

Topic: MapReduce Implementation for Analysing Guest Satisfaction in the Hotel Industry

Contents

- 1. Introduction**
- 2. Problem Description**
- 3. Associated Dataset**
- 4. Design & Implementation**
 - 4.1 Design of each Mapper and Reducer**
 - 4.1.1 Average guest satisfaction overall scores by city and room type- Job-1**
 - 4.1.2 Average guest satisfaction scores overall by only City type- Job2**
 - 4.1.3 City scoring the highest and the lowest average guest satisfaction score- Job3**
 - 4.2 Internal working of each Mapper and Reducer**
 - 4.2.1 Job- 1**
 - 4.2.2 Job- 2**
 - 4.2.3 Job- 3**
- 5. Results & Evaluation**
- 6. Conclusion**

1. Introduction

The rise of big data has led to a increase in the quantity of data that needs to be processed and analysed. Old data analysis methods are insufficient for processing the large datasets, and there is a need to develop new methods that can handle the volume, velocity, and variety of big data. In this context, High Performance Computing (HPC) infrastructure has become an essential tool for processing and analysing large datasets efficiently and effectively.

HDFS has been specifically designed to store and process substantial volumes of data across distributed clusters. Its architecture comprises a master node, referred to as the NameNode, which oversees metadata and access control, along with multiple worker nodes known as DataNodes that are responsible for data storage and management. The system's foundation on commodity hardware renders it both cost-effective and scalable.

Two principal components constitute the backbone of HDFS: the NameNode and the DataNode. The NameNode is charged with maintaining the file system namespace, monitoring block locations, and managing client access requests. Conversely, the DataNode attends to read and write requests, stores actual data, and handles block replication as directed by the NameNode.

Employing a master-slave architecture, HDFS enables the NameNode to communicate with DataNodes for the purpose of data storage and access management. Clients interact with the NameNode to acquire metadata and obtain data block locations before directly liaising with DataNodes to read or write data.

Among the key advantages of HDFS is its capacity to store and process immense amounts of data at scale. The system is also cost-efficient due to its utilization of commodity hardware. Furthermore, HDFS boasts high fault tolerance and rack awareness, ensuring data availability while minimizing latency. Scalability permits resource adjustments in accordance with file system size, while support for streaming reads enhances data processing efficacy.

A diverse range of industries relies on HDFS, spanning from electric companies that monitor smart grids with phasor measurement units (PMUs) to search engines, ad placement, large-scale image conversion, log processing, machine learning, and data mining applications. By leveraging HDFS, organizations can effectively manage and analyse vast pools of big data, ultimately yielding valuable insights and informed decision-making capabilities (Anushkakhatri, 2022).

2. Problem Description

The examination of Airbnb listings across numerous cities represents a real-world data analytics challenge that can benefit from the application of distributed processing techniques. Specifically, the analysis aims to know the average guest satisfaction rating for each room type (shared, private, or entire home) in each city, as well as identifying the cities with the highest and lowest overall guest satisfaction ratings.

To tackle this challenge, it requires infrastructure to facilitate distributed data processing. By utilizing the multiple map-reduce approach, the data can be processed smoothly and efficiently, allowing for the rapid extraction of insights. This methodology can save time and resources that would be necessary to complete the analysis.

The primary objective of this study is to ascertain: **How do the average guest satisfaction scores for various accommodation categories, such as shared rooms, private rooms, and entire homes, compare across different urban areas, and which specific city locations demonstrate the highest and lowest aggregate satisfaction indices based on a comprehensive analysis of guest experiences?**

Associated Dataset

Column name	Description	Type
realSum	The total price of the Airbnb listing.	Numeric
room_type	The type of room offered (e.g., private room, shared room, entire home/apt).	Categorical
room_shared	Whether the room is shared or not.	Boolean
room_private	Whether the room is private or not.	Boolean
person_capacity	The maximum number of people that can be accommodated in a single listing.	Numeric
host_is_superhost	Whether or not a particular host is identified as a superhost on Airbnb.	Boolean
multi	Whether multiple rooms are provided in one individual listing or not.	Boolean
Biz	Whether a particular listing offers business facilities like meeting area/conference rooms in addition to accommodation options.	Boolean
cleanliness_rating	The rating associated with how clean an individual property was after guests stayed at it.	Numeric
guest_satisfaction_overall	The overall rating which shows how satisfied are guests with their stay after visiting an Airbnb property.	Numeric
bedrooms	The total quantity of bedrooms available among all properties against a single hosting id.	Numeric
dist	Distance from city centre associated with every rental property. (Measurement may vary depending upon scale eg kilometers/miles etc)	Numeric
metro_dist	Distance from metro station associated with every rental property. (Measurement may vary depending upon scale eg kilometers/miles etc)	Numeric
lng	Longitude measurement corresponding to each rental unit.	Numeric
lat	Latitude measurement corresponding to each rental unit.	Numeric
City	Different European cities	Categorical

Table- 1, Description of the dataset

Source of the dataset: <https://zenodo.org/record/4446043#.ZCBtsXbMK3B>

The selected dataset provides a comprehensive overview of various Airbnb listing characteristics across cities. The dataset includes key variables such as the total price of the listing, the type of room offered (private room, shared room, entire home/apartment), the maximum capacity of the listing,

and whether or not the host is designated as an Airbnb superhost. In addition, the dataset contains information regarding the property's cleanliness rating, the overall satisfaction of guests, and the listing's distance from the city centre and metro station. To answer our research question, we will use the 'room type,' 'City,' and 'guest satisfaction overall' characteristics of the dataset.

Nevertheless, the original dataset is separated into separate CSV files based on the names of the cities and weekdays and weekends. The research question necessarily requires merging the files and adding a new column, City.

```
import glob
import pandas as pd

# get all csv files in the current directory
csv_files = glob.glob('C:/Users/amit kedia/Downloads/Hotel/*.csv')

# add a new column with the file name and concatenate
df_concat = pd.concat([pd.read_csv(f).assign(source=os.path.basename(f).rsplit('.')[0]) for f in csv_files], ignore_index=True)

# save the output to a new csv file
df_concat.to_csv('C:/Users/amit kedia/Downloads/Hotel/Concatated_files.csv', index=False)
```

Fig. 1, Concatenation of multiple CSVs

Figure 1 depicts the method for merging multiple CSV files that were saved based on the city name + (weekends/weekdays). Here in the code, I have implemented a For loop in order to read all CSV files in a single directory. Simultaneously, I created a new column named 'source' in which all values originating from a particular CSV file will be assigned the same name as the CSV file. The file is then saved locally within the same directory.

```
def remove_string(s):
    return s.split("_")[0]

mydf['City'] = mydf['City'].apply(remove_string)
```

Fig. 2, Removing the unnecessary part

Figure 2 explains the code implementation for removing the weekends/weekdays written with the name of the city.

Finally, we got the desired values in the 'City' column (Fig. 3).

lat	City
52.41772	amsterdam
52.37432	amsterdam
52.36103	amsterdam
52.37663	amsterdam
52.37508	amsterdam

Fig.3, Output

The dataset is prepared for the further analysis. Before analysing the dataset, it is crucial to understand the structure of input and output and the flow of the Map Reduce algorithm. That is explained in the further sections.

3. Design & Implementation

This section will give a comprehensive explanation of the implementation of map-reduce algorithm and design. The map-reduce algorithm consists of two phases: the Map phase and the reduce phase.

The input data is going to be divided into smaller chunks and will be distributed parallelly on multiple machines during the mapping phase. The map function accepts a key-value pair as input and returns one or more key-value pairs.

In Reduce phase, the key-value pairs generated by the mapper will get shuffled and sorted so that each key's associated with the values are grouped together. Then it will process by reduce function, which produces a final result by aggregating the values. The reducer will accept the output of mapper and return single output value from the array of values.

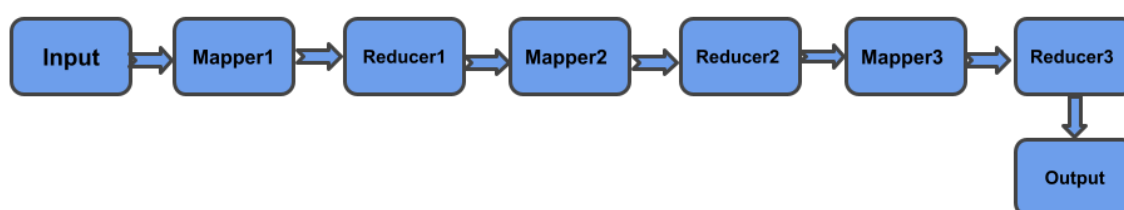


Fig. 4, Design of algorithm

The Chain Mapper algorithm, on the other hand, works by chaining multiple map functions into a pipeline, with the output of each map function serving as input for the subsequent map function in the sequence (Nguyen, 2020). Due to the fact that answering our research question requires multiple processing steps. First, the average Guest Satisfaction score by City and Room Type must be determined. Consequently, it is necessary to find the average city-by-city (see fig. 5). Finally, it is necessary to identify the city with the highest and lowest average guest satisfaction scores. Therefore, it is readily apparent that we must employ the Chain Mapper algorithm.

index	realSum	City	room_type	guest_satisfaction_overall	room_shared	room_private	person_capacity
965	174.34912	amsterdam	Private room	87	FALSE	TRUE	2
966	387.130035	amsterdam	Private room	84	FALSE	TRUE	2
967	195.67408	amsterdam	Private room	89	FALSE	TRUE	2
968	378.693788	amsterdam	Entire home/apt	83	FALSE	FALSE	2
969	319.640053	amsterdam	Private room	86	FALSE	TRUE	2
970	405.642912	amsterdam	Private room	95	FALSE	TRUE	4
971	2486.58402	amsterdam	Entire home/apt	100	FALSE	FALSE	2
972	473.132894	amsterdam	Entire home/apt	98	FALSE	FALSE	2
973	491.645771	amsterdam	Entire home/apt	98	FALSE	FALSE	3
974	1812.8559	amsterdam	Entire home/apt	84	FALSE	FALSE	4
975	399.315727	amsterdam	Entire home/apt	89	FALSE	FALSE	2
976	728.798069	amsterdam	Entire home/apt	97	FALSE	FALSE	4
0	129.824479	athens	Entire home/apt	100	FALSE	FALSE	4
1	138.963748	athens	Entire home/apt	96	FALSE	FALSE	4
2	156.304924	athens	Entire home/apt	98	FALSE	FALSE	3
3	91.6270241	athens	Entire home/apt	99	FALSE	FALSE	4
4	74.051508	athens	Private room	100	FALSE	TRUE	2
5	113.889345	athens	Entire home/apt	96	FALSE	FALSE	6

City	Room_type	Avg_guest_satisfaction
amsterdam	Private room	93.46293
	Entire home/ apt	95.41119
Athens	Private room	95.41119

City with highest satisfaction score: Budapest, 95.16134004943886
City with lowest satisfaction score: Lisbon, 88.99131114874656

Fig. 5, Flow of input and output (Gyódi and Nawaro, 2021)

3.1 Design of each Mapper and Reducer

This section focuses specifically on the design of the individual mappers and reducers used to answer the research question. It is divided into three tasks, each of which is described in detail in one of three subsections.

3.1.1 Average guest satisfaction overall scores by city and room type- Job-1

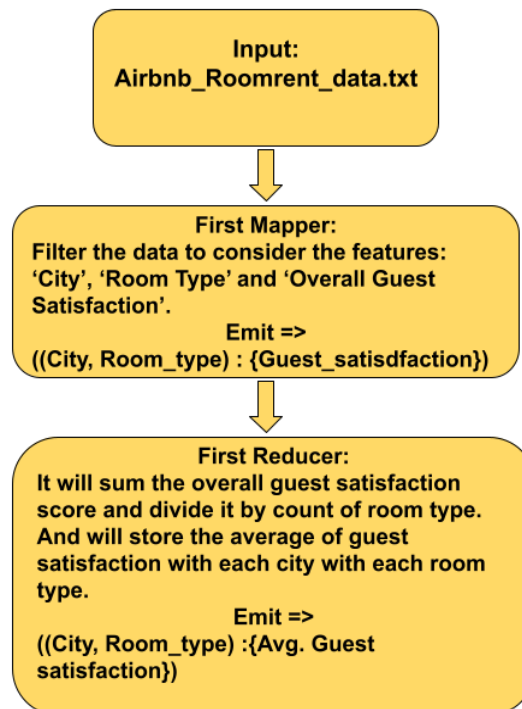


Fig. 6, Job 1 flow chart

A dataset in the form of the text file Airbnb_Roomrent_data.txt serves as the input for the algorithm. The first mapper filters the data to consider only the essential features, such as 'City,' 'Room Type,' and 'Overall Guest Satisfaction'. The mapper then outputs the data as a key-value pair in which the key is a combination of "City" and "Room Type" and the value is "Guest satisfaction."

The first reducer takes the key-value pairs that are output and calculates the average 'Guest satisfaction' score for each unique combination of 'City' and 'Room Type'. This is determined by adding the 'Guest satisfaction' scores for each combination and dividing by the total number of 'Room Type' entries. The reducer finally outputs the data as a key-value pair, where the key is the combination of 'City' and 'Room Type' and the value is the calculated average of 'Guest satisfaction'.

3.1.2 Average guest satisfaction scores overall by only City type- Job2

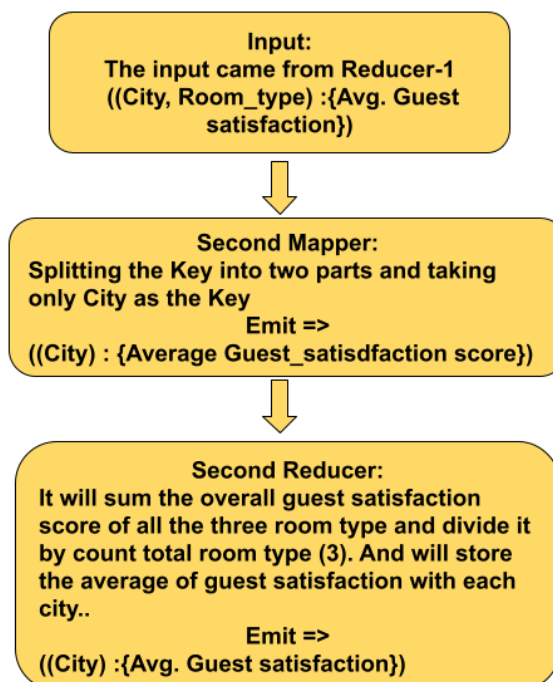


Fig 7, Job-2 flow diagram

Job 2 receives as input the output of Job 1, which contains the average guest satisfaction scores for each city and room type combination, as its input. The second cartographer divides the original key into two parts, retaining only the city as the new key. The system then outputs a key-value pair containing the city and the average guest satisfaction rating for all room types.

This input is used by the second reducer, which calculates the overall average guest satisfaction score for each city by summing the average guest satisfaction scores for all room types and dividing by the total number of room types (which is 3 in this case). The output of the second reducer is a key-value pair consisting of the city and its average rating for overall guest satisfaction.

3.1.3 City scoring the highest and the lowest average guest satisfaction score- Job3

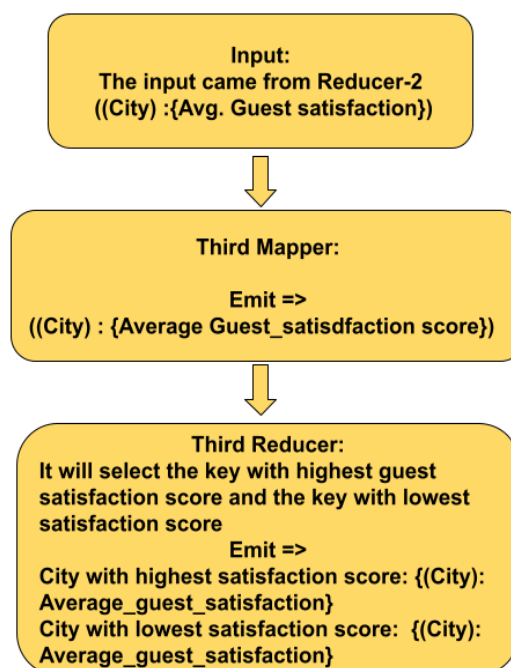


Fig. 8, Job-3 Flow Diagram

The third job receives input from Reducer-2 in the form of pairs of {(City): Average Guest Satisfaction}. The third mapper performs no data transformation and merely outputs identical key-value pairs. The output of the third mapper is fed to the third reducer once more. By iterating through all the keys and values, the third reducer selects the key with the highest guest satisfaction score and the key with the lowest satisfaction score. It then outputs the two selected keys with their respective average guest satisfaction scores in the format (City): Average guest satisfaction for the two selected keys.

In conclusion, the third job is a straightforward operation consisting of a mapper that does not perform data transformation and a reducer that selects the keys with the highest and lowest guest satisfaction scores among all cities. The output of the third reducer produces the desired result in the form (City): Average guest satisfaction for the cities with the highest and lowest satisfaction scores.

3.2 Internal working of each Mapper and Reducer

3.2.1 Job- 1

Stage	Input	Input Type	Output	Output Type
Mapper1 Filter (City, Room type, Guest satisfaction overall)	Data set and Row Number	K: LongWritable V: Text	(City, room_type) : (Guest satisfaction score) Eg: Amsterdam, Entire home/apt: (95, 96, 100...)	K: Text V: DoubleWritable
Reducer1 Finds the mean of guest satisfaction score by city and room type wise	Keys will contain list of unique city and room type combination and values will contain list of all the guest satisf. score	K: Text V: DoubleWritable	(City, room_type) : (Avg. Guest satisfaction score) Eg: Amsterdam, Entire home/apt: (95.0568)	K: Text V: DoubleWritable

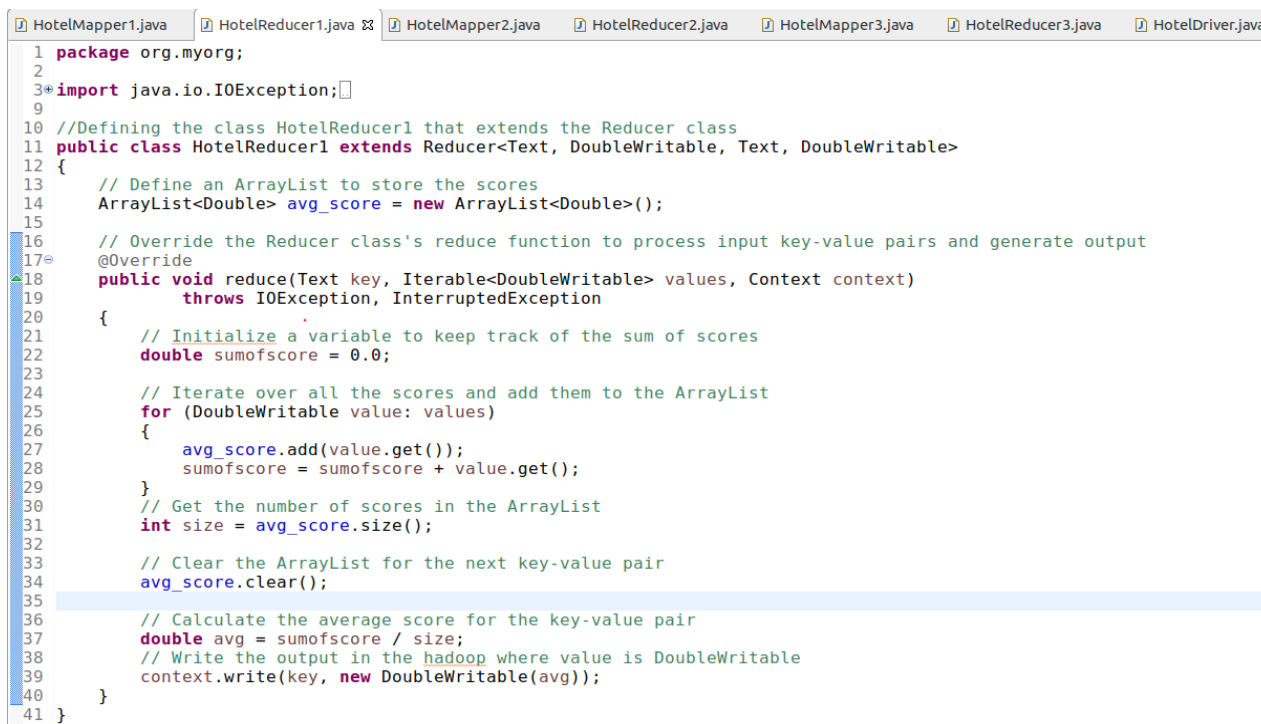
Table-2, Job-1: input and output stage of map reduce

```

1 package org.myorg;
2
3 import java.io.IOException;
4
5 //Define the class HotelMapper1 that extends the Mapper class
6 public class HotelMapper1 extends Mapper<LongWritable, Text, Text, DoubleWritable>{
7     // Define a DoubleWritable object to hold the score for each line of input
8     private final static DoubleWritable tempDoubleWritable = new DoubleWritable();
9     // Define a Text object to hold the concatenation of the city and room type for each line of input
10    private Text City_roomtype = new Text();
11
12    //Override the Mapper class's map function to read and process input records
13    @Override
14    public void map(LongWritable key, Text value, Context context)
15        throws IOException, InterruptedException
16    {
17        // Split the input line into an array of values using comma as the delimiter
18        String[] line = value.toString().split(",");
19        // Concatenate the city and room type fields and set it as the key for this record
20        City_roomtype = new Text(line[20] + ',' + line[2]);
21
22        // Check if the score field is a valid numeric value because the first line is text
23        if (line[10].matches("-?\\d+(\\.\\d+)?")) {
24            // Convert the score value to a double and set it as the value for this record
25            double score = Double.parseDouble(line[10].trim());
26            tempDoubleWritable.set(score);
27            // Emit the key-value pair to the output collector
28            context.write(City_roomtype, tempDoubleWritable);
29        } else {
30            // Handle the case where the value is not numeric
31            System.err.println("Invalid score: " + line[10]);
32        }
33
34        // Emit the key-value pair to the output collector
35        context.write(City_roomtype, tempDoubleWritable);
36    }
37 }

```

Fig. A, Mapper1



```

1 package org.myorg;
2
3 import java.io.IOException;
4
5 //Defining the class HotelReducer1 that extends the Reducer class
6 public class HotelReducer1 extends Reducer<Text, DoubleWritable, Text, DoubleWritable>
7 {
8     // Define an ArrayList to store the scores
9     ArrayList<Double> avg_score = new ArrayList<Double>();
10
11     // Override the Reducer class's reduce function to process input key-value pairs and generate output
12     @Override
13     public void reduce(Text key, Iterable<DoubleWritable> values, Context context)
14         throws IOException, InterruptedException
15     {
16         // Initialize a variable to keep track of the sum of scores
17         double sumofscore = 0.0;
18
19         // Iterate over all the scores and add them to the ArrayList
20         for (DoubleWritable value: values)
21         {
22             avg_score.add(value.get());
23             sumofscore = sumofscore + value.get();
24         }
25         // Get the number of scores in the ArrayList
26         int size = avg_score.size();
27
28         // Clear the ArrayList for the next key-value pair
29         avg_score.clear();
30
31         // Calculate the average score for the key-value pair
32         double avg = sumofscore / size;
33         // Write the output in the hadoop where value is DoubleWritable
34         context.write(key, new DoubleWritable(avg));
35     }
36 }

```

Fig. B, Reducer 1

The table-2 and the code(A & B) provide an overview of the input, input type, output, and output type for the first stage of the MapReduce algorithm. In this stage, the input data is a dataset consisting of rows containing the city, room type, and overall guest satisfaction rating.

The mapper function filters the data to only consider the city and room type features and emits the key-value pairs of city and room type to the guest satisfaction score. The mapper output key-value pairs have the following types:

Key: Text (city and room type combination)

Value: DoubleWritable (guest satisfaction score)

The reducer function then calculates the mean guest satisfaction score for each unique city and room type combination. The reducer input key-value pairs have the same types as the mapper output. The reducer output key-value pairs have the following types:

Key: Text (city and room type combination)

Value: DoubleWritable (average guest satisfaction score)

The output of reducer1 is a list of key-value pairs of city and room type combinations to their corresponding average guest satisfaction score. This output will be passed on as input to the next stage of the MapReduce algorithm.

3.2.2 Job- 2

Stage	Input	Input Type	Output	Output Type
Mapper2 takes the input from Reducer1 and filter (City, Avg. Guest satisfaction overall)	Key value pair from the Reducer1	K: Text V: DoubleWritable	(City): {Ave. Guest satisfaction score with different room type} Eg: (Amsterdam): {(65.56, 89.45, 98.78)}	K: Text V: DoubleWritable
Reducer2 Finds the mean of guest satisfaction score by city wise (adding the average of room type and divide it by count of room types)	Keys will contain list of combination of unique city names & room type and values will contain list of Avg. satisf. score	K: Text V: DoubleWritable	((City): {Avg. Guest satisfaction}) Eg: (Amsterdam): {(93.89)}	K: Text V: DoubleWritable

Table-3, Job-2: input and output stage of map reduce

```

HotelMapper1.java  HotelReducer1.java  HotelMapper2.java  HotelReducer2.java  HotelMapper3.java  HotelReducer3.java  HotelDriver.java
1 package org.myorg;
2
3 import java.io.IOException;
4
5 //Defining the class for the second mapper which takes input from the output of Reducer1
6 public class HotelMapper2 extends Mapper<LongWritable, Text, Text, DoubleWritable>{
7     // Creating a DoubleWritable object to hold the score
8     private final static DoubleWritable tempDoubleWritable = new DoubleWritable();
9     // Creating a Text object to hold the city name
10    private Text City = new Text();
11
12    @Override
13    public void map(LongWritable key, Text value, Context context)
14        throws IOException, InterruptedException
15    {
16        // Split the input value by tab character as key and value are seperated by tab
17        String[] line = value.toString().split("\t");
18
19        // Split the first field (which is city and room type) by comma character
20        String[] temp = line[0].toString().split(",");
21
22        // Setting the city name as Text object
23        City = new Text(temp[0]);
24        // To check if this is the first line (header) and skip it
25        if (key.get() == 0) {
26            return;
27        }
28        // Get the average score from the second field of the input
29        double avg_score = Double.parseDouble(line[1].trim());
30        // Set the score as DoubleWritable object
31        tempDoubleWritable.set(avg_score);
32        // Emit the key-value pair (city name, score) to the reducer
33        context.write(City, tempDoubleWritable);
34    }
35 }

```

Fig. C, Mapper 2

```

1 package org.myorg;
2
3 import java.io.IOException;
4
5
6
7
8
9
10
11 public class HotelReducer2 extends Reducer<Text, DoubleWritable, Text, DoubleWritable>
12 {
13     ArrayList<Double> final_avg_score = new ArrayList<Double>();
14
15     @Override
16     public void reduce(Text key, Iterable<DoubleWritable> values, Context context)
17         throws IOException, InterruptedException
18     {
19         double sumofscore = 0.0;
20
21         // Loop through all the values in the Iterable
22         for (DoubleWritable value: values)
23         {
24             // Add each value to the final_avg_score ArrayList
25             final_avg_score.add(value.get());
26
27             // Calculate the sum of all the scores for the current key
28             sumofscore = sumofscore + value.get();
29         }
30
31         // Get the number of scores for the current key
32         int size = final_avg_score.size();
33
34         // Clear the ArrayList to avoid memory leaks
35         final_avg_score.clear();
36
37         // Calculate the average score for the current key
38         double avg = sumofscore / size;
39
40         // Write the key-value pair to the context
41         context.write(key, new DoubleWritable(avg));
42     }
43 }

```

Fig. D, Reducer 2

Inferring from Table-3 and code (C & D), in the second stage of the MapReduce algorithm, the input data is taken from the output of Reducer1 and filtered by Mapper2 to retrieve only the (City, Avg. Guest satisfaction overall) key-value pairs. Reducer2 calculates the mean guest satisfaction score for each city by adding the average of room types and dividing it by the count of room types. The output of Reducer2 is in the form of ((City): {Avg. Guest satisfaction}). The choice of input and mapper is based on the requirement to calculate the average guest satisfaction score for each city, regardless of the room type. The choice of reducer function is based on the need to aggregate the average guest satisfaction scores for each city efficiently.

3.2.3 Job-3

Stage	Input	Input Type	Output	Output Type
Mapper3 takes the input from Reducer2 and filter (City, Avg. Guest satisfaction overall)	Key value pair from the Reducer2	K: LongWritable V: Text	Key: ((City): {Avg. Guest satisfaction}) Value: NullWritable	K: Text V: NullWritable
Reducer3 Finds the city with highest and lowest guest satisfaction score.	Keys will contain list of combination of unique city names & room type and values will contain list of Avg. satisf. score	K: Text V: NullWritable	City with highest satisfaction score: Budapest, 95.16134004943886 City with lowest satisfaction score: Lisbon, 88.99131114874656	K: Text V: Text

Table-4, Job-3: input and output stage of map reduce.

```

1 package org.myorg;
2
3 import java.io.IOException;
4
5
6
7
8
9 public class HotelMapper3 extends Mapper<LongWritable, Text, Text, NullWritable> {
10
11     @Override
12
13     // the input for the Mapper3 comes from the output of the reducer2
14     public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
15
16         // Write each input line as a Text key with a NullWritable value
17         // This is done to ensure that all records pass through to the reducer
18         context.write(value, NullWritable.get());
19     }
20 }

```

Fig. E, Mapper 3

```

1 package org.myorg;
2
3 import java.io.IOException;
4 import org.apache.hadoop.io.*;
5 import org.apache.hadoop.mapreduce.*;
6
7 //The reducer takes input in the form of Text and NullWritable and outputs Text and Text.
8 public class HotelReducer3 extends Reducer<Text, NullWritable, Text, Text> {
9
10     // Creating four private instances to store city names and satisfaction scores
11     private Text highestCity = new Text();
12     private Text lowestCity = new Text();
13     private DoubleWritable highestScore = new DoubleWritable(Double.MIN_VALUE);
14     private DoubleWritable lowestScore = new DoubleWritable(Double.MAX_VALUE);
15
16     @Override
17     public void reduce(Text key, Iterable<NullWritable> values, Context context)
18         throws IOException, InterruptedException {
19
20         // Splitting the key value pair that is received from the mapper
21
22         String line = key.toString();
23         String[] tokens = line.split("\t");
24
25         // Extract city name and satisfaction score
26         // Storing the first city and score into the static variable
27         String city = tokens[0];
28         double score = Double.parseDouble(tokens[1]);
29
30         // Will compare the static variables with the next value
31         // Update highest and lowest scores if necessary
32         if (score > highestScore.get()) {
33             highestScore.set(score);
34             highestCity.set(city);
35         }
36         if (score < lowestScore.get()) {
37             lowestScore.set(score);
38             lowestCity.set(city);
39         }
40     }
41
42     @Override
43     protected void cleanup(Context context) throws IOException, InterruptedException {
44         // Output the final result
45         context.write(new Text("City with highest satisfaction score: "), new Text(highestCity + " " + highestScore));
46         context.write(new Text("City with lowest satisfaction score: "), new Text(lowestCity + " " + lowestScore));
47     }
48 }

```

Fig. F, Reducer 3

The third and final stage of the MapReduce algorithm takes input from Reducer2 and identifies the city with the highest and lowest guest satisfaction score. Mapper3 takes the key-value pair of the city and its average guest satisfaction score from Reducer2 and passes it to Reducer3 after filtering the data. Reducer3 selects the city with the highest and lowest guest satisfaction score and outputs the result as a key-value pair.

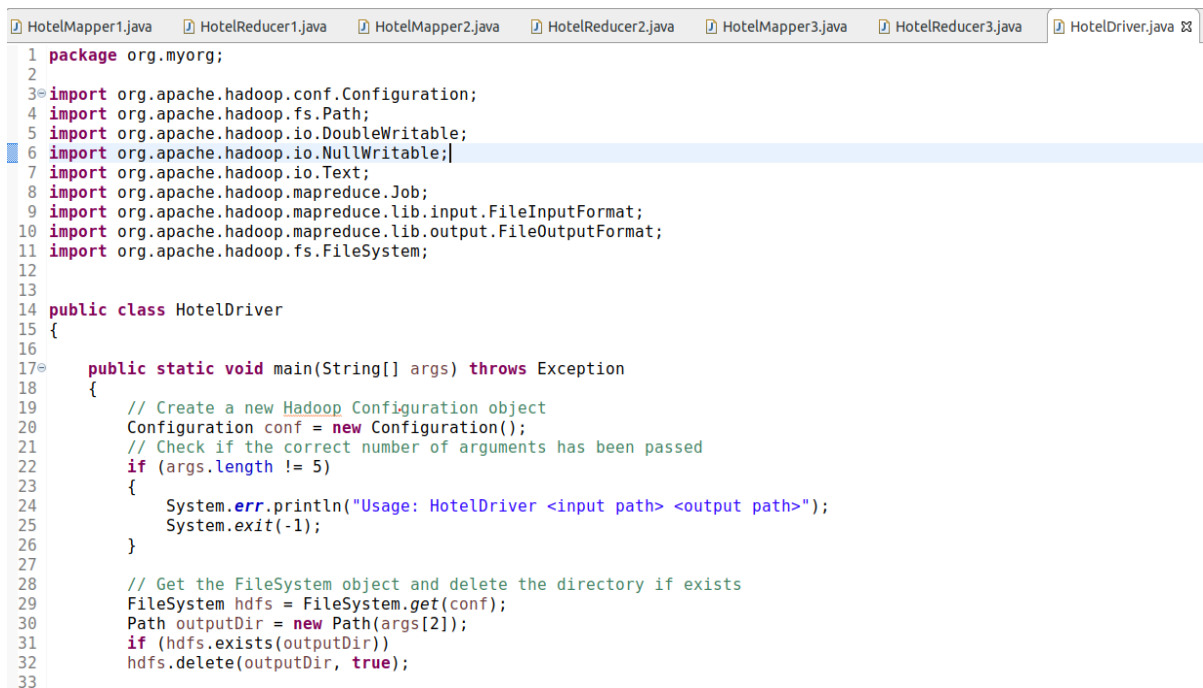
The choice of using NullWritable as the value type in Mapper3 allows for the keys to be passed through to Reducer3 without any additional processing or filtering. This simplifies the logic in Reducer3 and enables the selection of the city with the highest and lowest satisfaction scores.

The final output received from the job would be in the name of the city with the highest and lowest guest satisfaction score with the values of corresponding satisfaction score pairs. It will provide the insights to hotel manager to improve their services.

Overall, the MapReduce algorithm efficiently processes and analyses large datasets to derive meaningful insights, making it a valuable tool for data analysis in various industries.

4. Results & Evaluation

This section will discuss and analyse the results of the three Jobs, as well as answer the previously posed research question: "How do the average guest satisfaction scores for various accommodation categories, such as shared rooms, private rooms, and entire homes, compare across different urban areas, and which specific city locations demonstrate the highest and lowest aggregate satisfaction indices based on a comprehensive analysis of guest experiences?"



```

1 package org.myorg;
2
3 import org.apache.hadoop.conf.Configuration;
4 import org.apache.hadoop.fs.Path;
5 import org.apache.hadoop.io.DoubleWritable;
6 import org.apache.hadoop.io.NullWritable;
7 import org.apache.hadoop.io.Text;
8 import org.apache.hadoop.mapreduce.Job;
9 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
10 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
11 import org.apache.hadoop.fs.FileSystem;
12
13
14 public class HotelDriver
15 {
16
17     public static void main(String[] args) throws Exception
18     {
19         // Create a new Hadoop Configuration object
20         Configuration conf = new Configuration();
21         // Check if the correct number of arguments has been passed
22         if (args.length != 5)
23         {
24             System.err.println("Usage: HotelDriver <input path> <output path>");
25             System.exit(-1);
26         }
27
28         // Get the FileSystem object and delete the directory if exists
29         FileSystem hdfs = FileSystem.get(conf);
30         Path outputDir = new Path(args[2]);
31         if (hdfs.exists(outputDir))
32             hdfs.delete(outputDir, true);
33

```



```

34      // Define the first MapReduce job
35      Job job1;
36      job1=Job.getInstance(conf, "Avg score");
37      job1.setJarByClass(HotelDriver.class);
38
39      // Set input and output paths for the first job
40      FileInputFormat.addInputPath(job1, new Path(args[1]));
41      FileOutputFormat.setOutputPath(job1, new Path(args[2]));
42
43      // Set mapper and reducer classes for the first job
44      job1.setMapperClass(HotelMapper1.class);
45      job1.setReducerClass(HotelReducer1.class);
46
47      // Set output key and value classes for the first job
48      job1.setOutputKeyClass(Text.class);
49      job1.setOutputValueClass(DoubleWritable.class);
50
51      // Wait for the first job to complete
52      job1.waitForCompletion(true);
53
54      // Define the second MapReduce job
55      Job job2;
56      job2=Job.getInstance(conf, "Avg score");
57      job2.setJarByClass(HotelDriver.class);
58
59      // Set input and output paths for the second job
60      FileInputFormat.addInputPath(job2, new Path(args[2]));
61      FileOutputFormat.setOutputPath(job2, new Path(args[3]));
62
63      // Set mapper and reducer classes for the second job
64      job2.setMapperClass(HotelMapper2.class);
65      job2.setReducerClass(HotelReducer2.class);
66
67      // Set output key and value classes for the second job
68      job2.setOutputKeyClass(Text.class);
69      job2.setOutputValueClass(DoubleWritable.class);
70
71      // Wait for the second job to complete
72      job2.waitForCompletion(true);
73
74      // Define the third MapReduce job
75      Job job3;
76      job3=Job.getInstance(conf, "Min Max Score");
77      job3.setJarByClass(HotelDriver.class);
78
79      // Set input and output paths for the third job
80      FileInputFormat.addInputPath(job3, new Path(args[3]));
81      FileOutputFormat.setOutputPath(job3, new Path(args[4]));
82
83      // Set mapper and reducer classes for the third job
84      job3.setMapperClass(HotelMapper3.class);
85      job3.setReducerClass(HotelReducer3.class);
86      job3.setMapOutputKeyClass(Text.class);
87      job3.setMapOutputValueClass(NullWritable.class);
88
89      // Set output key and value classes for the third job
90      job3.setOutputKeyClass(Text.class);
91      job3.setOutputValueClass(Text.class);
92
93      // Wait for the third job to complete
94      System.exit(job3.waitForCompletion(true) ? 0 : 1);
95  }
96 }

```

Fig. G, Driver Class

The shown driver class is in charge of configuring and executing MapReduce jobs for processing Airbnb data. It first verifies that the correct number of arguments has been passed to the programme, and if the output directory already exists, it deletes it.

In this driver class, three MapReduce jobs are defined. The first task (job1) involves calculating the average guest satisfaction rating for each city and room type combination. The second task (job2) identifies the city with the highest average guest satisfaction rating. The third task (job3) is to determine which city has the highest and lowest average guest satisfaction scores.

Each job's input and output paths, mapper and reducer classes, and output key and value classes are configured. The output value class of the first and second jobs is DoubleWritable, which represents the average guest satisfaction score. NullWritable is the map output value class for the third job, while Text is the final output value class.

Text is the output key class for both the first and second jobs, representing the city and room type combination. The third job also uses Text as the output key class, but in a format that includes the city and the highest and lowest guest satisfaction ratings.

Note: The output of each of the jobs will be stored in the different directories. All the directories are named as first_output, second_output and third_output for Job-1, Job-2 & Job-3 respectively.



Fig. 8, output folder names

```
(base) hadoop@hadoop-VirtualBox:~$ start-dfs.sh
Starting namenodes on [localhost]
Starting datanodes
Starting secondary namenodes [hadoop-VirtualBox]
(base) hadoop@hadoop-VirtualBox:~$ start-yarn.sh
Starting resourcemanager
Starting nodemanagers
(base) hadoop@hadoop-VirtualBox:~$ hdfs dfs -mkdir /input
(base) hadoop@hadoop-VirtualBox:~$ hadoop jar Downloads/HotelDriver.jar HotelDriver /input/Airbnb_Roomrent_data.txt first_output second_output third_output
```

Fig. 9, to execute the jar file

Here, we made an input directory in the HDFS system to store our input text file in Hadoop system. we have made a directory first_output to store the output from the Job1. Similarly, Second_output and Third_output is to store output from the Job2 and Job3 respectively.

```
(base) hadoop@hadoop-VirtualBox:~$ hdfs dfs -cat first_output/part-r-00000
City,room_type 0.0
amsterdam,Entire home/apt 95.41119005328596
amsterdam,Private room 93.46292372881356
amsterdam,Shared room 92.8
athens,Entire home/apt 95.16194581280789
athens,Private room 93.14105793450882
athens,Shared room 92.090909090909091
barcelona,Entire home/apt 89.22878228782288
barcelona,Private room 91.56559894690653
barcelona,Shared room 89.33333333333333
berlin,Entire home/apt 93.99092970521542
berlin,Private room 94.71026814911707
berlin,Shared room 90.24657534246575
budapest,Entire home/apt 94.7383672332126
budapest,Private room 93.17422434367542
budapest,Shared room 97.57142857142857
lisbon,Entire home/apt 91.62429087158328
lisbon,Private room 90.20099392600773
lisbon,Shared room 85.14864864864865
london,Entire home/apt 89.375
london,Private room 91.66522755891347
london,Shared room 88.7
paris,Entire home/apt 91.95460824945727
paris,Private room 92.30779305828422
paris,Shared room 92.11702127659575
rome,Entire home/apt 93.51249775220285
rome,Private room 92.51389693109438
rome,Shared room 87.41666666666667
vienna,Entire home/apt 93.62722970513288
vienna,Private room 94.2157622739018
vienna,Shared room 88.125
```

Fig. 9, Output of First Job

Amsterdam	Entire home/apt	95.4111
	Private room	93.4629
	Shared room	92.8

From the above output, it can be clearly observed that we are getting the average guest satisfaction ratings for particularly city and each room type available in the dataset. This output is stored in first_output directory. Moreover, it answers our half of the research question: How do the mean guest satisfaction scores for various accommodation categories, such as shared rooms, private rooms, and entire homes, compare across different urban areas?

```
(base) hadoop@hadoop-VirtualBox:~$ hdfs dfs -cat second_output/part-r-00000
amsterdam 93.89137126069984
athens 93.46463761274192
barcelona 90.04257152268758
berlin 92.98259106559942
budapest 95.16134004943886
lisbon 88.99131114874656
london 89.91340918630449
paris 92.12647419477908
rome 91.14768711665464
vienna 91.98933065967822
```

Fig. 10, Output of second Job

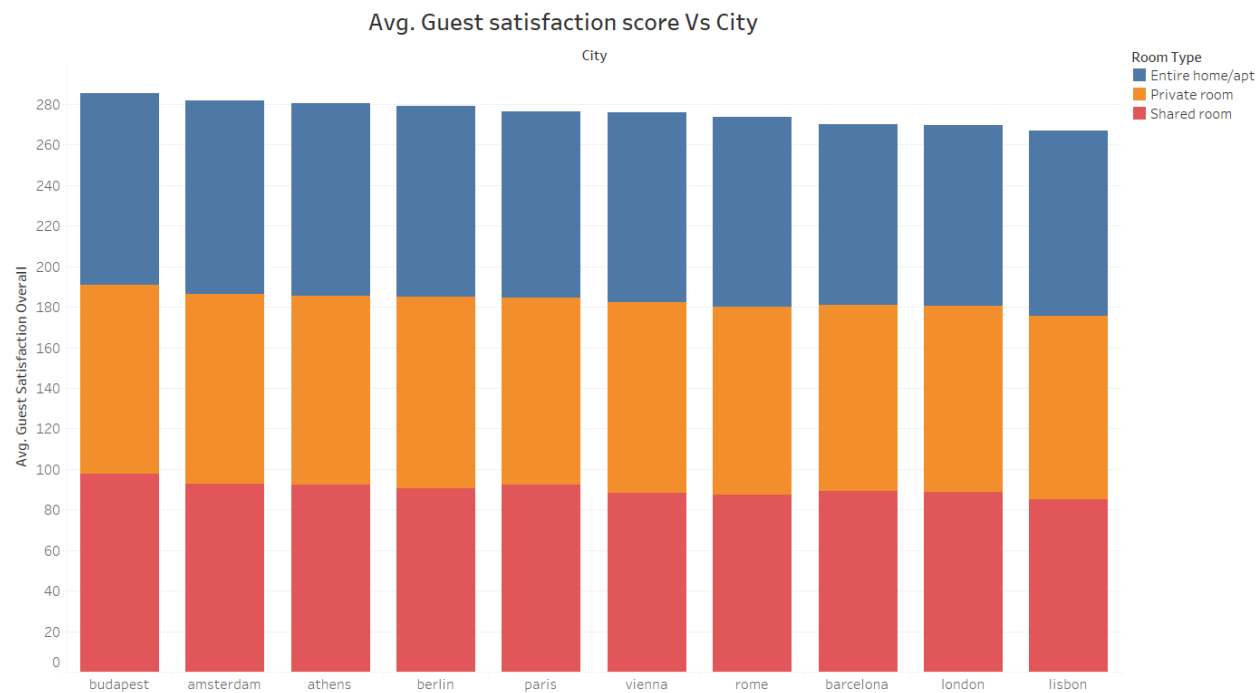
Amsterdam: 93.89137

The above output we got from the Job-2. The output stored in the second_output directory. Here, it can clearly be observed that the Job-2 have aggregated the average guest satisfaction score by each city and divided by 3 (Number of Room_type). The reason to implement this Job is to answer the second part of our research question. For that we need the average guest satisfaction score by each city. Therefore, to calculate the average we need to create this Job. This output would be useful for the next job and from that we will get our answer for the research question.

```
(base) hadoop@hadoop-VirtualBox:~$ hdfs dfs -cat third_output/part-r-00000
City with highest satisfaction score:  budapest 95.16134004943886
City with lowest satisfaction score:  lisbon 88.99131114874656
```

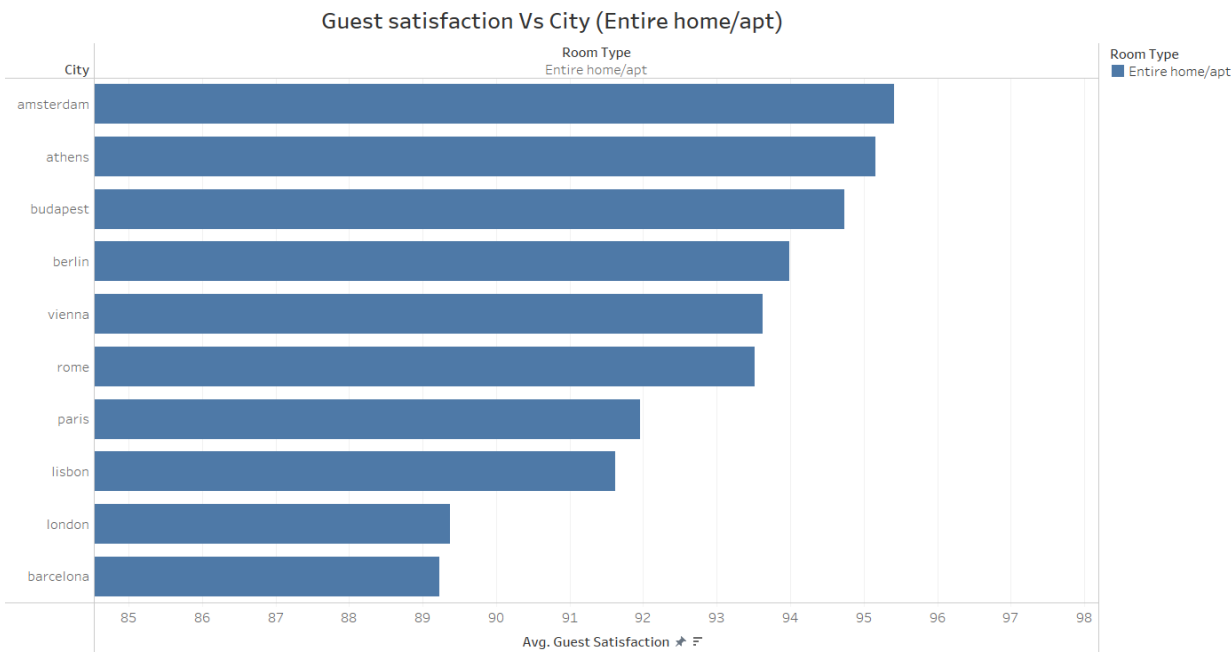
Fig. 11, Final output of third Job

From this we can clearly say that Budapest has topped the list of 10 European city which have the most satisfied guests for all room types included. On the other hand, Lisbon scored the least satisfaction score. Therefore, it gives the answer for our 2nd half of our research question.



Average of Guest Satisfaction Overall for each City. Color shows details about Room Type.

Fig. 12, Bar chart showing the city with highest and lowest overall satisfaction score



Average of Guest Satisfaction Overall for each City broken down by Room Type. Color shows details about Room Type. The view is filtered on Room Type, which keeps Entire home/apt.

Fig. 13, Bar chart showing the city with highest and lowest satisfaction score in Entire home/apt category of room type

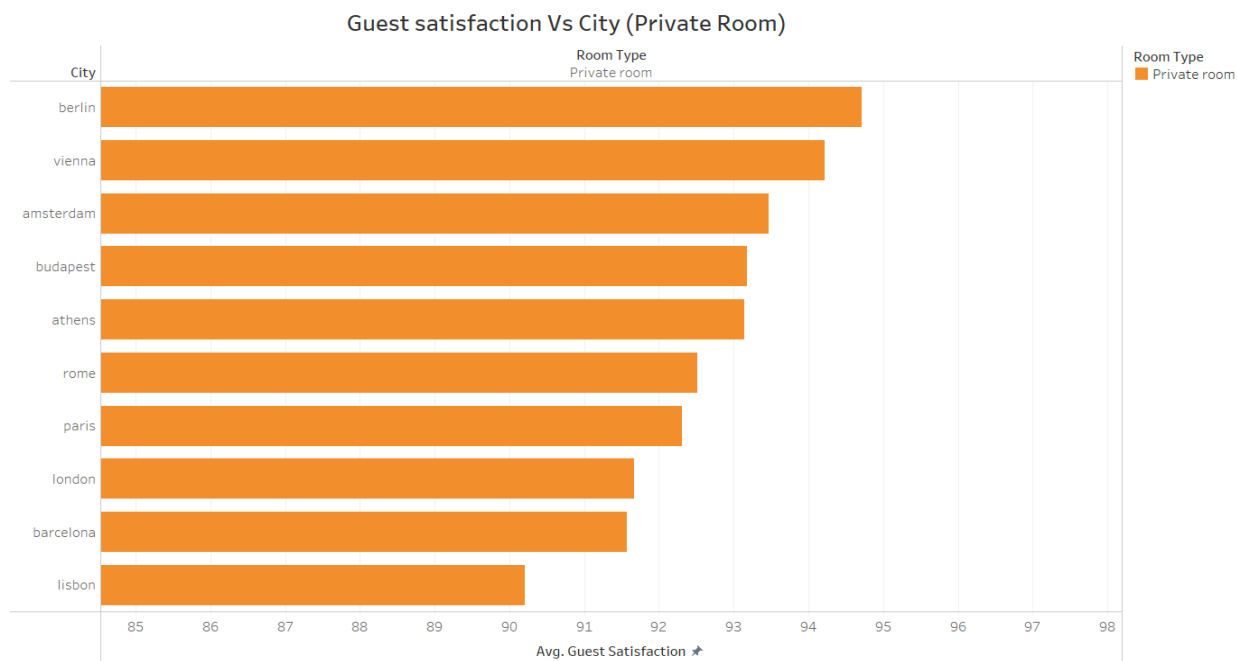


Fig. 14, Bar chart showing the city with highest and lowest satisfaction score in Private room category of room type

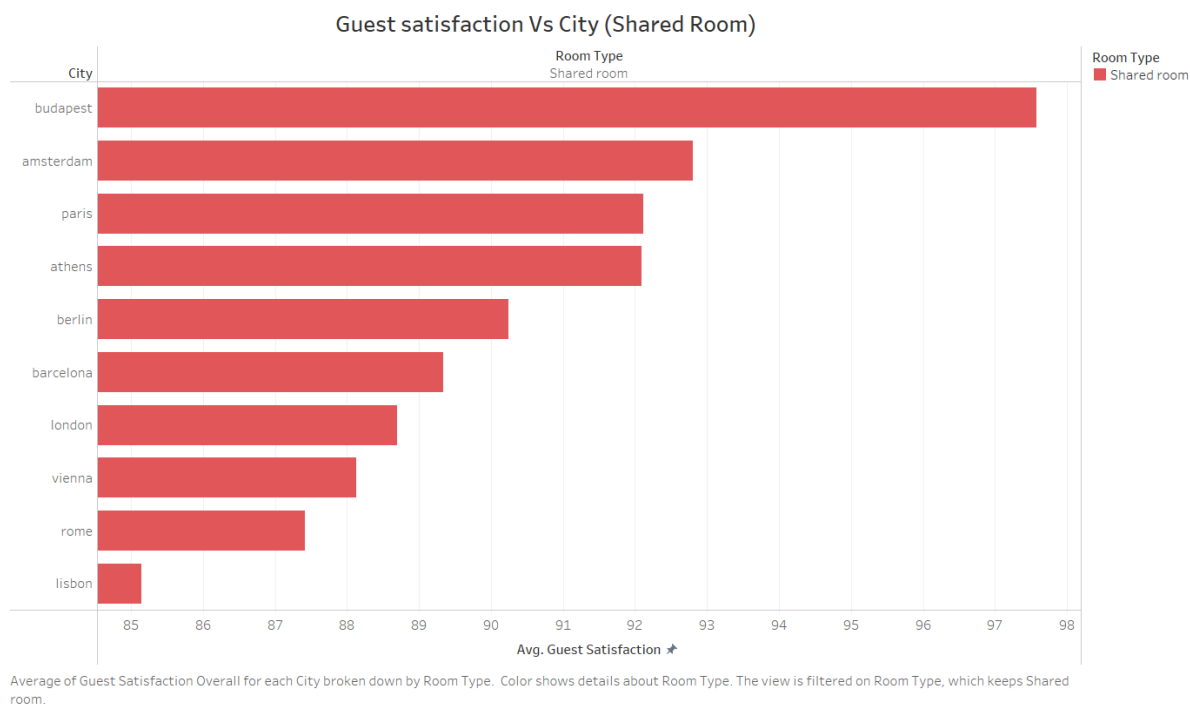


Fig. 15, Bar chart showing the city with highest and lowest satisfaction score in Shared room category of room type

From the above charts, it can be clearly concluded that which are the cities have the highest and lowest ratings and can also compare room type wise in the further graphs.

From Fig. 12, the Budapest has the highest guest satisfaction score whereas Lisbon has the lowest satisfaction score. Glancing at Fig. 13, Amsterdam tops the list in Entire home/apt room type while Barcelona has the least score. However, from Fig. 14, Berlin shows the highest score in Private room

category, but Lisbon again scored the lowest. From Fig. 15, it can be clearly visible that Budapest scored the highest score in Shared room category whereas Lisbon scored the lowest. Therefore, Airbnb need to look after the services in Lisbon specially in private and shared room categories.

5. Conclusion

Based on the map reduce output we received for the research question " How do the mean guest satisfaction scores for various accommodation categories, such as shared rooms, private rooms, and entire homes, compare across different urban areas, and which specific metropolitan locations demonstrate the highest and lowest aggregate satisfaction indices based on a comprehensive analysis of guest experiences?", we were able to derive valuable insights.

The output provides the average satisfaction score for each room type (shared, private, and entire home/apt) in each city. For instance, the average satisfaction score for an entire home/apt in Budapest is 94.74, whereas for a shared room in Lisbon, it is 85.15.

Furthermore, we derived the overall average satisfaction score for each city. The city with the highest guest satisfaction score is 95.16 and the city with the lowest score is 88.99. This analysis will help the stakeholders to improve the guest satisfaction score while analysing the reason for such difference in score.

In conclusion, this assignment has provided the effectiveness of using HPCI, specifically the Hadoop MapReduce framework, to process and analyse large datasets. Through the implementation of a multi-stage MapReduce algorithm, I successfully identified the highest and lowest average guest satisfaction across the hotel industry.

After implementing the project, I got to learn the HPCI techniques and concepts such as parallel processing, error solving and partitioning. I got the practical learning of how to implement the map-reduce program to get the insights from the large dataset.

References

1. Anushkakhatri (2022). *Workings of Hadoop Distributed File System (HDFS)*. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2022/05/workings-of-hadoop-distributed-file-system-hdfs/> [Accessed 5 Apr. 2023].
2. Gyódi, K. and Nawaro, Ł. (2021). Determinants of Airbnb prices in European cities: A spatial econometrics approach (Supplementary Material). [online] Zenodo. Available at: <https://zenodo.org/record/4446043#.ZDpS4HbMK3C> [Accessed 1 Apr. 2023].
3. Nguyen, T. (2020). Chaining Multiple MapReduce Jobs with Hadoop/ Java. [online] Medium. Available at: <https://towardsdatascience.com/chaining-multiple-mapreduce-jobs-with-hadoop-java-832a326cbfa7>. [Accessed 6 Apr. 2023]