<u>**Topic: Implementation of Map-reduce Program to find top 3 cities average spent in India**</u>

Contents

1. Introduction

In the rapidly evolving world of High-Performance Computational Infrastructure (HPCI), leveraging cutting-edge technologies and algorithms to process large volumes of data efficiently has become increasingly essential for various fields, including scientific research, business analytics, and artificial intelligence. HPCI systems have proven to be instrumental in unlocking new insights, optimizing resource usage, and accelerating decision-making processes. As such, a deep understanding of the concepts, techniques, and frameworks involved in building high-performance computing environments is vital for professionals and researchers alike.

In this context, we explore the design, implementation, and analysis of a MapReduce program built on the Hadoop framework to process a large dataset of credit card transactions from India. The purpose of this project is to identify the top 3 cities with the highest average transaction amounts, showcasing the power of HPCI in handling and analyzing massive datasets with speed and efficiency. Through this practical exercise, participants will gain hands-on experience in creating and optimizing a multi-stage MapReduce pipeline, working with Hadoop and its associated components, and deriving valuable insights from the data. Ultimately, this knowledge will serve as a foundation for future exploration and development within the HPCI domain, enabling individuals to tackle even more complex and challenging computational problems.

2. Problem Description

The provided dataset contains transaction information from various cities, with each record representing a unique transaction, including index, city, date, card type, expense type, gender, and transaction amount. The goal of this assignment is to develop a Hadoop MapReduce application using multiple mappers and reducers to analyze the transaction data and answer two primary research questions: **What is the average amount spent by each city, and which are the top 3 cities that have the highest average spent?**

To effectively answer these questions, the MapReduce program will process the dataset to produce two separate output files. The first output will display the city-wise average spent in a "City_name : Average_amount_spent" format. The second output will showcase the top 3 cities with the highest average spent, presented as "The Top 3 cities with highest average spent is: 'City_name' : 'Average_amount_spent', ...". The application should efficiently utilize Hadoop's data processing capabilities to yield accurate and organized results, enabling a comprehensive understanding of the cities' transactional behavior and spending patterns.

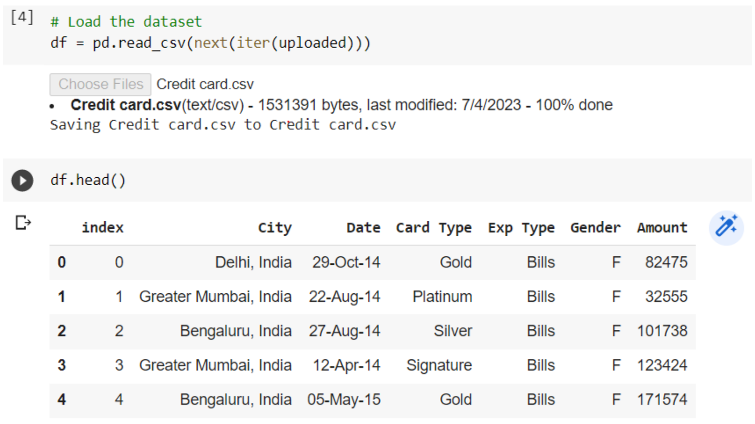
3. Associated Dataset

Column name	Description	Type
City	The city in which the transaction took place. (String)	Categorical
Date	The date of the transaction. (Date)	Date
Card Type	The type of credit card used for the transaction. (String)	Categorical
Exp Type	The type of expense associated with the transaction. (String)	Categorical
Gender	The gender of the cardholder. (String)	Binary
Amount	The amount of the transaction. (Number)	Number

This dataset is of credit card transactions in India, capturing a variety of variables such as gender, card type, city, and expense type to reveal insightful spending preferences and patterns. The dataset provides valuable insight into the financial behaviour and economic landscape of India by analysing the spending habits of individuals in various Indian cities. Researchers and businesses alike can use this data to uncover more in-depth trends in customer spending, uncover intriguing correlations between data points, and gain a deeper understanding of the dynamics underlying consumer decisions. The dataset is a valuable resource for individuals interested in exploring data analysis techniques, generating objective insights, and leveraging the power of data-driven decision-making to support informed business strategies.

The source of the data is: https://www.kaggle.com/datasets/thedevastator/analyzing-credit-card-spending-habits-in-india

However, in the column named City, each and every city mentions about the same country name separated by comma in between city name and country. But for research question to answer we have to remove the unnecessary part.


```
[4] # Load the dataset
    df = pd.read_csv(next(iter(uploaded)))
```

Choose Files Credit card.csv
- **Credit card.csv**(text/csv) - 1531391 bytes, last modified: 7/4/2023 - 100% done

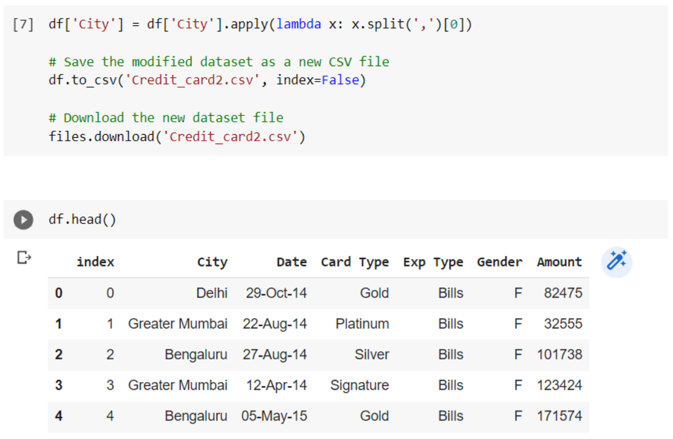
```
Saving Credit card.csv to Credit card.csv
```

```
df.head()
```

```
   index                  City       Date  Card Type  Exp Type  Gender  Amount
0      0          Delhi, India  29-Oct-14       Gold     Bills       F   82475
1      1  Greater Mumbai, India  22-Aug-14   Platinum     Bills       F   32555
2      2       Bengaluru, India  27-Aug-14     Silver     Bills       F  101738
3      3  Greater Mumbai, India   12-Apr-14  Signature     Bills       F  123424
4      4       Bengaluru, India  05-May-15       Gold     Bills       F  171574
```

Fig 1: Data before cleaning

From the fig 1, it can be clearly visible that in each value of city the country name India is there.



```
[7] df['City'] = df['City'].apply(lambda x: x.split(',')[0])

    # Save the modified dataset as a new CSV file
    df.to_csv('Credit_card2.csv', index=False)

    # Download the new dataset file
    files.download('Credit_card2.csv')
```

```
df.head()
```

	index	City	Date	Card Type	Exp Type	Gender	Amount
0	0	Delhi	29-Oct-14	Gold	Bills	F	82475
1	1	Greater Mumbai	22-Aug-14	Platinum	Bills	F	32555
2	2	Bengaluru	27-Aug-14	Silver	Bills	F	101738
3	3	Greater Mumbai	12-Apr-14	Signature	Bills	F	123424
4	4	Bengaluru	05-May-15	Gold	Bills	F	171574

Fig 2: Output after cleaning

From the fig 2, we got the desired dataset which will be useful for further analysis.

4. Design & Implementation

This section will discuss about the design and implementation of Map-Reduce algorithm. Here, we are going to use two mapper and two reducers which are linked with each other to answer our research question. This section is divided into two sub-sections. The first section will explain the design of each Job's mapper and reducer. The second subsection will discuss about the working of each Job's mapper and reducer.

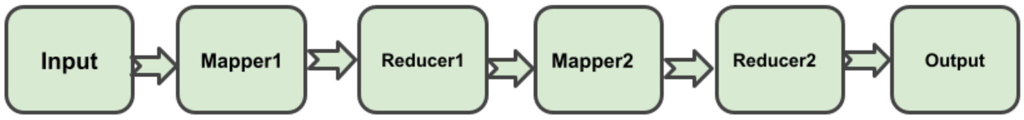
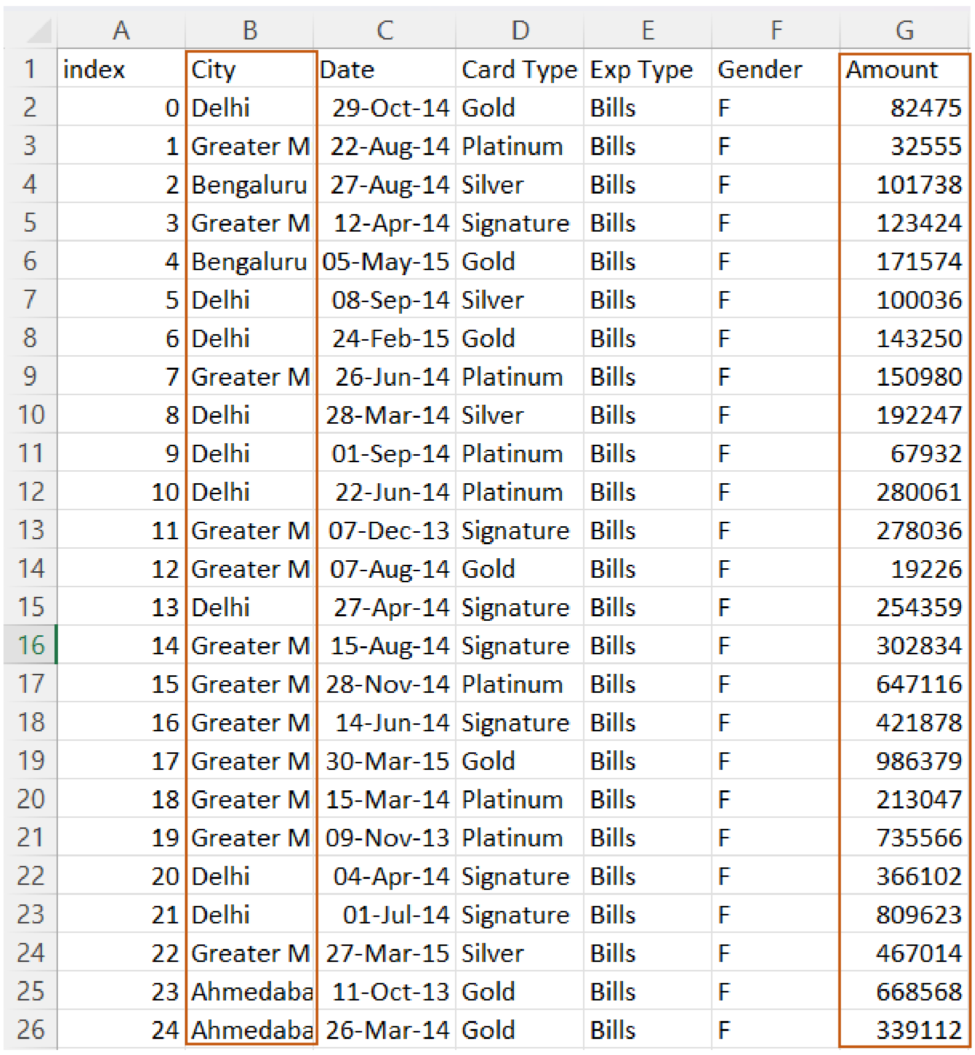


Fig 3: Overview of Map-Reduce Working

Here, the input will come from the file named: "CreditCard2.txt". The input will be first processed by Mapper-1 and that will give the output to Reducer-1 and it will further process it and emit the output which will be the input for Mapper- 2 and it will emit the output as it received. Finally, the Reducer-2 will further process the data and will emit the final output. It will answer finally answers our research question.



	A	B	C	D	E	F	G
1	index	City	Date	Card Type	Exp Type	Gender	Amount
2	0	Delhi	29-Oct-14	Gold	Bills	F	82475
3	1	Greater M	22-Aug-14	Platinum	Bills	F	32555
4	2	Bengaluru	27-Aug-14	Silver	Bills	F	101738
5	3	Greater M	12-Apr-14	Signature	Bills	F	123424
6	4	Bengaluru	05-May-15	Gold	Bills	F	171574
7	5	Delhi	08-Sep-14	Silver	Bills	F	100036
8	6	Delhi	24-Feb-15	Gold	Bills	F	143250
9	7	Greater M	26-Jun-14	Platinum	Bills	F	150980
10	8	Delhi	28-Mar-14	Silver	Bills	F	192247
11	9	Delhi	01-Sep-14	Platinum	Bills	F	67932
12	10	Delhi	22-Jun-14	Platinum	Bills	F	280061
13	11	Greater M	07-Dec-13	Signature	Bills	F	278036
14	12	Greater M	07-Aug-14	Gold	Bills	F	19226
15	13	Delhi	27-Apr-14	Signature	Bills	F	254359
16	14	Greater M	15-Aug-14	Signature	Bills	F	302834
17	15	Greater M	28-Nov-14	Platinum	Bills	F	647116
18	16	Greater M	14-Jun-14	Signature	Bills	F	421878
19	17	Greater M	30-Mar-15	Gold	Bills	F	986379
20	18	Greater M	15-Mar-14	Platinum	Bills	F	213047
21	19	Greater M	09-Nov-13	Platinum	Bills	F	735566
22	20	Delhi	04-Apr-14	Signature	Bills	F	366102
23	21	Delhi	01-Jul-14	Signature	Bills	F	809623
24	22	Greater M	27-Mar-15	Silver	Bills	F	467014
25	23	Ahmedaba	11-Oct-13	Gold	Bills	F	668568
26	24	Ahmedaba	26-Mar-14	Gold	Bills	F	339112

Fig 4: Credit Card Spending dataset

This is how the dataset looks like. From this table, we are going to use the features named: City & Amount (Highlighted) to answer our research question.

a. Design of each Mapper and Reducer

i. Design of Job-1

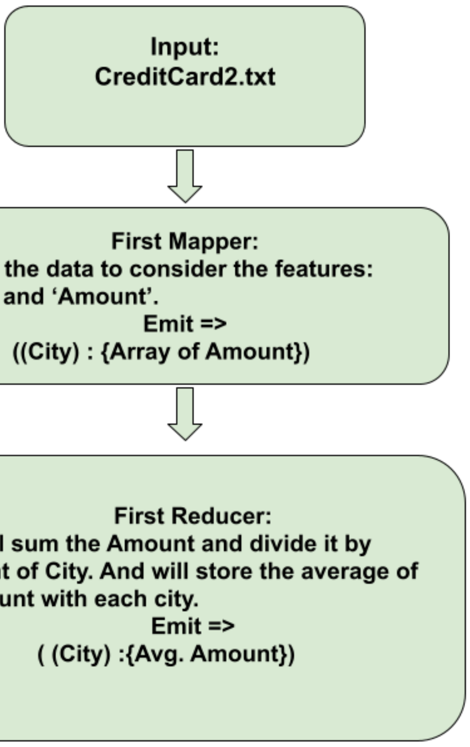

Input:
CreditCard2.txt
First Mapper:
Filter the data to consider the features:
'City' and 'Amount'.
Emit =>
((City) : {Array of Amount})
First Reducer:
It will sum the Amount and divide it by count of City. And will store the average of Amount with each city.
Emit =>
((City) :{Avg. Amount})

Fig 4: Flow diagram for Job-1

The given flow diagram represents the first stage of the MapReduce program, which calculates the average amount spent for each city.

Input: The input file, "CreditCard2.txt", contains the raw transaction data with various features, such as city, date, card type, expense type, gender, and transaction amount.

First Mapper: The first mapper's task is to filter the input data by extracting the relevant features: 'City' and 'Amount'. It reads each line of the input file and emits key-value pairs where the key is the city name, and the value is an array containing the transaction amounts associated with that city.

Emit:

(City): {Array of Amount}

First Reducer: The first reducer receives the key-value pairs generated by the first mapper, where the key is the city name and the values are arrays of transaction amounts for each city. The reducer computes the sum of transaction amounts and divides it by the count of occurrences of each city to calculate the average transaction amount. The reducer then emits a key-value pair where the key is the city name, and the value is the calculated average amount.

Emit:

(City): {Avg. Amount}

This flow diagram explains the process of calculating the average amount spent by each city using a mapper and a reducer. The mapper filters the input data by considering only the 'City' and 'Amount' features, and the reducer computes the average amount spent per city using the mapper's output.

ii. Design of Job2

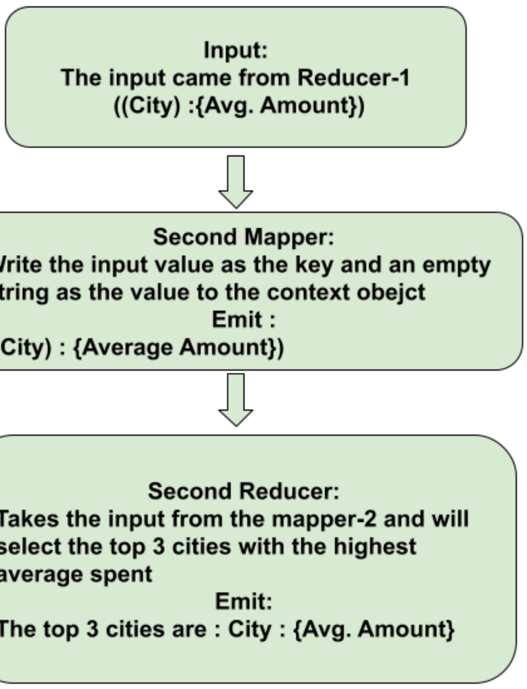

Input:
The input came from Reducer-1
((City) :{Avg. Amount})
Second Mapper:
Write the input value as the key and an empty string as the value to the context obejct
Emit :
((City) : {Average Amount})
Second Reducer:
Takes the input from the mapper-2 and will select the top 3 cities with the highest average spent
Emit:
The top 3 cities are : City : {Avg. Amount}

Fig 5: Flow diagram for Job-2

The flow diagram represents the second stage of the MapReduce program, which identifies the top 3 cities with the highest average spent.

Input:

The input for this stage comes from Reducer-1's output, which contains key-value pairs with the city name as the key and the average transaction amount as the value.

(City): {Avg. Amount}

Second Mapper:

The second mapper's task is to prepare the input data for the second reducer. It reads each line of the input and directly writes the input value as the key and an empty string as the value to the context object, without performing any calculations or transformations.

Emit: (City): {Average Amount}

Second Reducer:

The second reducer receives the key-value pairs generated by the second mapper, where the key contains the city name and the corresponding average transaction amount. The reducer's goal is to select the top 3 cities with the highest average spent. To accomplish this, it maintains a TreeMap, which stores the average spent as the key and the city name as the value. The TreeMap is sorted in ascending order of average spent, so the reducer can easily maintain only the top 3 cities by removing the entry with the lowest average whenever the TreeMap size exceeds 3.

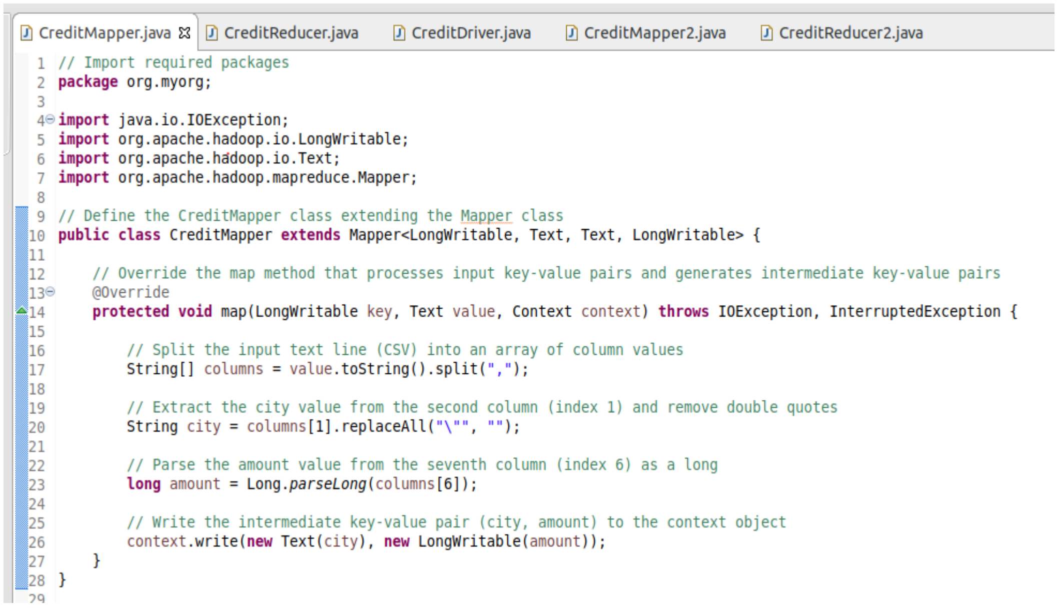
In the cleanup stage, the reducer constructs a formatted output string that includes the top 3 cities and their average spent. This string is emitted as the final result.

Emit: The top 3 cities are: City: {Avg. Amount}

This flow diagram explains the process of identifying the top 3 cities with the highest average spent using a mapper and a reducer. The mapper prepares the input data for the reducer by emitting the same key-value pairs it receives, while the reducer selects the top 3 cities using the TreeMap data structure and emits the result.

b. Internal working of each Mapper and Reducer

i. Working and code Job- 1

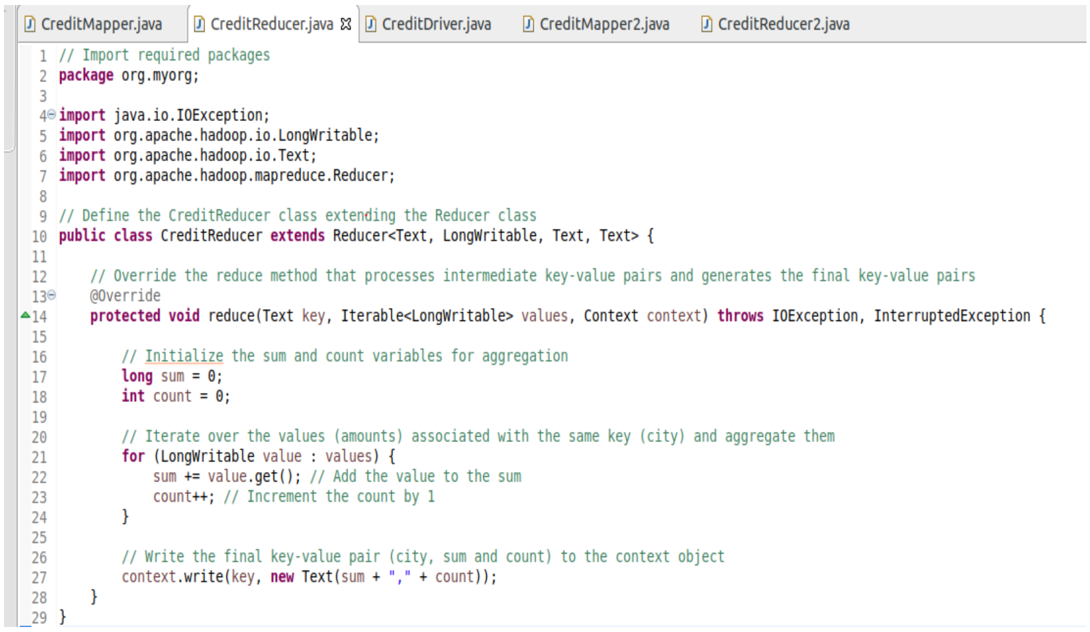


```java
// Import required packages
package org.myorg;

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

// Define the CreditMapper class extending the Mapper class
public class CreditMapper extends Mapper<LongWritable, Text, Text, LongWritable> {

    // Override the map method that processes input key-value pairs and generates intermediate key-value pairs
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

        // Split the input text line (CSV) into an array of column values
        String[] columns = value.toString().split(",");

        // Extract the city value from the second column (index 1) and remove double quotes
        String city = columns[1].replaceAll("\"", "");

        // Parse the amount value from the seventh column (index 6) as a long
        long amount = Long.parseLong(columns[6]);

        // Write the intermediate key-value pair (city, amount) to the context object
        context.write(new Text(city), new LongWritable(amount));
    }
}
```

Fig 6: Mapper-1 in Job-1

Here, Fig. 6 is a snippet of the Mapper-1 in Job-1, the CreditMapper class, a crucial component of the MapReduce programme that filters the input data and extracts pertinent features to compute the average amount spent per city. The map method processes input key-value pairs and generates intermediate key-value pairs. The class extends the Hadoop Mapper class and defines its functionality through the map method.

In the map method, a CSV-formatted input text line is split into an array of column values. The city name is extracted from the second column (index 1), removing any double quotes. Next, the transaction amount is extracted from the seventh column as a long integer (index 6). The intermediate key-value pair containing the city name as the key and the transaction amount as the value is finally written to the context object. The reducer will use these intermediate data to determine the average transaction amount for each city.



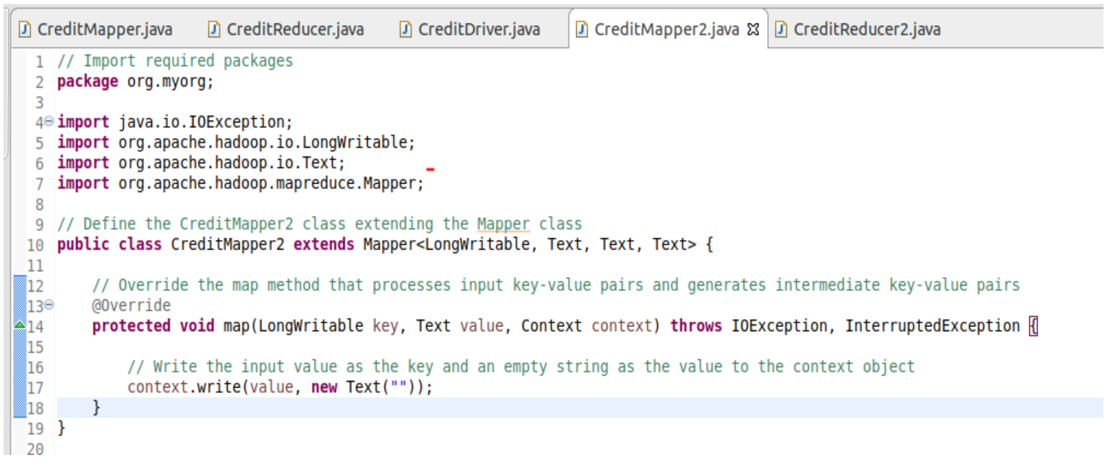
```java
// Import required packages
package org.myorg;

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

// Define the CreditReducer class extending the Reducer class
public class CreditReducer extends Reducer<Text, LongWritable, Text, Text> {

    // Override the reduce method that processes intermediate key-value pairs and generates the final key-value pairs
    @Override
    protected void reduce(Text key, Iterable<LongWritable> values, Context context) throws IOException, InterruptedException {

        // Initialize the sum and count variables for aggregation
        long sum = 0;
        int count = 0;

        // Iterate over the values (amounts) associated with the same key (city) and aggregate them
        for (LongWritable value : values) {
            sum += value.get(); // Add the value to the sum
            count++; // Increment the count by 1
        }

        // Write the final key-value pair (city, sum and count) to the context object
        context.write(key, new Text(sum + "," + count));
    }
}
```

Fig 7: Reducer-1 in Job-1

Figure 7 demonstrates the CreditReducer class, an essential component of the MapReduce programme responsible for aggregating the intermediate data generated by the mapper and calculating the total amount spent per city. The class extends the Reducer class in Hadoop and defines its functionality through the reduce method, which processes intermediate key-value pairs and generates final key-value pairs.

The sum and count variables are initialised within the reduce method to aggregate the transaction amounts associated with each city. The method iterates over the values (transaction amounts) associated with the same key (city name) and calculates their sum while maintaining a count. After processing all the values for a particular city, the context object is updated with the final key-value pair. This key-value pair is composed of the city's name as the key and a string containing the sum and the count, separated by a comma. The next phase of the MapReduce programme will use these aggregated data to determine the average transaction amount per city.

ii. Working and code Job- 2



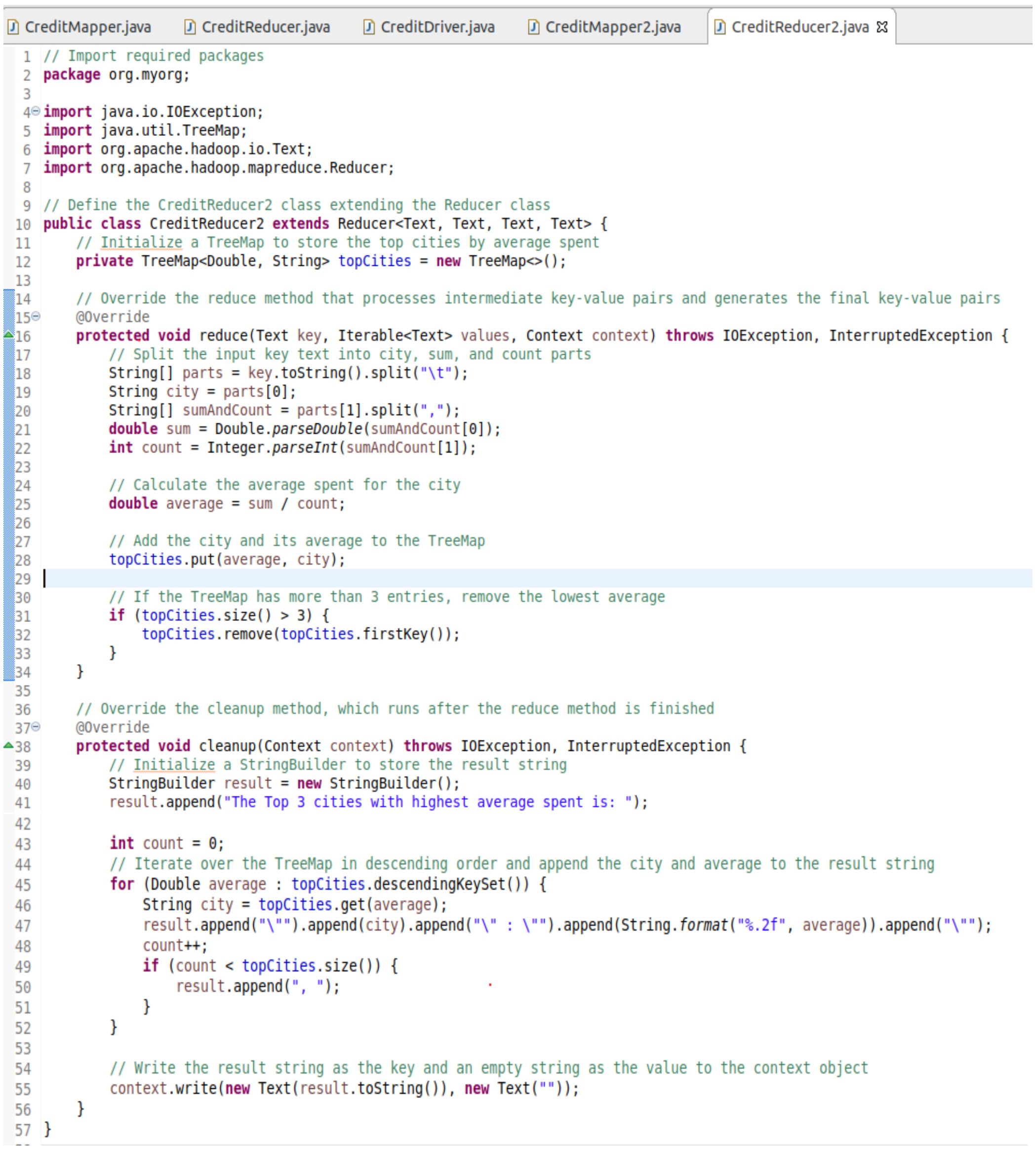
CreditMapper.java | CreditReducer.java | CreditDriver.java | CreditMapper2.java | CreditReducer2.java

```java
// Import required packages
package org.myorg;

import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

// Define the CreditMapper2 class extending the Mapper class
public class CreditMapper2 extends Mapper<LongWritable, Text, Text, Text> {

    // Override the map method that processes input key-value pairs and generates intermediate key-value pairs
    @Override
    protected void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {

        // Write the input value as the key and an empty string as the value to the context object
        context.write(value, new Text(""));
    }
}

```

Fig 8: Mapper-2 in Job-2

Figure 8 demonstrates the CreditMapper2 class, which is a component of the second MapReduce job (City Wise Average) within the overall programme. This mapper's purpose is to pass the input data from the first job's reducer (CreditReducer) to the second job's reducer (DebitReducer). The input data consists of the city name, the sum of transaction amounts, and the transaction count (CreditReducer2). CreditMapper2 is an extension of the Mapper class that provides simple pass-through functionality.

CreditMapper2's map method is overridden to process the input key-value pairs. CreditMapper2 does not perform any complex data manipulation or transformation, unlike its predecessor. It simply writes the input value to the context object as the key and an empty string as the value. This minimal data processing step prepares the intermediate key-value pairs for the second reducer, CreditReducer2, which determines the average transaction amount for each city and selects the top three cities with the highest average spending.



```java
// Import required packages
package org.myorg;

import java.io.IOException;
import java.util.TreeMap;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

// Define the CreditReducer2 class extending the Reducer class
public class CreditReducer2 extends Reducer<Text, Text, Text, Text> {
    // Initialize a TreeMap to store the top cities by average spent
    private TreeMap<Double, String> topCities = new TreeMap<>();

    // Override the reduce method that processes intermediate key-value pairs and generates the final key-value pairs
    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        // Split the input key text into city, sum, and count parts
        String[] parts = key.toString().split("\t");
        String city = parts[0];
        String[] sumAndCount = parts[1].split(",");
        double sum = Double.parseDouble(sumAndCount[0]);
        int count = Integer.parseInt(sumAndCount[1]);

        // Calculate the average spent for the city
        double average = sum / count;

        // Add the city and its average to the TreeMap
        topCities.put(average, city);

        // If the TreeMap has more than 3 entries, remove the lowest average
        if (topCities.size() > 3) {
            topCities.remove(topCities.firstKey());
        }
    }

    // Override the cleanup method, which runs after the reduce method is finished
    @Override
    protected void cleanup(Context context) throws IOException, InterruptedException {
        // Initialize a StringBuilder to store the result string
        StringBuilder result = new StringBuilder();
        result.append("The Top 3 cities with highest average spent is: ");

        int count = 0;
        // Iterate over the TreeMap in descending order and append the city and average to the result string
        for (Double average : topCities.descendingKeySet()) {
            String city = topCities.get(average);
            result.append("\"").append(city).append("\" : \"").append(String.format("%.2f", average)).append("\"");
            count++;
            if (count < topCities.size()) {
                result.append(", ");
            }
        }

        // Write the result string as the key and an empty string as the value to the context object
        context.write(new Text(result.toString()), new Text(""));
    }
}
```

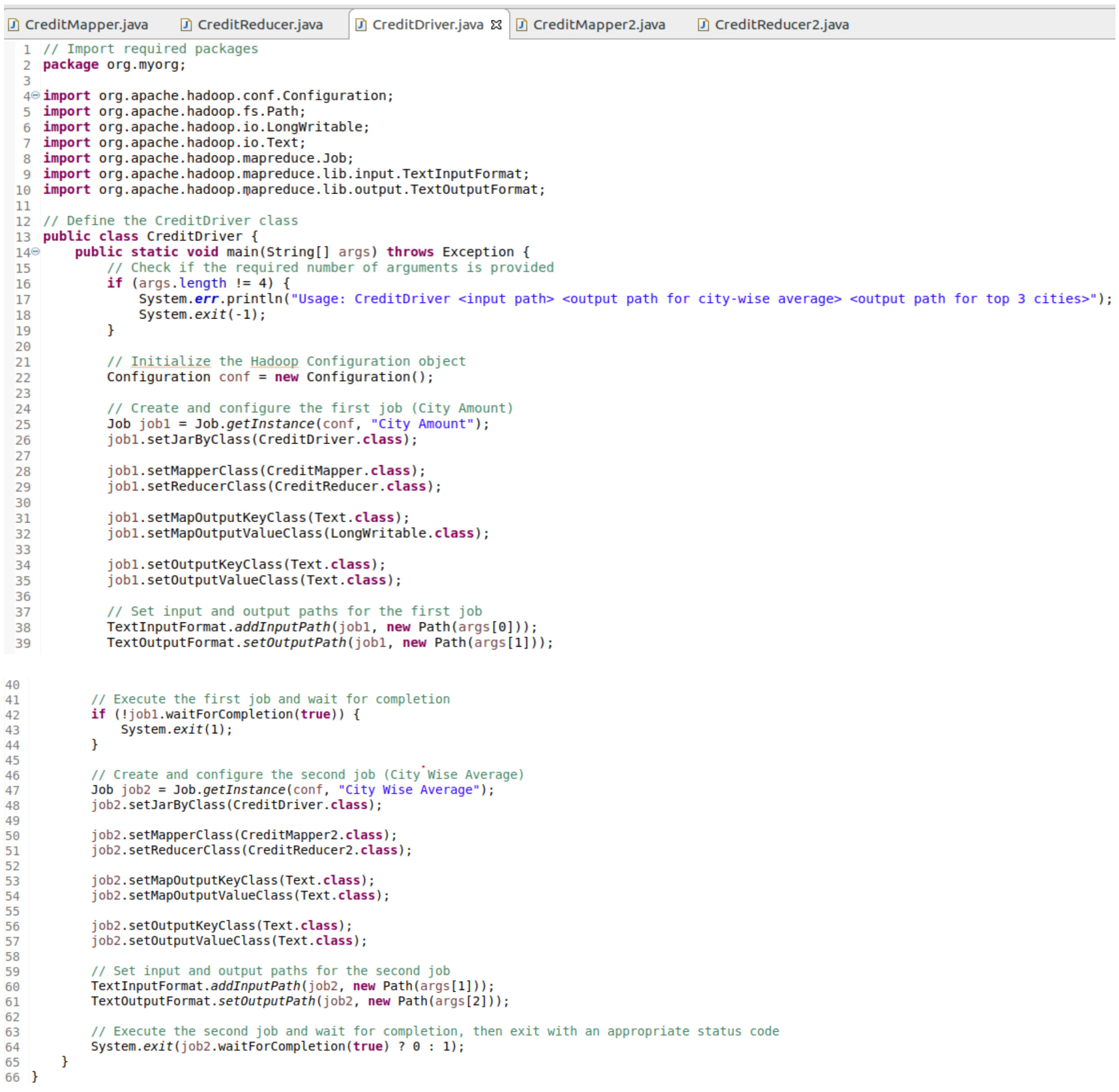
Fig 9: Reducer-2 in Job-2

The CreditReducer2 class, which is the reducer component of the second MapReduce job (City Wise Average) in the programme, is depicted in Fig. 9. This reducer utilises the CreditMapper2 intermediate key-value pairs to determine the top three cities with the highest average transaction amounts. The CreditReducer2 class extends the Reducer class and provides the ability to process input data and generate the desired output.

The reduce method is overridden to process the input key-value pairs, calculate each city's average spending, and store the top three cities in a TreeMap. The cleanup method, which runs after the reduce method has completed, is also overridden. This method iterates over the TreeMap in descending order, constructing a string containing the top three cities and their respective average transaction amounts. The final step involves writing the result string to the context object as the output key and an empty string as the output value. This result is then written to the output path specified, providing a clear and concise response to the research question by identifying the top three cities with the highest average transaction amounts.

5. Results & Evaluation

This section discusses about the working of driver class, which maintains the whole map-reduce program in a sequence. Moreover, it discusses about the results we got from the Reducer-1 and Reducer-2 and evaluation of those results and will answer our research question: **“What is the average amount spent by each city, and which are the top 3 cities that have the highest average spent?”**

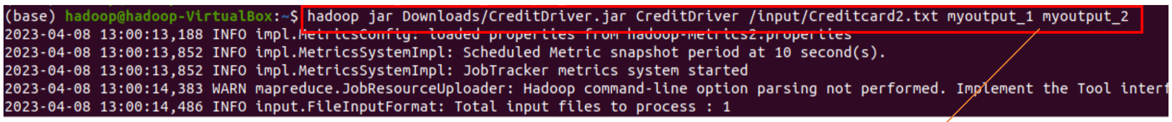


```java
// Import required packages
package org.myorg;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

// Define the CreditDriver class
public class CreditDriver {
    public static void main(String[] args) throws Exception {
        // Check if the required number of arguments is provided
        if (args.length != 4) {
            System.err.println("Usage: CreditDriver <input path> <output path for city-wise average> <output path for top 3 cities>");
            System.exit(-1);
        }

        // Initialize the Hadoop Configuration object
        Configuration conf = new Configuration();

        // Create and configure the first job (City Amount)
        Job job1 = Job.getInstance(conf, "City Amount");
        job1.setJarByClass(CreditDriver.class);

        job1.setMapperClass(CreditMapper.class);
        job1.setReducerClass(CreditReducer.class);

        job1.setMapOutputKeyClass(Text.class);
        job1.setMapOutputValueClass(LongWritable.class);

        job1.setOutputKeyClass(Text.class);
        job1.setOutputValueClass(Text.class);

        // Set input and output paths for the first job
        TextInputFormat.addInputPath(job1, new Path(args[0]));
        TextOutputFormat.setOutputPath(job1, new Path(args[1]));

        // Execute the first job and wait for completion
        if (!job1.waitForCompletion(true)) {
            System.exit(1);
        }

        // Create and configure the second job (City Wise Average)
        Job job2 = Job.getInstance(conf, "City Wise Average");
        job2.setJarByClass(CreditDriver.class);

        job2.setMapperClass(CreditMapper2.class);
        job2.setReducerClass(CreditReducer2.class);

        job2.setMapOutputKeyClass(Text.class);
        job2.setMapOutputValueClass(Text.class);

        job2.setOutputKeyClass(Text.class);
        job2.setOutputValueClass(Text.class);

        // Set input and output paths for the second job
        TextInputFormat.addInputPath(job2, new Path(args[1]));
        TextOutputFormat.setOutputPath(job2, new Path(args[2]));

        // Execute the second job and wait for completion, then exit with an appropriate status code
        System.exit(job2.waitForCompletion(true) ? 0 : 1);
    }
}
```

Fig 10: Driver Class

Figure 10 demonstrates the CreditDriver class, which serves as the driver programme coordinating the two-stage MapReduce program's execution. This program's primary objective is to calculate the average amount spent in each city and identify the three cities with the highest average expenditures. The CreditDriver class verifies the validity of the input parameters before initialising the Hadoop Configuration object.

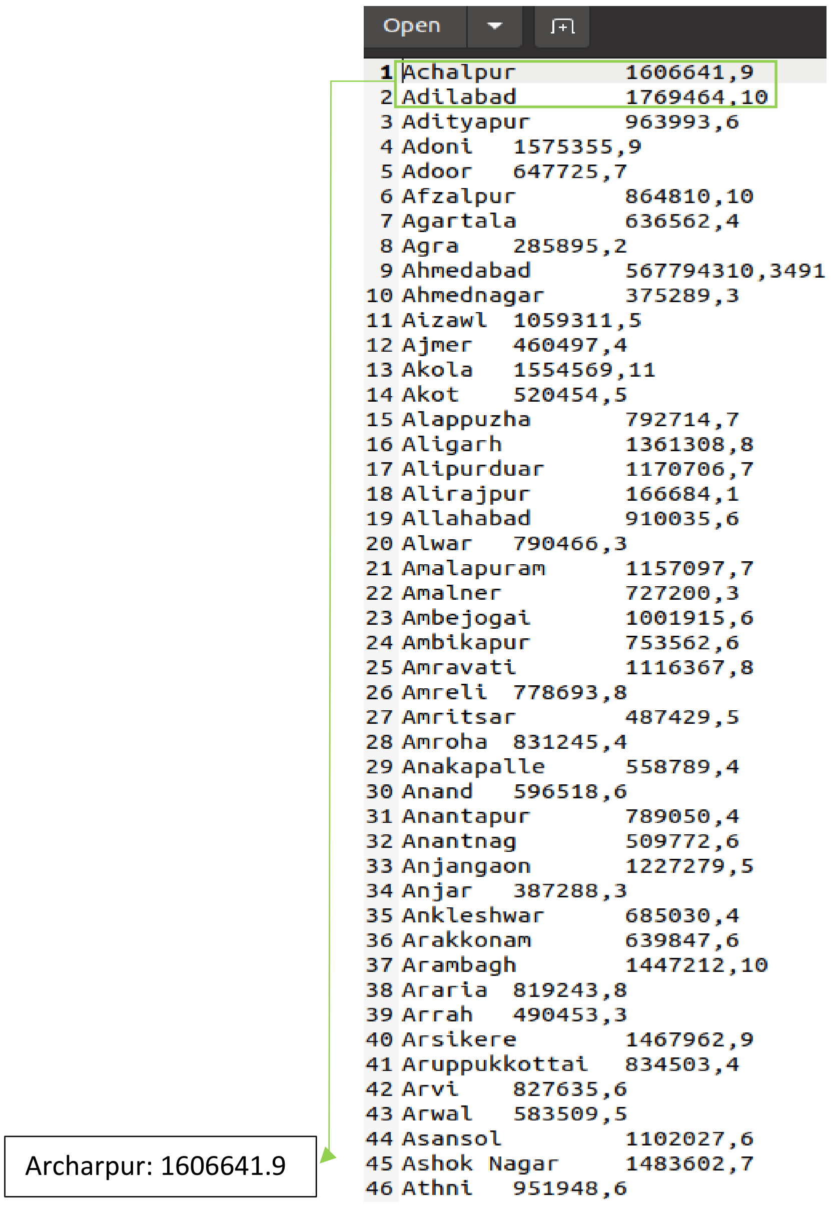
The CreditDriver class creates two distinct MapReduce jobs, namely job1 (City Amount) and job2 (All Other Amounts) (City Wise Average). using the CreditMapper and CreditReducer classes, job1 is responsible for aggregating the transaction amounts by city. Using command-line arguments, the input and output paths for job1 are specified. After job1 is completed, the programme moves on to job2. job2 computes the average transaction amount for each city and identifies the three cities with the highest average expenditure. It accomplishes this with the CreditMapper2 and CreditReducer2 classes. Additionally, the output paths for job2 are specified via command-line arguments. The programme awaits the completion of both jobs before exiting with the appropriate status code based on their success or failure.



```
(base) hadoop@hadoop-VirtualBox:~$ hadoop jar Downloads/CreditDriver.jar CreditDriver /input/Creditcard2.txt myoutput_1 myoutput_2
2023-04-08 13:00:13,188 INFO impl.MetricsConfig: loaded properties from hadoop-metrics2.properties
2023-04-08 13:00:13,852 INFO impl.MetricsSystemImpl: Scheduled Metric snapshot period at 10 second(s).
2023-04-08 13:00:13,852 INFO impl.MetricsSystemImpl: JobTracker metrics system started
2023-04-08 13:00:14,383 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interf
2023-04-08 13:00:14,486 INFO input.FileInputFormat: Total input files to process : 1
```

Fig 11: Executing Jar file

Will store the output in two different directories: “myoutput_1” & “myoutput_2”

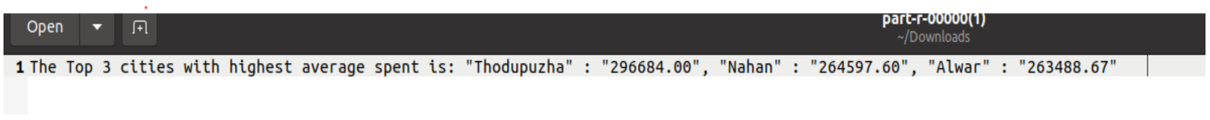


```
Achalpur        1606641,9
Adilabad        1769464,10
Adityapur       963993,6
Adoni   1575355,9
Adoor   647725,7
Afzalpur        864810,10
Agartala        636562,4
Agra    285895,2
Ahmedabad       567794310,3491
Ahmednagar      375289,3
Aizawl  1059311,5
Ajmer   460497,4
Akola   1554569,11
Akot    520454,5
Alappuzha       792714,7
Aligarh         1361308,8
Alipurduar      1170706,7
Alirajpur       166684,1
Allahabad       910035,6
Alwar   790466,3
Amalapuram      1157097,7
Amalner         727200,3
Ambejogai       1001915,6
Ambikapur       753562,6
Amravati        1116367,8
Amreli  778693,8
Amritsar        487429,5
Amroha  831245,4
Anakapalle      558789,4
Anand   596518,6
Anantapur       789050,4
Anantnag        509772,6
Anjangaon       1227279,5
Anjar   387288,3
Ankleshwar      685030,4
Arakkonam       639847,6
Arambagh        1447212,10
Araria  819243,8
Arrah   490453,3
Arsikere        1467962,9
Aruppukkottai   834503,4
Arvi    827635,6
Arwal   583509,5
Asansol         1102027,6
Ashok Nagar     1483602,7
Athni   951948,6
```

Fig 12: Output from Job-1

Here, the Figure 12 shows the screenshot of the output we got from the execution of Job-1. From this output we can answer the first half of our research question: “**What is the average amount spent by each city?**” The output is in the format of {key = City_name : value = Average amount spent by the people in the city}

For example, Archarpur city has average spending from the credit card is around 1606641.9 Rs in India. Similarly, we can see the average spent in each city in the figure 12.



```
The Top 3 cities with highest average spent is: "Thodupuzha" : "296684.00", "Nahan" : "264597.60", "Alwar" : "263488.67"
```

Fig 12: Output from Job-2

Figure 12 represents the output of Job-2. Which answers our remaining half of the research question: “**Which are the top 3 cities that have the highest average spent?**”. Therefore, the answer we got from the last output is that city named Thodupuzha has the people with highest amount spent with credit card that is 296684 Rs in India. Followed by Nahan, spent around 264597 Rs and the third city is Alwar in which the average spent is 263488.67 Rs.

6. Conclusion

This project has demonstrated the effectiveness of High-Performance Computational Infrastructure, with a particular emphasis on the Hadoop MapReduce framework, as a tool for processing and extracting valuable insights from massive datasets. By designing and implementing a multi-stage MapReduce pipeline, I was able to identify the top three cities in India with the highest average transaction amounts, revealing significant spending patterns throughout the nation.