Name: Drishti Harshit Doshi

Student ID: 2203898

Module Code: CS5810

Module Title: High-Performance Computational Infrastructures

Academic Year: 2022-2023

Assignment Title: Coursework Assignment using MapReduce

Section 1:

Introduction:

Big Data refers to the collection of datasets that are large and complex and which cannot be stored or processed using traditional data processing systems. Organizations are realizing the importance of big data analytics for getting useful business insights. They are using large datasets to explore hidden patterns, find correlations, understand customer preferences, and other useful information. Analyzing big data helps the organization generate more revenue, formulate effective marketing strategies, provide better customer service by preventing customer churn, and compete well with its opponents.

The traditional approach of using a **Relational Database Management system** has certain **limitations** which affect the business of the organization. Some of them include:

- Storing a large amount of data generated by companies every second.
- Storing heterogeneous data from the website, API, social media, text data, etc.
- Accessing and processing speed to handle big data is less.

In the year 2008, **Yahoo released Hadoop** as an open-source project to Apache Software Foundation.
Hadoop is a framework for storing Big-Data sets in a distributed environment that can be processed parallelly.

Hadoop has the following main components:

- Hadoop Distributed File System (HDFS): A block-structured file system where each file is divided into blocks of a pre-determined size. These blocks are stored across a cluster of one or several machines. HDFS runs on standard or low-end hardware. It has high fault tolerance and handles large datasets effectively. HDFS is preferable for low-latency access.
- Yet Another Resource Negotiator (YARN): It is a Hadoop cluster resource management system. It Performs all processing activities by allocating resources and scheduling tasks.
- Map-reduce: It is a programming model for data processing. MapReduce has a framework that allows the programs to be executed on parallel computers. The map task (mapper) takes input data and converts it into a dataset computed in the form of key-value pairs. The reduce tasks (reducer) aggregates and provides the output.

Problem Description:

Every day, supermarkets sell millions of products to millions of people. In developed countries, people are dependent on supermarkets for their daily necessities. In addition to product pricing, providing good customer service and allowances are additional factors for competing with other stores. During this period, the retail sector is heavily facing crises due to global economic recession and financial instability. With the rise in technology and users, it is becoming increasingly difficult for retailers to meet customers' expectations. E-commerce, entrepreneurship, innovations, increase in private businesses, etc., all these factors have caused a massive increase in competition.

Use case: The US retail store aims to improve customer service as a strategy to prevent customer churn and retain existing customers. Big-data technology can be used here to extract insights into customer details that contribute to higher sales of the store and lower sales of the store. Accordingly, the Supermarket stores will focus on the strategies that will increase customer loyalty i.e., by providing a personalized experience, and promotions that will help to achieve customer retention.

Research Question:

What is the total amount spent by each customer on their purchases, combined?

<u>Dataset:</u>

The US Superstore Sales data has been used for the analysis. The dataset has been collected from Kaggle.
The dataset has two files in .xls format:

a) **Customer data:** Data of the customers buying the products.

Name	**Description**	**Datatype**
Customer ID	A unique ID assigned to each customer	String
Customer Name	Name of customers	String
Segment	Consumer or Co-operate sector	String
Country	Country of the Superstore outlet	String
City	City of the Superstore outlet	String
State	State of the Superstore outlet	String
Postal Code	Postal code of the Superstore outlet	String

b) **Transaction data**: Transaction details of the superstore.

Name	**Description**	**Datatype**
Transaction ID	Transaction details of orders	String
Order Date	Date of transaction	Ordinal
Customer ID	A unique ID assigned to each customer	String
Sales	Sales of each transaction	Numerical

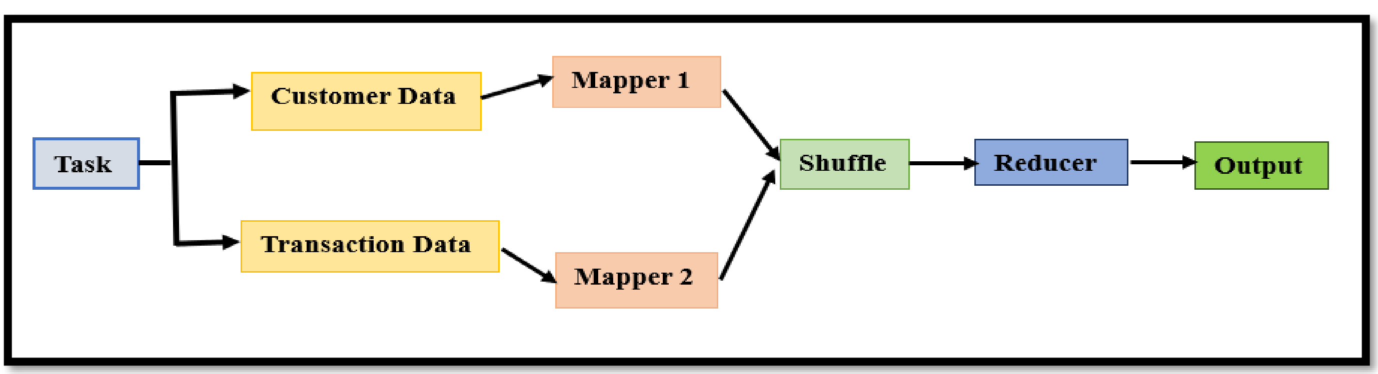
Desired Output:

Customer Name	Total number of transactions	Total Sales

The Hadoop MapReduce framework can perform Joins by combining two or more database tables on foreign keys. Customer ID is the common variable in both data files. **The Customer ID** (foreign key) can be used to **combine** both datasets and get the total number of transactions and total sales for each customer.
There are two types of Joins in MapReduce:

a) Map-side Join: The join operation is performed by the mapper. It is convenient when one of the tables is small (Distributed cache) and not recommended for larger tables.
b) Reduce-side Join: The join operation is performed by the reducer. They are simpler and most widely used joins because Hadoop sends identical keys to the same reducer.

The given research question: What is the total amount spent by each customer on their purchases, combined?
will be answered using **Reduce-Side Joins** of Hadoop MapReduce.



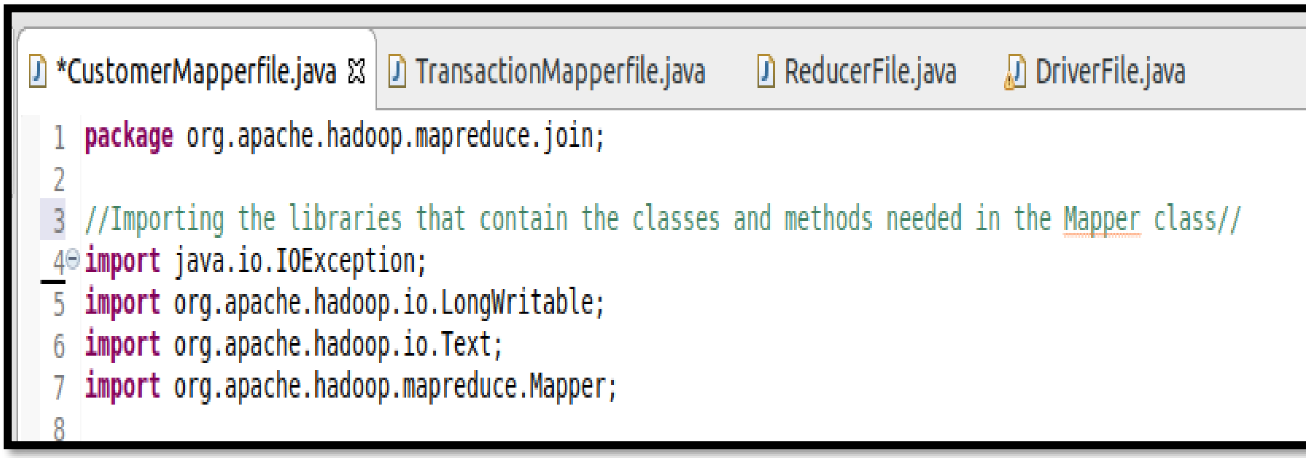
Section 2:

Design & Implementation:

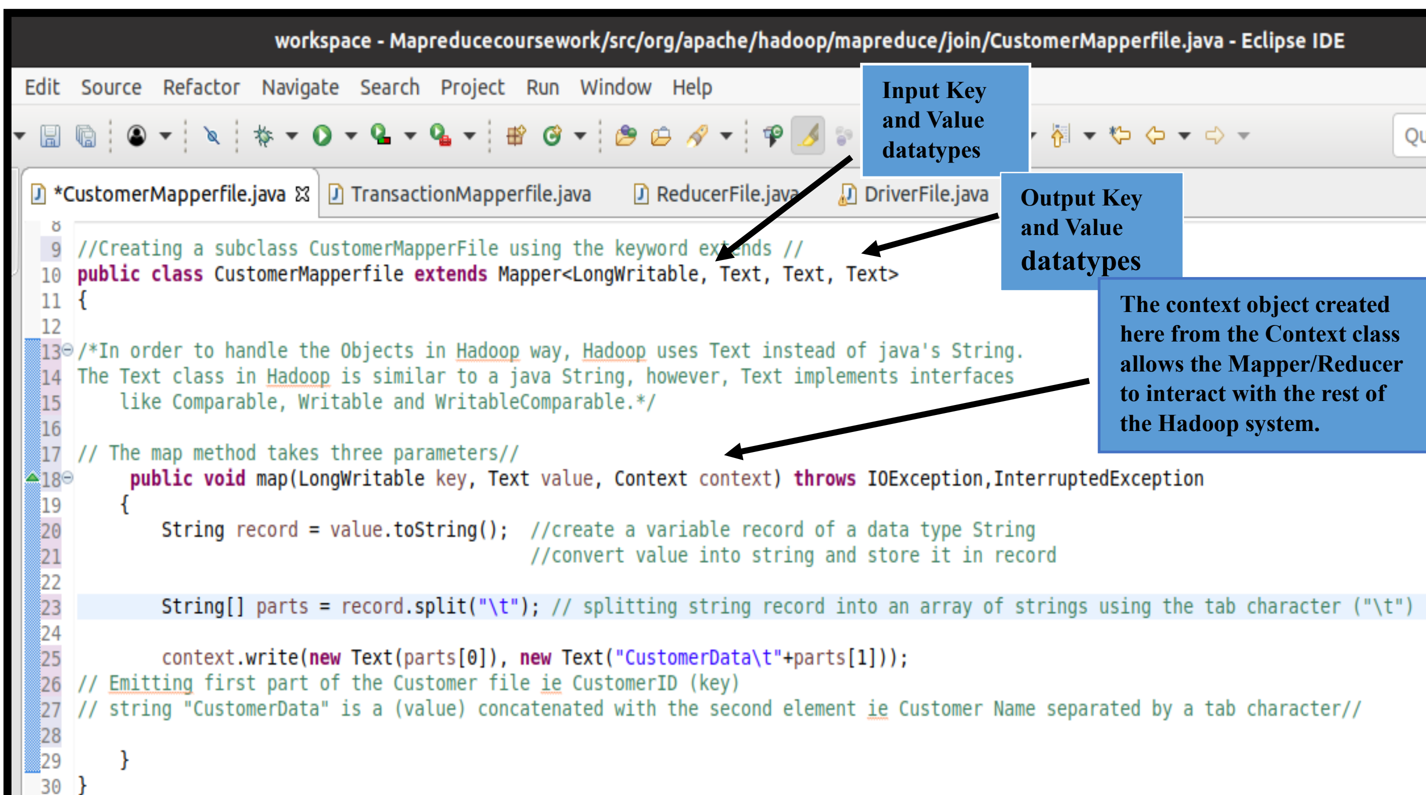
Reduce-side joins have been implemented to find the research answer using MapReduce in the Hadoop framework.
The research question asks to find the total amount of purchases made by each customer, by combining their multiple transactions and assigning the total number of sales, and the total number of transactions for each customer as the final output.Firstly, using the Virtual Machine which is provided by the Computer Science department at Brunel University London. Using Eclipse IDE for writing the JAVA MapReduce code. Steps for opening the JAVA project: Open Eclipse, open a new JAVA project, export external libraries, and create four classes in the given order- CustomerMapperfile, TransactionMapperfile, ReducerFile, and DriverFile using package: *org.apache.hadoop.mapeduce.join*

1) **Customer Mapper:**

Customer Mapperfile:




```java
package org.apache.hadoop.mapreduce.join;

//Importing the libraries that contain the classes and methods needed in the Mapper class//
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
```



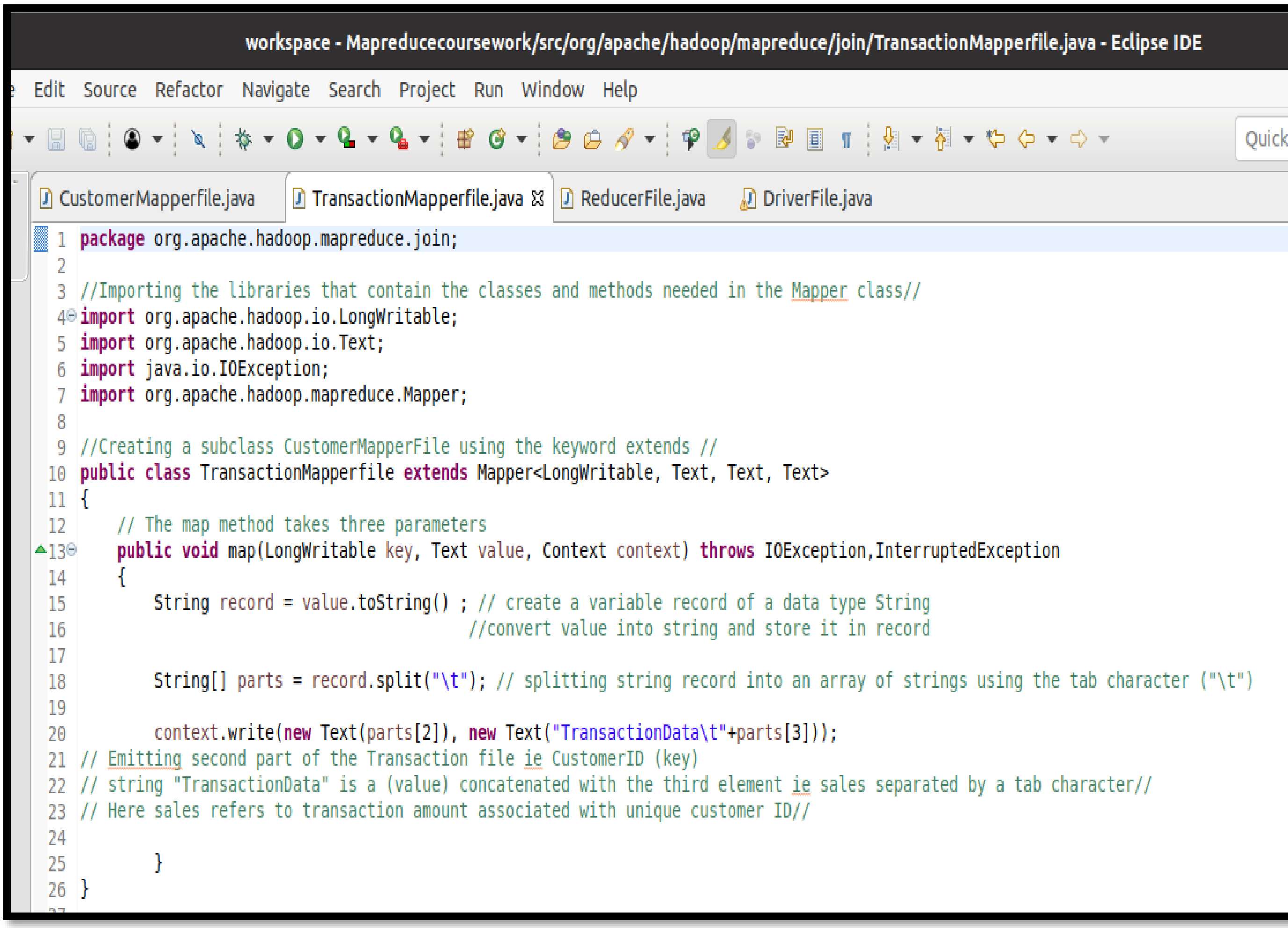

```java
//Creating a subclass CustomerMapperFile using the keyword extends //
public class CustomerMapperfile extends Mapper<LongWritable, Text, Text, Text>
{

/*In order to handle the Objects in Hadoop way, Hadoop uses Text instead of java's String.
The Text class in Hadoop is similar to a java String, however, Text implements interfaces
    like Comparable, Writable and WritableComparable.*/

// The map method takes three parameters//
    public void map(LongWritable key, Text value, Context context) throws IOException,InterruptedException
    {
        String record = value.toString();  //create a variable record of a data type String
                                           //convert value into string and store it in record

        String[] parts = record.split("\t"); // splitting string record into an array of strings using the tab character ("\t")

        context.write(new Text(parts[0]), new Text("CustomerData\t"+parts[1]));
// Emitting first part of the Customer file ie CustomerID (key)
// string "CustomerData" is a (value) concatenated with the second element ie Customer Name separated by a tab character//

    }
}
```

The Customer Mapperfile: Importing necessary libraries. Creating a new class named CustomerMapperfile. Using **extend** to assign data types. The **input key** is **LongWritable, input value** is **Text**. The **output key** is **Text** and the **output value** is **Text.** One tuple is taken as an input at a time i.e., one input = one tuple. Then each word in that tuple is tokenized. The Customer ID along with Customer Name is retrieved.The Customer ID will be the Mapper Key. The tag ”CustomerData” is added to assign each input tuple to CustomerMapperfile.
End output of CustomerMapperfile: Key-Value Pair: [Customer ID, Customer Name]

2) Transaction Mapper:

It will have same steps above, but the only difference will be in the fetching process. Sales will be taken instead of Customer Name and TransactionData will be taken as a tag to assign each input tuple to TransactionMapperfile.
End output of Transaction Mapper: [Customer ID, Sales**]**

Transaction Mapperfile:



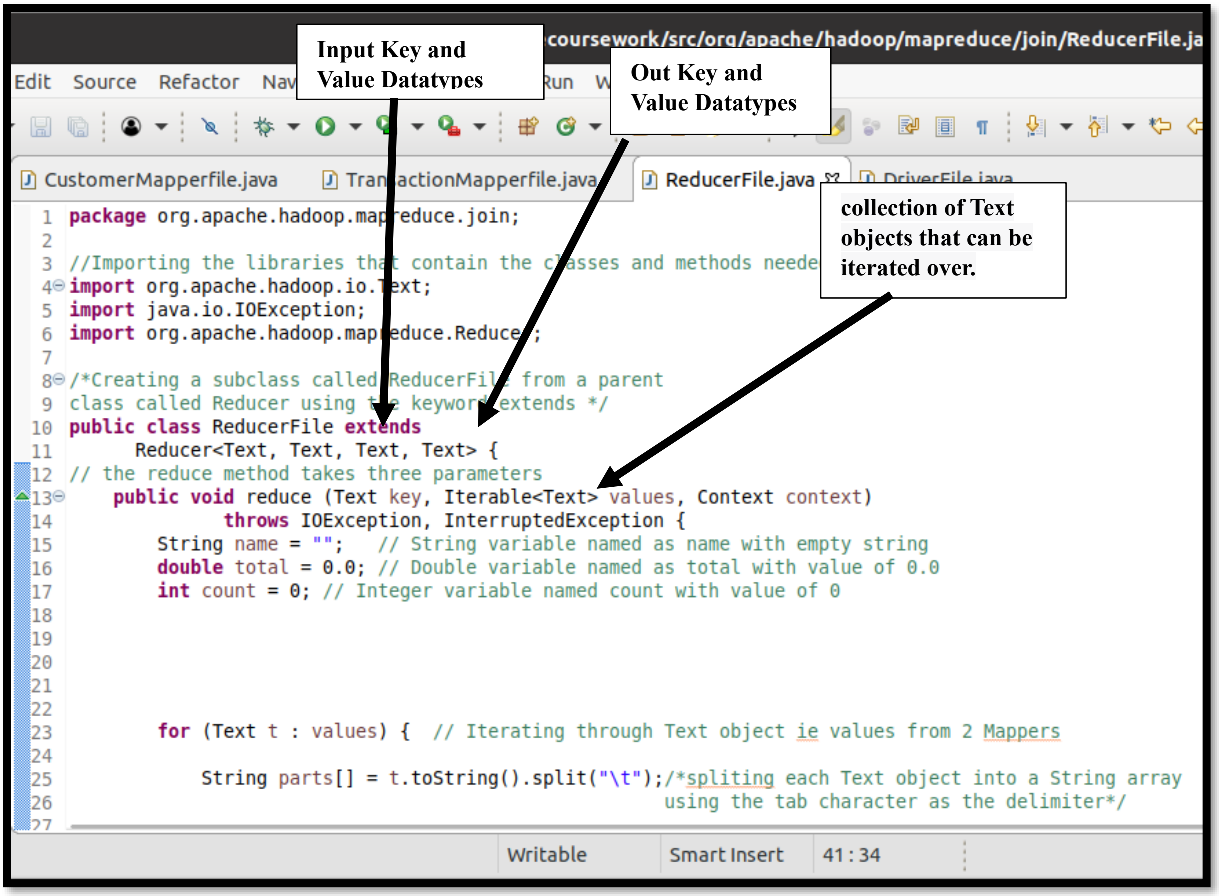

```java
package org.apache.hadoop.mapreduce.join;

//Importing the libraries that contain the classes and methods needed in the Mapper class//
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import java.io.IOException;
import org.apache.hadoop.mapreduce.Mapper;

//Creating a subclass CustomerMapperFile using the keyword extends //
public class TransactionMapperfile extends Mapper<LongWritable, Text, Text, Text>
{
    // The map method takes three parameters
    public void map(LongWritable key, Text value, Context context) throws IOException,InterruptedException
    {
        String record = value.toString() ; // create a variable record of a data type String
                                          //convert value into string and store it in record

        String[] parts = record.split("\t"); // splitting string record into an array of strings using the tab character ("\t")

        context.write(new Text(parts[2]), new Text("TransactionData\t"+parts[3]));
// Emitting second part of the Transaction file ie CustomerID (key)
// string "TransactionData" is a (value) concatenated with the third element ie sales separated by a tab character//
// Here sales refers to transaction amount associated with unique customer ID//

        }
}
```

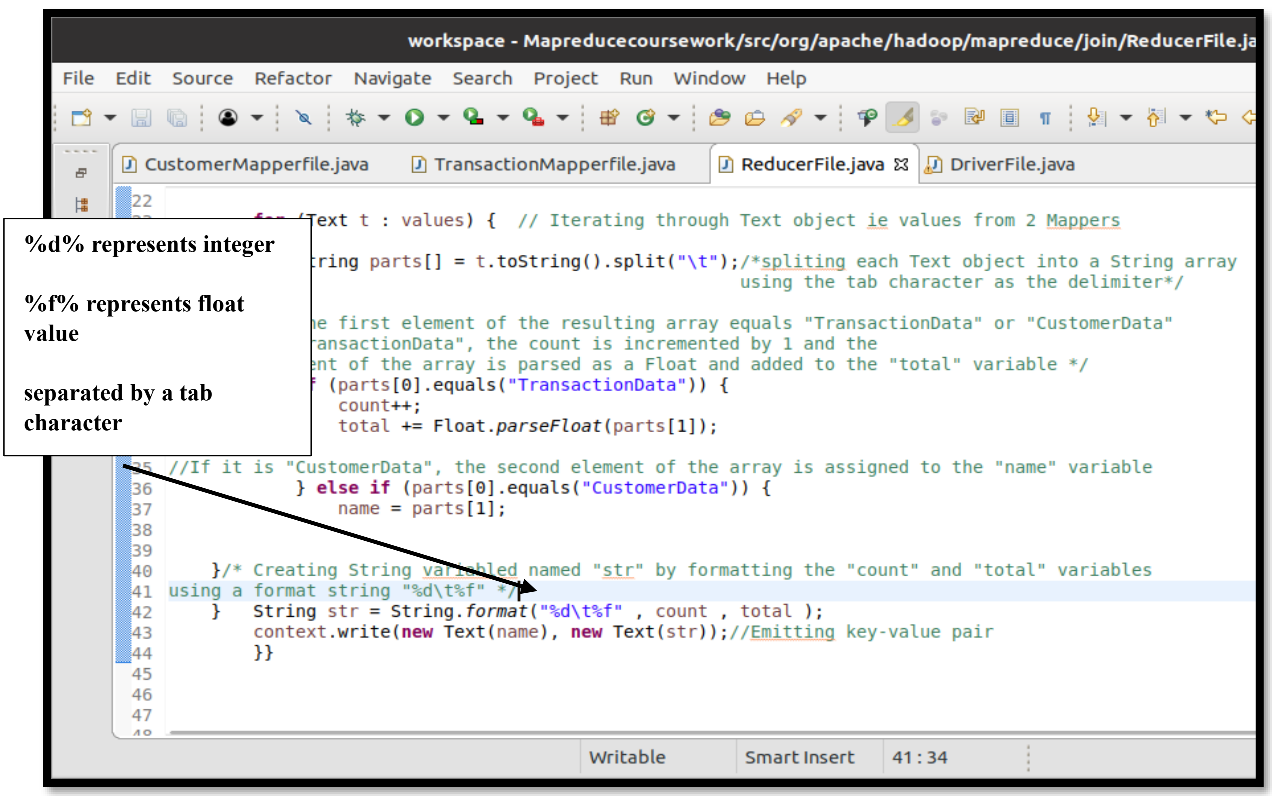
3) Sorting and Shuffling Phase:

The Sorting and Shuffling Phase will generate **an array list of values** assigned to **each unique key**.
The output at the end:
{Customer **ID 1** – [(Customer **Name 1**), (Sales 1) , (Sales 2) , (Sales 3) , (Sales 4),…]}
{Customer **ID 2** – [(Customer **Name 2**), (Sales 1) , (Sales 2) , (Sales 3) , (Sales 4),…]}

Addition of sales at every transaction

ReducerFile:





4) Reducer Phase:

The framework will call reduce method() for each unique join key ie Customer ID and list of values ie Customer Name and an array of Sales.
Reduce Method: (reduce(Text key, Iterable<Text>values, Context context)
Reducer will perform **join operation** on the values present in the respective list of values. This will give the desired output eventually.
Each reducer task will be performed for each key ie Customer ID => Number of reducer = Number of Customer ID
Each reducer will have: key (unique customer ID) and value(Customer Name from CustomerMapperfile and Sales from TransactionMapperfile)
This will be followed by looping over all the values present in the list of reducer. *Using if-elseif statement*
The list of values will be splitted to check if it belongs to CustomerData or TransactionData.
If it belongs to TransactionData: then the counter value will be increased by one to calculate the frequency of transactions by that particular customer. This will be cumulatively added through each iterations. In the end, total amount of sales will be retrieved after summation of sales at every transaction.
Alternatively, if the value is from the CustomerData, then it will be stored in a string variable "name"
At the end of reduce phase, customer names = key
number of transactions and total sales = value

5) Driver Phase:

Importing multiple input packages.Creating a user-defined job named Reduce-join.
Specifying the driver and reducer class in the Jar File
Assigning datatype of output key as Text and output value as Text which corresponds to String in JAVA.
Setting multiple input path to assign:
For input path 1:
Customer Name: passing argument [1]
Input format = Text Input Format class
Mapper = CustomerMapper class
For input path 2:
Number of transactions , sales :passing argument [2]
Input format = Text Input Format class
Mapper = TransactionMapper class
Assigning argument[3] for setting output path where the final output of Mapreduce file will be stored.

HDFS and Linux:

This will be followed by exporting the Jar file **ie HPC.jar** along with its specified class. Then open the Linux command terminal on the Hadoop environment. Formatting the name node. This will be followed by starting the hdfs and yarn system. Checking if all the six nodes are working using jps and the local host server. If all the six nodes ie name node, data node, resource manager, secondary name node, node manager, and jps working.
Creating a new directory in the HDFS named /input_file. Download two datafiles named CustomerData.txt and TransactionData.txt and put them in the HDFS directory /input_file. Execute the MapReduce program HPC.jar using Hadoop jar and store the output in the folder named output.

DriverFile:

CustomerMapperfile.java | TransactionMapperfile.java | ReducerFile.java | DriverFile.java

```java
package org.apache.hadoop.mapreduce.join;
//Importing the libraries that contain the classes and methods needed in the driver class
import java.io.IOException;
import org.apache.hadoop.conf.Configuration ;
import org.apache.hadoop.fs.Path ;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job ;
import org.apache.hadoop.mapreduce.lib.input.MultipleInputs ;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat ;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat ;

public class DriverFile {
    public static void main(String[] args) throws IOException,ClassNotFoundException, InterruptedException {
        /*creating the “conf” object from the “Configuration” class, which provides access to
         configuration parameters necessary for Hadoop job*/
         Configuration conf = new Configuration(); // creates a new instance of the Job class named "job"
/*The Job.getInstance() method returns a new Job instance with the
specified Configuration object and a user-defined job name "Reduce-side join"*/
         Job job;Job.getInstance() ;
         job=Job.getInstance(conf, "Reduce-side join");


         job.setJarByClass(DriverFile.class);//specifying the driver class in the Jar file
         job.setReducerClass(ReducerFile.class);//specifying the reducer class in the Jar file
         job.setOutputKeyClass(Text.class);//setting the data type of the key output for the MapReduce job to Text
         job.setOutputValueClass(Text.class);//setting the data type of the value output for the MapReduce job to Text
```

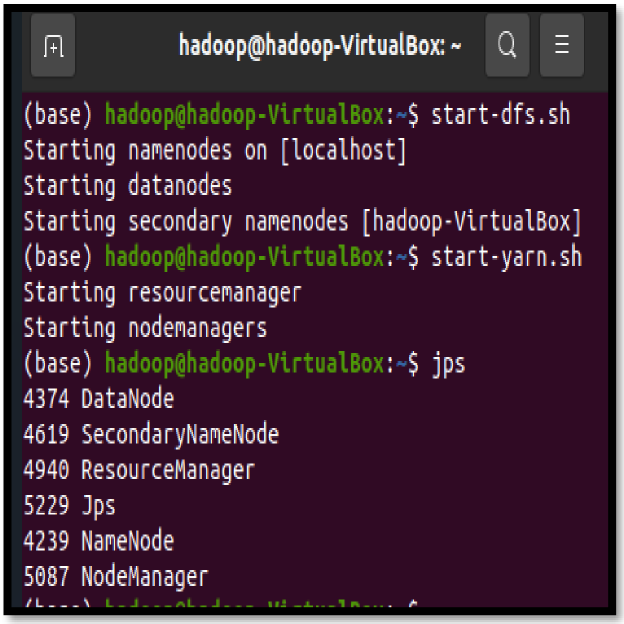
Output key: Text

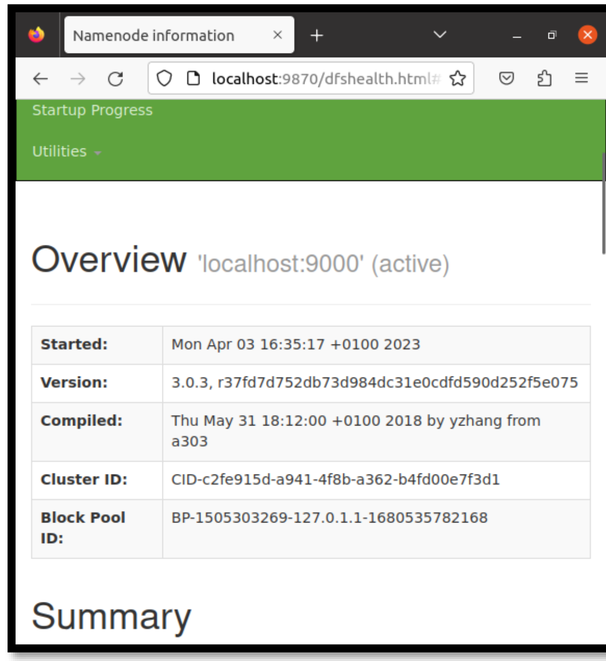
Output value: Text

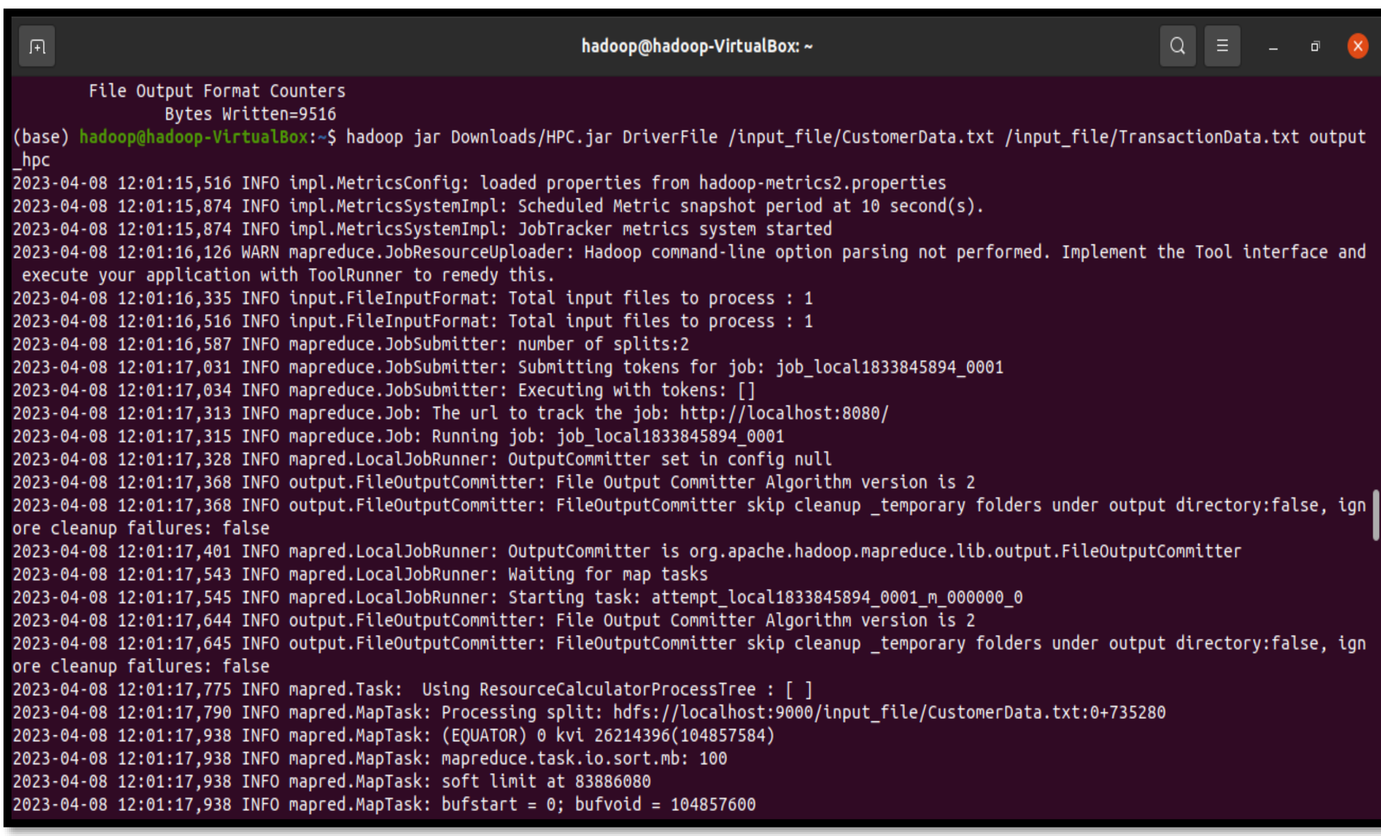
```java
//Setting multiple input paths for the Hadoop MapReduce job using the MultipleInputs class.
            MultipleInputs.addInputPath(job, new

//adding the input path of the customer data file specified by the second argument ie Customer I
//Input format:TextInputFormat class and the Mapper function:CustomerMapperfile class
            Path(args[1]),TextInputFormat.class, CustomerMapperfile.class);

// adding the input path of the transaction data file specified by the third argument ie Customer ID
//Input format: TextInputFormat class and the Mapper function: TransactionMapperfile class
            MultipleInputs.addInputPath(job, new
            Path(args[2]), TextInputFormat.class, TransactionMapperfile.class);
            Path outputPath = new Path(args[3]);/*output path is where the final output of
                                                the MapReduce job will be stored*/

            //Setting the output path for Mapreduce Jobs results
            FileOutputFormat.setOutputPath(job, outputPath);
            // Delete the outputPath directory from the file system
            outputPath.getFileSystem(conf).delete(outputPath);
             //check the status of the job (completed or not)
            System.exit(job.waitForCompletion(true)? 0:1);
```

2 Mapper files hence Multiple Inputs

```
(base) hadoop@hadoop-VirtualBox:~$ hdfs namenode -format
2023-04-03 16:29:39,878 INFO namenode.NameNode: STARTUP_MSG:
/************************************************************
STARTUP_MSG: Starting NameNode
STARTUP_MSG:   host = hadoop-VirtualBox/127.0.1.1
STARTUP_MSG:   args = [-format]
STARTUP_MSG:   version = 3.0.3
STARTUP_MSG:   classpath = /usr/local/hadoop/etc/hadoop:/usr/
/common/lib/jackson-annotations-2.7.8.jar:/usr/local/hadoop/s
/jsr311-api-1.1.1.jar:/usr/local/hadoop/share/hadoop/common/l
jar:/usr/local/hadoop/share/hadoop/common/lib/commons-lang3-3
doop/share/hadoop/common/lib/jaxb-impl-2.2.3-1.jar:/usr/local
p/share/hadoop/common/lib/jersey-server-1.19.jar:/usr/local/h
```


hadoop@hadoop-VirtualBox: ~
(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:~$ jps
4374 DataNode
4619 SecondaryNameNode
4940 ResourceManager
5229 Jps
4239 NameNode
5087 NodeManager


Namenode information
localhost:9870/dfshealth.html#
Startup Progress
Utilities
Overview 'localhost:9000' (active)
Started: Mon Apr 03 16:35:17 +0100 2023
Version: 3.0.3, r37fd7d752db73d984dc31e0cdfd590d252f5e075
Compiled: Thu May 31 18:12:00 +0100 2018 by yzhang from a303
Cluster ID: CID-c2fe915d-a941-4f8b-a362-b4fd00e7f3d1
Block Pool ID: BP-1505303269-127.0.1.1-1680535782168
Summary


hadoop@hadoop-VirtualBox: ~
File Output Format Counters
Bytes Written=9516
(base) hadoop@hadoop-VirtualBox:~$ hadoop jar Downloads/HPC.jar DriverFile /input_file/CustomerData.txt /input_file/TransactionData.txt output_hpc
2023-04-08 12:01:15,516 INFO impl.MetricsConfig: loaded properties from hadoop-metrics2.properties
2023-04-08 12:01:15,874 INFO impl.MetricsSystemImpl: Scheduled Metric snapshot period at 10 second(s).
2023-04-08 12:01:15,874 INFO impl.MetricsSystemImpl: JobTracker metrics system started
2023-04-08 12:01:16,126 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
2023-04-08 12:01:16,335 INFO input.FileInputFormat: Total input files to process : 1
2023-04-08 12:01:16,516 INFO input.FileInputFormat: Total input files to process : 1
2023-04-08 12:01:16,587 INFO mapreduce.JobSubmitter: number of splits:2
2023-04-08 12:01:17,031 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_local1833845894_0001
2023-04-08 12:01:17,034 INFO mapreduce.JobSubmitter: Executing with tokens: []
2023-04-08 12:01:17,313 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
2023-04-08 12:01:17,315 INFO mapreduce.Job: Running job: job_local1833845894_0001
2023-04-08 12:01:17,328 INFO mapred.LocalJobRunner: OutputCommitter set in config null
2023-04-08 12:01:17,368 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 2
2023-04-08 12:01:17,368 INFO output.FileOutputCommitter: FileOutputCommitter skip cleanup _temporary folders under output directory:false, ignore cleanup failures: false
2023-04-08 12:01:17,401 INFO mapred.LocalJobRunner: OutputCommitter is org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter
2023-04-08 12:01:17,543 INFO mapred.LocalJobRunner: Waiting for map tasks
2023-04-08 12:01:17,545 INFO mapred.LocalJobRunner: Starting task: attempt_local1833845894_0001_m_000000_0
2023-04-08 12:01:17,644 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 2
2023-04-08 12:01:17,645 INFO output.FileOutputCommitter: FileOutputCommitter skip cleanup _temporary folders under output directory:false, ignore cleanup failures: false
2023-04-08 12:01:17,775 INFO mapred.Task: Using ResourceCalculatorProcessTree : [ ]
2023-04-08 12:01:17,790 INFO mapred.MapTask: Processing split: hdfs://localhost:9000/input_file/CustomerData.txt:0+735280
2023-04-08 12:01:17,938 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
2023-04-08 12:01:17,938 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
2023-04-08 12:01:17,938 INFO mapred.MapTask: soft limit at 83886080
2023-04-08 12:01:17,938 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600

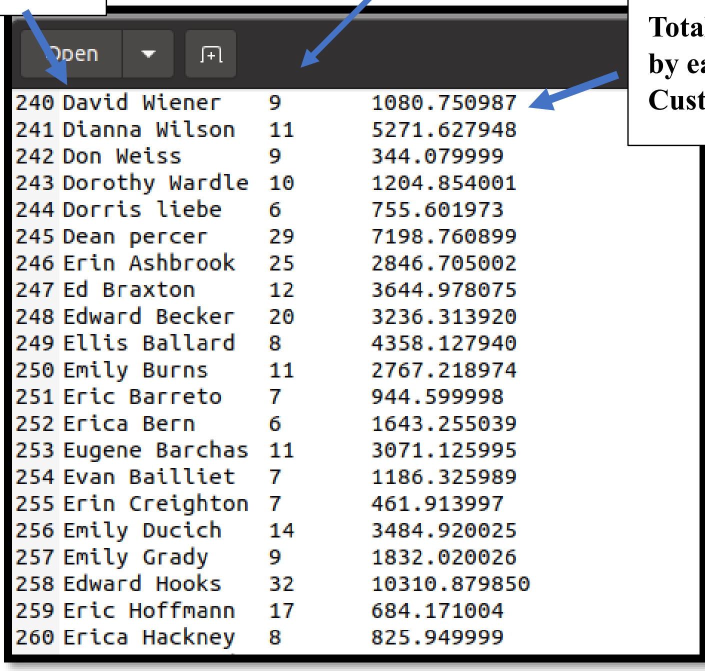
Section 3:

Results and Evaluations:

Total number of Transactions made by each Customer

Customer Name

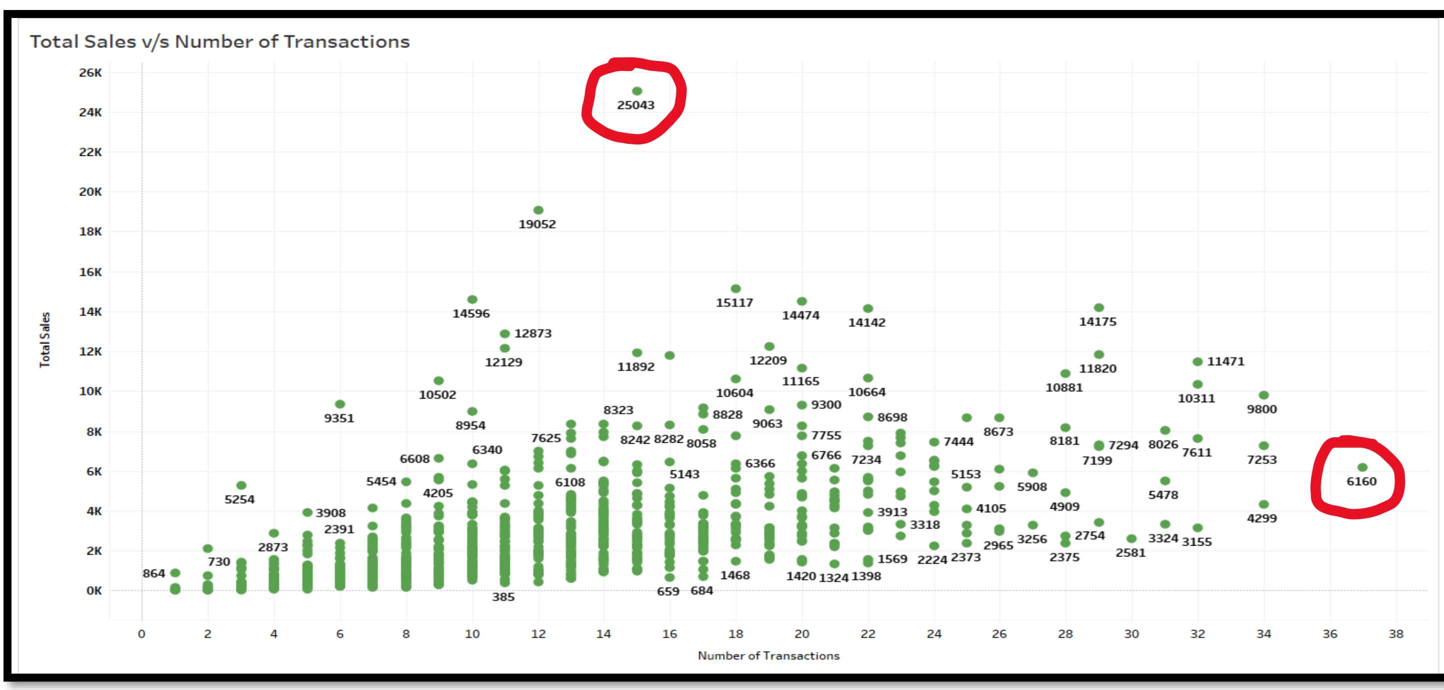
Total Sales by each Customer



240	David Wiener	9	1080.750987
241	Dianna Wilson	11	5271.627948
242	Don Weiss	9	344.079999
243	Dorothy Wardle	10	1204.854001
244	Dorris liebe	6	755.601973
245	Dean percer	29	7198.760899
246	Erin Ashbrook	25	2846.705002
247	Ed Braxton	12	3644.978075
248	Edward Becker	20	3236.313920
249	Ellis Ballard	8	4358.127940
250	Emily Burns	11	2767.218974
251	Eric Barreto	7	944.599998
252	Erica Bern	6	1643.255039
253	Eugene Barchas	11	3071.125995
254	Evan Bailliet	7	1186.325989
255	Erin Creighton	7	461.913997
256	Emily Ducich	14	3484.920025
257	Emily Grady	9	1832.020026
258	Edward Hooks	32	10310.879850
259	Eric Hoffmann	17	684.171004
260	Erica Hackney	8	825.949999

Output File

Customer Names with their total number of transactions and total sales were the output of the MapReduce program which helped to answer the desired research question: What is the total amount spent by each customer on their purchases, combined? **Insights from the Output: Scatterplot**



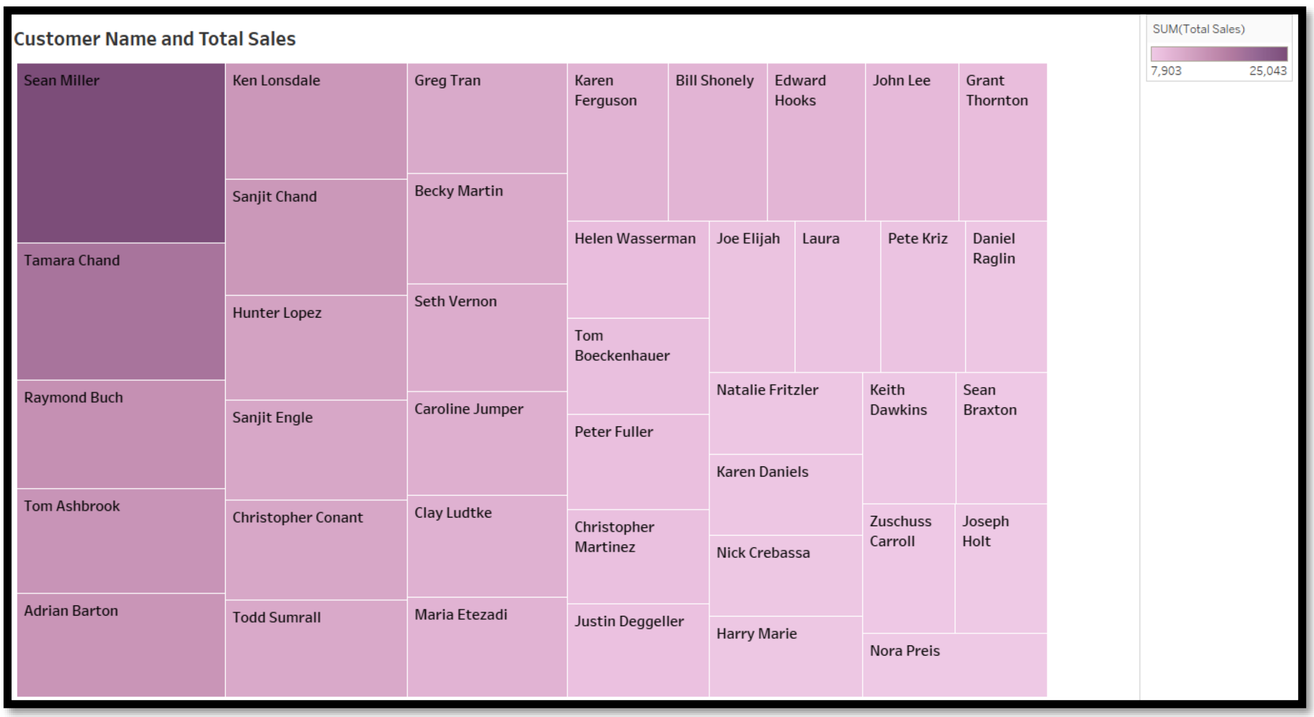
It can be seen from the above graph: **Correlation of Total Sales with the Number of Transactions** that the data points have homogenous clustering for the lower transaction numbers. The measure of the spread of the data points increases with the increase in the number of transactions. There is a positive correlation between the Total sales and the number of transactions.

However, being dependent on the number of transactions cannot help the store to predict future sales probability because it can be seen from the graph that one customer has 38(highest) transactions history but contributes to only $6160 sales amount. On the other hand, a customer with 16 transaction history has sales values of $25043. Finding a better variable for predicting total sales is important.

Tree Map:

The top 40 customers contributing to maximum sales for the superstore were sampled. They were plotted on the Tree Map to understand further details and target them for customer loyalty rewards, promotions, better services, marketing, etc. The customer named Sean Miller contributed to the highest sales whereas Nora Pries contributed to one-third of the amount spent by Sean.

Further, the region of people with the highest sales can be targeted and those outlets can have more discount offers to retain and satisfy their customers. Additionally, regions, where customer sales are less, can be investigated further to formulate strategies for optimizing their customer acquisition.



Reflection:

The rich data can be used to target the customers personally, congratulating them on their birthdays. Developing systems that communicate personalized recommendations to customers about deals and promotions. Big data can be processed faster and hence can be effectively implemented in the supermarket domain for joining two or more attributes, and for finding more hidden patterns in the dataset. Reduce side joins can be used for joining big data in the traditional HADOOP framework without the use of SQL. Additionally, the HADOOP environment can use Spark for the real-time streaming and batch processing of the superstore sales data.