UNIVERSITY OF SALERNO

DEPARTMENT OF INFORMATION ENGINEERING AND ELECTRICAL AND APPLIED MATHEMATICS

Master’s Degree in Computer Engineering

Big Data Project report

Course: High Performance Computing

Lecturer: Giuseppe D'Aniello gidaniello@unisa.it

Alberti Andrea 0622702370 a.alberti2@studenti.unisa.it

ACADEMIC YEAR 2023/2024

TABLE OF CONTENT

Introduction 4
- 1.1 Dataset Analysis 4
- 1.2 Dataset Changes 4

Hadoop Map-Reduce Exercise 5
- 2.1 PROBLEM'S DESCRIPTION 5
- 2.2 How To Run 5
- 2.3 PROPOSED SOLUTION 6
  - 2.3.1 Driver 6
  - 2.3.2 Mapper 7
  - 2.3.3 Reducer 7
  - 2.3.4 ApartmentType 8
- 2.4 RESULTS AND FINAL CONSIDERATIONS 9

Apache Spark Exercise 10
- 3.1 PROBLEM'S DESCRIPTION 10
- 3.2 PROPOSED SOLUTION 10
- 3.3 RESULTS 12

LIST OF FIGURES

Figure 1: Driver class 6
Figure 2: Mapper class 7
Figure 3: Reducer Class 7
Figure 4: ApartmentType class 1 8
Figure 5: ApartmentType class 2 8
Figure 6: Results 9
Figure 7: Preliminary instructions 10
Figure 8: Data cleansing 10
Figure 9: Final Trasformations 11
Figure 10: Results 12

Introduction

1.1 Dataset Analysis

The dataset contains data of rooms and apartments for rent on Airbnb in New York City. The dataset, called "airbnb2.csv", contains approximately 5000 lines including the first header line.

Dataset fields are:

- *Id* - A unique identifier for each Airbnb listing.
- *Name* - The name of the Airbnb listing.
- *Host_id* - A unique identifier for the Airbnb host.
- *Host_name* - The name of the Airbnb host.
- *Neighbourhood_group* - The borough of the Airbnb listing.
- *Neighbourhood* - The neighbourhood of the Airbnb listing.
- *Latitude* - The latitude of the Airbnb listing.
- *Longitude* - The longitude of the Airbnb listing.
- *Room_type* - The type of room available for rent (e.g. private room, entire home/apt, shared room).
- *Price* - The nightly room price.
- Minimum_nights
- *Number_of_reviews*
- *Last_review*: the date of the last review
- *Reviews_per_month*
- *Calculated_host_listings_count*:
- *Avaiability_365*: the annual availability of the B&B

1.2 Dataset Changes

To satisfy the exercises, a small modification was made to the initial dataset. Some lines of the dataset have the '\n' character (new return) in the "Name" field, not allowing a correct reading of the dataset by rows. These lines (8 lines in particular) were then modified by eliminating the \n character.

Hadoop Map-Reduce Exercise

2.1 PROBLEM'S DESCRIPTION

The exercise chosen for hadoop map-reduce concerns finding the average prices of B&Bs with a number greater than 10 reviews, grouped by neighborhood and room_type.

2.2 How To Run

(ON MASTER BASH)
hdfs namenode -format (format file system, run on first startup)
$HADOOP_HOME/sbin/start-dfs.sh (start hdfs)
$HADOOP_HOME/sbin/start-yarn.sh (start yarn)

START APPLICATION
cd /data/MapReduce (go in the folder that contains the jar file and the input folder)
hdfs dfs -put Input/airbnb2.csv hdfs:///input (load on hdfs the input file)
hadoop jar Project_mapreduce.jar /input /output (execute jar program)

CHECK OUTPUT
hdfs dfs -cat /output/part-r-00000 (print the output)
hdfs dfs -get /output output (load the output file locally)

CLEAN FILES
hdfs dfs -rm -r hdfs:///input (remove input file)
hdfs dfs -rm -r hdfs:///output (remove output file)

STOP CLUSTER
$HADOOP_HOME/sbin/stop-dfs.sh (stop hdfs)
$HADOOP_HOME/sbin/stop-yarn.sh (stop yarn)

2.3 PROPOSED SOLUTION

The proposed solution involves the implementation of 4 classes: Driver, Mapper, Reducer and ApartmentType. Pattern Numerical Summarization has been used.

2.3.1 Driver

This Java class, DriverBigData, serves as the driver program for a Hadoop MapReduce job (Figure 1):

- configure the input and output folder;
- creates a new Hadoop configuration;
- sets a custom key-value separator for the input format to a comma (,);
- initializes a new MapReduce job named "Project_BigData_MapReduce";
- sets the input format class to KeyValueTextInputFormat and the output format class to TextOutputFormat;
- specifies the main class (DriverBigData) Mapper and Reducer classes for the job;
- specifies the output key and value classes for the mapper and the reducer;
- specifies the number of reducers(1);

```java
public static void main(String[] args) throws Exception {
    Path inputPath; //input file
    Path outputPath; //output file
    // Parse the parameters
    inputPath = new Path(args[0]);
    outputPath = new Path(args[1]);

    Configuration conf = new Configuration();
    //set new separator forKeyValueTextInputFormat
    conf.set(name: "key.value.separator.in.input.line", value: ",");
    // Define a new job
    Job job = Job.getInstance(conf, jobName: "Project_BigData_MapReduce");
    /// Set path of the input file/folder (if it is a folder, the job reads all the files in the specified folder) for this job
    FileInputFormat.addInputPath(job, path: inputPath);
    // Set path of the output folder for this job
    FileOutputFormat.setOutputPath(job, outputDir: outputPath);
    // Specify the class of the Driver for this job
    job.setJarByClass(cls: DriverBigData.class);
    // Set job input format
    job.setInputFormatClass(cls: KeyValueTextInputFormat.class);
    // Set job output format
    job.setOutputFormatClass(cls: TextOutputFormat.class);
    // Set map class
    job.setMapperClass(cls: MapperBigData.class);
    // Set map output key and value classes
    job.setMapOutputKeyClass(theClass: ApartmentType.class);
    job.setMapOutputValueClass(theClass: FloatWritable.class);
    // Set reduce class
    job.setReducerClass(cls: ReducerBigData.class);
    // Set reduce output key and value classes
    job.setOutputKeyClass(theClass: ApartmentType.class);
    job.setOutputValueClass(theClass: Text.class);
    // Set number of reducers
    job.setNumReduceTasks(tasks: 1);
    // Execute the job and wait for completion
    System.exit(job.waitForCompletion(verbose: true) ? 0 : 1);
```

Figure 1: Driver class

2.3.2 Mapper

The Mapper class receives as input a key, value pair. Then it filters the data, not processing either the header and the apartments with the number of reviews lower than the minimum required. Finally, it writes the key value pair (apartmentType, price) to the context. It also handles the case where the "Name" field contains commas, which would cause problems in rows splitting. In the dataset, when the name field contains commas it is delimited by quotation marks. In this case, the contents of the name field are deleted, being useless for the following analysis (Figure 2).

```java
public class MapperBigData extends Mapper<Text, Text, ApartmentType, FloatWritable>{
   // Minimum number of reviews required  for the b&b
   private final static Integer MinReview = Integer.valueOf(i: 9);

  @Override
  public void map(Text key, Text value, Context context) throws IOException, InterruptedException{
    //Remove the header and handles the case where commas appear in the "name" field
    if(!key.toString().contains(s:"id")){
      String parts[] = value.toString().split(regex:"\"");
      StringBuilder line = new StringBuilder(parts[0]);
      for (int i = 1; i < parts.length; i += 2) {
        line.append(str: "\"\"").append(parts[i + 1]);
      }

      String fields[] = line.toString().split(regex:",");
       if(fields.length>=10){
          if(!(fields[4].isEmpty() || fields[7].isEmpty() || fields[10].isEmpty())){
            Integer reviewCount=Integer.valueOf(fields[10]);

            // Compare the value of reviewCount with the Minimum number of reviews required
            if (reviewCount.compareTo(anotherInteger: MinReview)>0){
              context.write(new ApartmentType(fields[4],fields[7]),new FloatWritable (value:Float.parseFloat(fields[8])));
```

Figure 2: Mapper class

2.3.3 Reducer

The ReducerBigData class extends Reducer and implements the reduce method. This method receives as input a key of type ApartmentType, and calculates the average by iterating over the list of price average values. At the end, results are written in the context (Figure 3).

```java
public class ReducerBigData extends Reducer<ApartmentType,FloatWritable,ApartmentType,Text> {

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

  float sum = (float) 0.0;
  int n = 0;
  for (FloatWritable val : values) {
   sum += val.get();
   n += 1;
  }
  float average = sum/n;
  context.write(keyout: key, new Text("(" + average + ", " + n + ")"));
```

Figure 3: Reducer Class

2.3.4 ApartmentType

The ApartmentType class is used as a key by the mapper, so it must implement the WritableComparable interface (Figure 4). Class maintains neighborhood and room type as attributes, and it implements the write, readFields, compareTo, equals, hashCode, toString methods as well as the constructor and getters (Figure 5).

```java
public class ApartmentType implements WritableComparable<ApartmentType> {
   private final Text neighborhood;
   private final Text roomType;

   public ApartmentType() {
      this.neighborhood = new Text();
      this.roomType = new Text();
   }

   public ApartmentType(String neighborhood, String roomType) {
      this.neighborhood = new Text(string:  neighborhood);
      this.roomType = new Text(string:  roomType);
   }
```

Figure 4: ApartmentType class 1

```java
@Override
public void write(DataOutput out) throws IOException {
   neighborhood.write(out);
   roomType.write(out);
}
@Override
public void readFields(DataInput in) throws IOException {
   neighborhood.readFields(in);
   roomType.readFields(in);
}
//compare the neighborhood, if equal, compare the room_type
@Override
public int compareTo(ApartmentType other) {
   int neighborhoodComparison = neighborhood.toString().compareToIgnoreCase(str: other.neighborhood.toString());
   if (neighborhoodComparison != 0) {
      return neighborhoodComparison;
   }
   return roomType.toString().compareToIgnoreCase(str: other.roomType.toString());
}
//two ApartmentType object are equals if the neighborhood and the room_type are both equals ignoring case
@Override
public boolean equals(Object o) {
   if (this == o)
      return true;
   if (o == null || getClass() != o.getClass())
      return false;
   ApartmentType a = (ApartmentType) o;
   return (a.getNeighborhood().toString().equalsIgnoreCase(anotherString:neighborhood.toString()) && a.getRoomType().toString().equalsIgnoreCase(anotherString:roomType.toString()));
}
//Note that hashCode() is frequently used in Hadoop to partition keys. It's important that your implementation of hashCode() returns the same result across different instances of the JVM.
//Note also that the default hashCode() implementation in Object does not satisfy this property.
@Override
public int hashCode() {
   final int prime = 31;
    int result = 1;
    result = prime * result + neighborhood.hashCode();
    result = prime * result + roomType.hashCode();
    return result;
```

Figure 5: ApartmentType class 2

2.4 RESULTS AND FINAL CONSIDERATIONS

Performed exercise allowed to underline the following considerations:

- combiner has not been used, because the averaging operation is not associative and commutative;
- use of configurations with multiple reducers (2,4) have been discarded due to a longer execution time, caused by communication overhead. Only one reducer was used, which proved to be the best choice for the input dataset.

Results are contained in the Output folder for all the number of reducer's configurations. An example is shown in Figure 6.

- ❖ (neighborhood, Room_type) (average price, count apartments)

```
(Allerton, Entire home/apt)     (104.0, 1)
(Arverne, Entire home/apt)      (383.33334, 3)
(Arverne, Private room) (82.0, 2)
(Astoria, Entire home/apt)      (131.78947, 19)
(Astoria, Private room) (63.863636, 22)
(Astoria, Shared room)  (500.0, 1)
(Battery Park City, Entire home/apt)    (245.0, 1)
(Bay Ridge, Private room)       (35.0, 1)
(Baychester, Entire home/apt)   (95.0, 1)
(Bedford-Stuyvesant, Entire home/apt)   (161.74074, 81)
(Bedford-Stuyvesant, Private room)      (62.825397, 63)
(Bedford-Stuyvesant, Shared room)       (40.0, 1)
(Belle Harbor, Private room)    (225.0, 1)
(Bensonhurst, Entire home/apt)  (121.0, 1)
(Bensonhurst, Private room)     (68.0, 1)
(Bensonhurst, Shared room)      (79.0, 1)
(Bergen Beach, Entire home/apt) (235.0, 1)
(Boerum Hill, Entire home/apt)  (123.0, 1)
(Boerum Hill, Private room)     (71.0, 1)
(Borough Park, Entire home/apt) (147.66667, 3)
(Brighton Beach, Entire home/apt)       (99.0, 1)
(Bronxdale, Private room)       (40.0, 1)
(Brooklyn Heights, Entire home/apt)     (131.5, 2)
(Brownsville, Private room)     (71.0, 3)
(Bushwick, Entire home/apt)     (129.0, 29)
(Bushwick, Private room)        (62.365852, 41)
(Bushwick, Shared room) (75.0, 1)
(Cambria Heights, Entire home/apt)      (79.0, 1)
(Canarsie, Entire home/apt)     (112.166664, 6)
(Carroll Gardens, Entire home/apt)      (169.4, 5)
(Carroll Gardens, Private room) (100.0, 1)
(Castleton Corners, Entire home/apt)    (299.0, 1)
(Chelsea, Entire home/apt)      (304.1613, 31)
(Chelsea, Private room) (101.833336, 12)
(Chinatown, Entire home/apt)    (165.66667, 9)
(Chinatown, Private room)       (103.0, 5)
(Chinatown, Shared room)        (70.0, 1)
(City Island, Entire home/apt)  (95.0, 1)
(Civic Center, Entire home/apt) (200.0, 1)
(Civic Center, Private room)    (89.333336, 3)
(Claremont Village, Entire home/apt)    (83.5, 2)
(Clifton, Entire home/apt)      (120.0, 3)
(Clinton Hill, Entire home/apt) (167.2, 15)
(Clinton Hill, Private room)    (89.6, 5)
(Cobble Hill, Entire home/apt)  (141.8, 5)
(Cobble Hill, Private room)     (82.0, 2)
```

Figure 6: Results

Apache Spark Exercise

3.1 PROBLEM'S DESCRIPTION

The problem concerns identifying the top 3 hosts by number of B&B, grouped by their neighborhoods.

3.2 PROPOSED SOLUTION

As a first step, input and output folders are set (passed as arguments) and the configuration and context objects are created (Figure 7).

Following, an initial RDD, in which data is read from the input file as strings, is created. Each element of the initial RDD corresponds to a line of the input file. Transformations are then performed on the initial RDD, with the aim of cleaning the data. These transformations modify the RDD, eliminating the header row and any duplicate rows, and handling the case where the "Name" field contains commas, which would cause problems to rows splitting. In the dataset, when the name field contains commas, it is delimited by quotation marks. In this case, the contents of the name field are deleted, being useless for the following analysis (Figure 8).

```
public static void main(String[] args) {
    //input and output path
    String inputPath = args[0];
    String outputPath = args[1];

    // Create a configuration object and set the name of the application
    SparkConf conf = new SparkConf().setAppName(name: "Project-Big_Data-Spark");

    // Create a Spark Context object
    JavaSparkContext sc = new JavaSparkContext(conf);
```

Figure 7: Preliminary instructions

```
// Build an RDD of Strings from the input textual file
// Each element of the RDD is a line of the input file
JavaRDD<String> readingsRDD = sc.textFile(path: inputPath);
String header = readingsRDD.first();
//Duplicate elimination
JavaRDD<String> readingsNoDuplicateRDD = readingsRDD.distinct();

//Remove the content of the second field (name), enclosed within quotation marks, as it may contain commas that could pose issues during analyses.
JavaRDD<String> readingsNoDuplicateNewRDD = readingsNoDuplicateRDD.map(line ->{
    String parts[] = line.split(regex:"\"");
    StringBuilder nuovaLinea = new StringBuilder(parts[0]);
    for (int i = 1; i < parts.length; i += 2) {
        nuovaLinea.append(str: "\"\"").append(parts[i + 1]);
    }
    return nuovaLinea.toString();
});
// Removal of header row and potentially error-causing rows
JavaRDD<String> dataWithoutHeader = readingsNoDuplicateNewRDD.filter(line -> {
    if (!header.equals(anObject: line)){
        String[] fields = line.split(regex:",");
        if(fields.length>=6)
            if(!(fields[2].isEmpty() || fields[5].isEmpty()))
                return true;}
    return false;
```

Figure 8: Data cleansing

Subsequently, several transformations are carried out (Figure 9).

- the RDD is mapped into a pairRDD with key value pair (neighborhood,hostID);
- the pairRDD is mapped into another pairRDD with key pair value ((neighborhood,hostID)count), with the count set to 1, which indicates the number of apartments;
- a reduceByKey is then performed to sum all the host counts grouped by key (neighborhood, hostID). This way you can count all the hosts' apartments for each neighborhood;
- a mapToPair is then performed to change the key, setting the neighborhood as the key and the pair (hostID, countApartments) as the value;
- RDD is subsequently grouped by key, to perform a top 3 on the count value, for each neighborhood;
- a mapValues is used where a local LinkedList is used to sort the list of pairs (hostId, countApartments). A linkedList was chosen for use of the removeLast method, used after sorting the list by count. The list is of size <=3, so it turned out to be a better solution than importing the entire list locally, as it may have been too large in size;
- at the end, elements are sorted by their keys.

```
// Create a javaPairRDD mapping each row to a Tuple2<Neighborhood, hostID>
JavaPairRDD<String, String> hostPerNeighborhood = dataWithoutHeader.mapToPair(line -> {
        String[] fields = line.split(regex:",");
        return new Tuple2<>(fields[5],fields[2]);
    });
//  Maps in Tuple2<<Neighborhood, hostID>, 1>
JavaPairRDD<Tuple2<String, String>, Integer> hostPerNeighborhoodCount = hostPerNeighborhood.mapToPair(
    neighborhoodHost -> new Tuple2<>(_1: neighborhoodHost, _2: 1));

//  Reduces by key (Tuple2<Neighborhood, hostID>), sum the value for every host per neighborhood
JavaPairRDD<Tuple2<String, String>, Integer> hostPerNeighborhoodCounted = hostPerNeighborhoodCount.reduceByKey((a,b) -> a+b);

//  Maps in Tuple2<Neighborhood, Tuple2<hostID, Count>>
JavaPairRDD<String, Tuple2<String, Integer>> neighbourhoodTupleRDD = hostPerNeighborhoodCounted .mapToPair(
    tuple -> new Tuple2<>(_1: tuple._1()._1(), new Tuple2<>(_1: tuple._1()._2(), _2: tuple._2())));

// Group by keys (neighborhood), creating a PairRDD < neighborhood, Iterable(Tuple2<hostID,count>)>
JavaPairRDD<String, Iterable<Tuple2<String, Integer>>> groupedNeighbourhoods = neighbourhoodTupleRDD.groupByKey();

// Map the values of the old RDD to the top 3 hosts for the count value.
//Use a local LinkedList that contains 3 elements to avoid moving potentially large values locally each time and to utilize removeLast to remove the smallest element after sorting.
JavaPairRDD<String, Iterable<Tuple2<String, Integer>>> top3HostPerNeighbourhoods = groupedNeighbourhoods.mapValues(values -> {
   LinkedList<Tuple2<String, Integer>> sortedList = new LinkedList<>();
  for ( Tuple2<String, Integer> t : values){
    sortedList.add(e:t);
    sortedList.sort((t1, t2) -> t2._2().compareTo(anotherInteger: t1._2()));
    while(sortedList.size()>3){
      sortedList.removeLast();
  }}
  return sortedList;
});
// Sort elements by key
JavaPairRDD<String, Iterable<Tuple2<String, Integer>>> orderedTop3HostPerNeighbourhoods = top3HostPerNeighbourhoods.sortByKey();
// Save the results to an output file
orderedTop3HostPerNeighbourhoods.saveAsTextFile(path: outputPath);
// Close the Spark context
sc.close();
```

Figure 9: Final Trasformations

3.3 RESULTS

Results are contained in the Output folder for both the "Cluster" version and the "Local" version, formatted as:

- (neighborhood, [Top 3 Host by number of b&b in that neighborhood]);
  where the top 3 is formatted as:
  - (Host_id, count of b&b)

An example is shown in the figure (Figure 10).

```
(Allerton,[(11305944,1)])
(Arverne,[(9040879,2), (22591516,1), (19866189,1)])
(Astoria,[(49620552,3), (3250450,2), (2736755,2)])
(Battery Park City,[(52950465,1), (50520440,1), (50540315,1)])
(Bay Ridge,[(27634654,2), (50828435,1), (50762019,1)])
(Baychester,[(57165692,1)])
(Bayside,[(73445541,1), (10135994,1)])
(Bedford-Stuyvesant,[(68787921,3), (14933972,3), (60346942,3)])
(Belle Harbor,[(65096495,1)])
(Bensonhurst,[(69977115,2), (75793151,1), (31628863,1)])
(Bergen Beach,[(66810906,1)])
(Boerum Hill,[(16437254,2), (64098159,1), (10474877,1)])
(Borough Park,[(61393656,1), (43192686,1), (62915940,1)])
(Brighton Beach,[(46798824,1), (62535444,1), (6909591,1)])
(Bronxdale,[(52228249,1), (55618434,1)])
(Brooklyn Heights,[(6555513,2), (15239567,1), (14521821,1)])
(Brownsville,[(72318418,1), (47784768,1), (66513544,1)])
(Bushwick,[(41616878,4), (73737053,3), (50600973,3)])
(Cambria Heights,[(48671504,1), (14798138,1)])
(Canarsie,[(36579485,2), (75730551,1), (63291733,1)])
(Carroll Gardens,[(72701423,1), (5238157,1), (40306612,1)])
(Castleton Corners,[(10721093,1)])
(Chelsea,[(22541573,10), (16677326,4), (23863809,2)])
(Chinatown,[(6980995,2), (53308963,2), (36054372,1)])
(City Island,[(56714504,1)])
(Civic Center,[(6980995,2), (9059810,1), (8243103,1)])
(Claremont Village,[(2988712,2)])
(Clifton,[(68252461,1), (25059970,1), (80470317,1)])
(Clinton Hill,[(10366292,3), (54378184,2), (38068387,2)])
(Cobble Hill,[(124866,1), (4337162,1), (3302537,1)])
(Columbia St,[(37820765,1), (14611780,1), (347475,1)])
(Concord,[(13373889,1), (43392243,1)])
(Concourse,[(73831041,1), (69668616,1), (5944946,1)])
(Concourse Village,[(58857198,1), (11017415,1)])
(Corona,[(51278789,2), (5261297,1), (35660592,1)])
(Crown Heights,[(116382,4), (70058270,3), (60059749,2)])
(Cypress Hills,[(66309874,3), (9346894,2), (48764969,1)])
(DUMBO,[(75739500,1)])
(Ditmars Steinway,[(16823940,2), (49568280,2), (61962183,1)])
(Downtown Brooklyn,[(5397742,1), (1642326,1), (5777244,1)])
(Dyker Heights,[(73857837,1), (50045329,1), (11904916,1)])
(East Elmhurst,[(37312959,2), (53199312,1), (4348515,1)])
(East Flatbush,[(77778146,3), (47351539,2), (34991003,2)])
(East Harlem,[(66807700,2), (74330820,2), (50638417,2)])
```

Figure 10: Results