciones_permitidas).contains(id_estacion)) {
        String radiacion = linea[16];

        if(isCorrect(radiacion)) {
            int radiacion_entero = Integer.parseInt(radiacion);
            context.write(new Text(nombre_estacion), new IntWritable(radiacion_entero));
        }
    }
  }
}
```

En esta clase nos encargamos del procesado línea a línea del fichero .csv de entrada con todos los datos meteorológicos.

Como podemos comprobar en primer lugar, iteramos línea por línea, obteniendo un string de ella, tras esto, hacemos un “Split” (Troceamos el string en varios substrings con este método), de modo que conseguimos un array de strings con cada parte de este separada por un “;”.

Una vez troceado comprobamos si el ID de la estación de esa línea esta dentro de las estaciones permitidas, es decir, las estaciones para las que buscamos datos, en caso afirmativo, comprobamos seguidamente si la radiación correspondiente es un numero correcto (Comprobamos con una expresión regular si tiene formato de numero decimal, para evitar valores incorrectos).

En caso de que todo sea correcto grabamos en la salida los valores del nombre de la estación y su correspondiente radiación para agruparlos todos por cada estación.

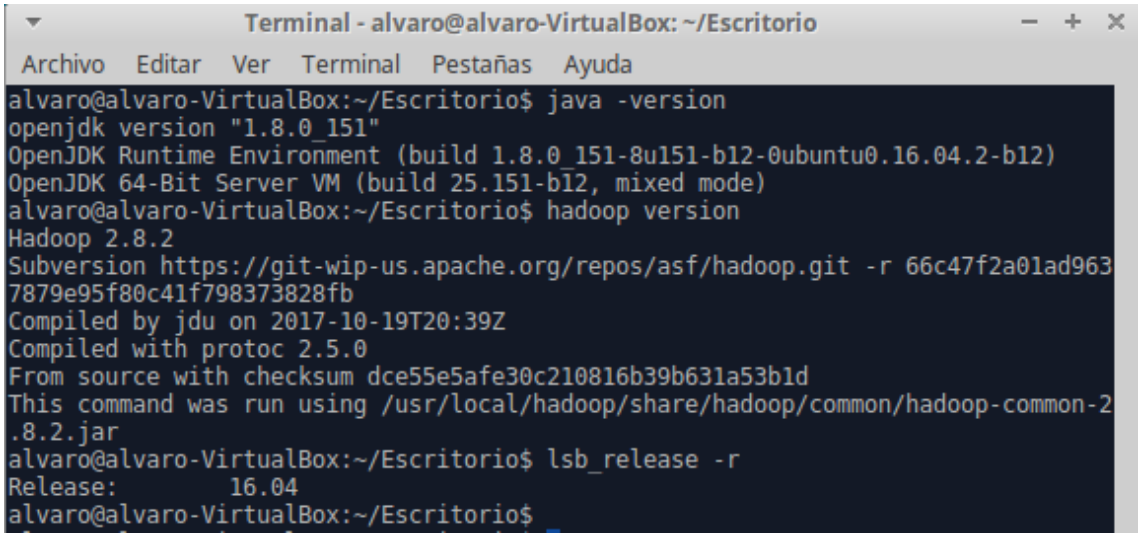
Clase AverageRadiationReducer

```
public class AverageRadiationReducer
  extends Reducer<Text, IntWritable, Text, Text> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
      Context context)
      throws IOException, InterruptedException {

    int average_sum = 0;
    int counter = 0;

    for (IntWritable value : values) {
        average_sum += value.get();
        counter++;
    }

    float average_radiation = (float)average_sum/counter;

    context.write(key, new Text(String.format("%.02f", average_radiation)));
  }
}
```

Esta clase se encarga de, una vez recibidos los datos agrupados por estación, hacer la media de todos los valores recibidos de radiación para esa clave (Estación), proporcionando así una salida con el nombre de cada una de ellas y la correspondiente media (formateada con 2 decimales).

Experimentación

Para probar nuestro sistema hemos hecho pruebas prácticas para comprobar el correcto funcionamiento, dichas pruebas han sido llevadas a cabo en el siguiente entorno:



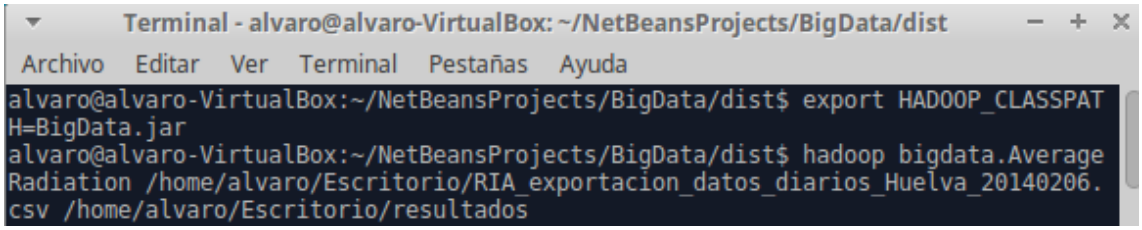
Versión de Java: ***1.8.0_151***

Versión de Hadoop: ***2.8.2***

Versión de Ubuntu: ***Xubuntu release 16.04***

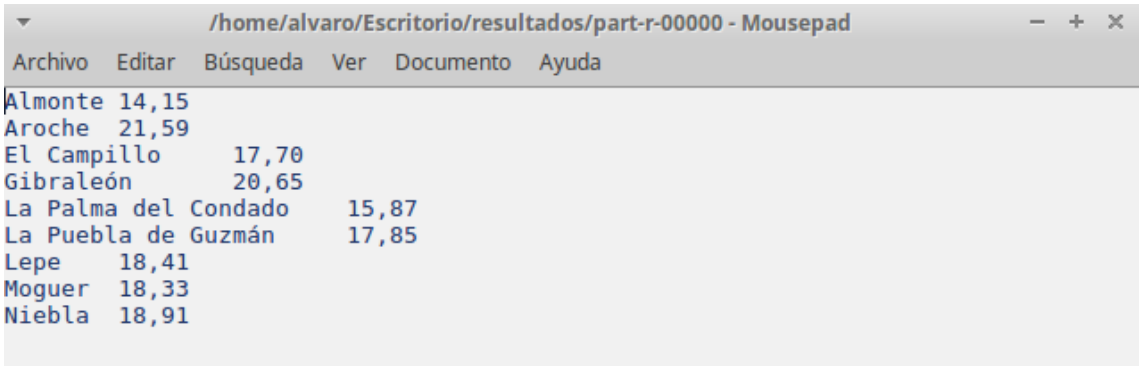
Para realizar las pruebas se necesitaba compilar el proyecto en un archivo .jar, el cual sería utilizado luego por Hadoop para realizar las correspondientes tareas.

Por tanto, una vez compilado y generado el archivo .jar (BigData.jar), solo teníamos que movernos al directorio donde se generaba y ejecutar los siguientes comandos:



En ellos le definimos el archivo .jar que necesita utilizar Hadoop, la clase que se encarga de lanzar el job, el archivo de entrada de datos, y el directorio de salida.

Finalmente el comando nos arrojaba los resultados en el correspondiente directorio de salida:



En el podíamos comprobar la media de radiación de cada una de las estaciones para las cuales habíamos realizado el estudio.