**DS644:INTRODUCTION TO BIG DATA**

**SECTION: 852**

REPORT

On

MILESTONE 2

# **LOAN - CREDIT RISK & POPULATION STABILITY DATA ANALYSIS**

By

|  |  |  |
| --- | --- | --- |
| **NAME** | **UCID** | **EMAIL** |
| Kenil Pravinbhai Avaiya | ka683 | ka683@njit.edu |
| Neel Chimanbhai Barvaliya | nb585 | nb585@njit.edu |

**CONTENTS**

|  |  |  |
| --- | --- | --- |
| **NO** | **CONTENTS** | **PAGE NO.** |
| 1 | Introduction | 1 |
| 2 | MapReduce Code and Output | 2 |
| 3 | C**hallenges & Troubleshooting** | 17 |
| 4 | S**ummary & Key Learnings** | 19 |

**INTRODUCTION**

Understanding credit risk is paramount for sound financial decision-making, particularly for lending institutions. The advent of peer-to-peer lending platforms, such as LendingClub, underscores the imperative for effective analysis and management of extensive loan datasets. This project harnesses the power of big data technologies, specifically Hadoop and Oozie, to analyze approximately 1.8 million consumer loans issued by LendingClub from 2014 to 2018. The primary objective is to discern substantial patterns and trends in loan approvals, defaults, and borrower attributes.

The project performs three targeted MapReduce jobs:

1. **Default Rate by Term**: Calculates loan default percentages across terms, highlighting high-risk durations.
2. **Loan Amount by State**: Aggregates total loan amounts in each state, revealing geographical distribution and regions with high lending.
3. **Average Interest Rate by Grade**: Determines average interest rates for each loan grade, aiding understanding of loan grades and borrower risk.

Through this structured analysis, the project aims to uncover critical insights that enhance the comprehension of credit risk, inform lending decisions, and support strategic financial planning. The practical application of big data tools presented here demonstrates their efficacy in managing extensive datasets and extracting actionable information that is crucial for informed business and policy decisions.

**MAPREDUCE CODE AND OUTPUT**

Three MapReduce jobs were implemented to process loan data: default rate by term, loan amount by state, and average interest rate by grade. Each job comprises its own Mapper, Reducer, and Driver classes to facilitate data extraction, aggregation, and job configuration.

1. **Default Rate by Term**

In this MapReduce job, we tried to calculate the percentage of loans that are in default for the specified loan terms.

**Mapper (DefaultRateMapper):**

* Reads each CSV line, skips the header, splits into fields, checks for at least 17 fields, extracts loan term (index 5) and status (index 16), validates term (“36 months” or “60 months”), checks for default status (“Default” or “Charged Off”), creates CountWritable object with default count (1 if default, 0 otherwise), and emits key-value pair (loan term, CountWritable).

**Reducer (DefaultRateReducer):**

* Receives key-value pairs with loan term as key and CountWritable as value. Sums default and total loan counts over CountWritable objects. Computes default rate as (defaultCount / totalCount) \* 100. Outputs final key-value pair (loan term, default rate).

**Driver (DefaultRateByTermDriver):**

* Configures Hadoop job with custom JAR, Mapper, and Reducer classes, specifies output types, sets input and output paths, and executes the job.

**Java Code:**

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.DoubleWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.DataInput;

import java.io.DataOutput;

import java.io.IOException;

/\*\*

\* Hadoop MapReduce program to calculate the loan default rate by term.

\*/

public class DefaultRateByTermDriver {

/\*\*

\* Custom Writable class to hold default count and total count for rate calculation.

\*/

public static class CountWritable implements org.apache.hadoop.io.Writable {

private long defaultCount;

private long totalCount;

// Default constructor is required

public CountWritable() {}

public CountWritable(long defaultCount, long totalCount) {

this.defaultCount = defaultCount;

this.totalCount = totalCount;

}

public void set(long defaultCount, long totalCount) {

this.defaultCount = defaultCount;

this.totalCount = totalCount;

}

public long getDefaultCount() {

return defaultCount;

}

public long getTotalCount() {

return totalCount;

}

@Override

public void write(DataOutput out) throws IOException {

out.writeLong(defaultCount);

out.writeLong(totalCount);

}

@Override

public void readFields(DataInput in) throws IOException {

defaultCount = in.readLong();

totalCount = in.readLong();

}

}

/\*\*

\* Mapper class to extract the term and loan status from the input dataset.

\*/

public static class DefaultRateMapper extends Mapper<Object, Text, Text, CountWritable> {

private Text term = new Text();

private CountWritable count = new CountWritable();

@Override

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

String line = value.toString();

// Skip header row (assumes header starts with "Unnamed: 0")

if (line.startsWith("Unnamed: 0")) {

return;

}

// Split the input line. Adjust the delimiter if your file uses a different character.

String[] fields = line.split(",");

// Verify that we have enough columns (we need at least 17 fields)

if (fields.length > 16) {

// Based on header:

// Column index 5: term (e.g., "36 months", "60 months")

// Column index 16: loan\_status (e.g., "Current", "Default", "Charged Off")

String loanTerm = fields[5].trim();

String loanStatus = fields[16].trim();

// Validate that the loan term is one of our expected values and that loanStatus is not empty

if (loanTerm.matches("^(36|60) months$") && !loanStatus.isEmpty()) {

term.set(loanTerm);

// Check if loan status indicates a default condition

boolean isDefault = loanStatus.equalsIgnoreCase("Default") ||

loanStatus.equalsIgnoreCase("Charged Off");

// Set the count: default count is 1 if default, total count is always 1

count.set(isDefault ? 1 : 0, 1);

context.write(term, count);

}

}

}

}

/\*\*

\* Reducer class to compute the default rate for each loan term.

\*/

public static class DefaultRateReducer extends Reducer<Text, CountWritable, Text, DoubleWritable> {

private DoubleWritable result = new DoubleWritable();

@Override

public void reduce(Text key, Iterable<CountWritable> values, Context context)

throws IOException, InterruptedException {

long totalDefaultCount = 0;

long totalLoanCount = 0;

// Sum up the default counts and total loan counts for the given term

for (CountWritable val : values) {

totalDefaultCount += val.getDefaultCount();

totalLoanCount += val.getTotalCount();

}

// Calculate default rate as a percentage

if (totalLoanCount > 0) {

double defaultRate = ((double) totalDefaultCount / totalLoanCount) \* 100;

result.set(defaultRate);

context.write(key, result);

}

}

}

/\*\*

\* Driver method to configure and run the Hadoop MapReduce job.

\*

\* @param args Command-line arguments: input path and output path.

\*/

public static void main(String[] args) throws Exception {

// Expecting 2 arguments: input path and output path

if (args.length != 2) {

System.err.println("Usage: DefaultRateByTermDriver <input path> <output path>");

System.exit(-1);

}

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Loan Default Rate by Term");

job.setJarByClass(DefaultRateByTermDriver.class);

job.setMapperClass(DefaultRateMapper.class);

job.setReducerClass(DefaultRateReducer.class);

// Set output key/value types for mapper

job.setMapOutputKeyClass(Text.class);

job.setMapOutputValueClass(CountWritable.class);

// Set output key/value types for reducer (final output)

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(DoubleWritable.class);

FileInputFormat.addInputPath(job, new Path(args[0])); // input file/folder in HDFS

FileOutputFormat.setOutputPath(job, new Path(args[1])); // output folder in HDFS

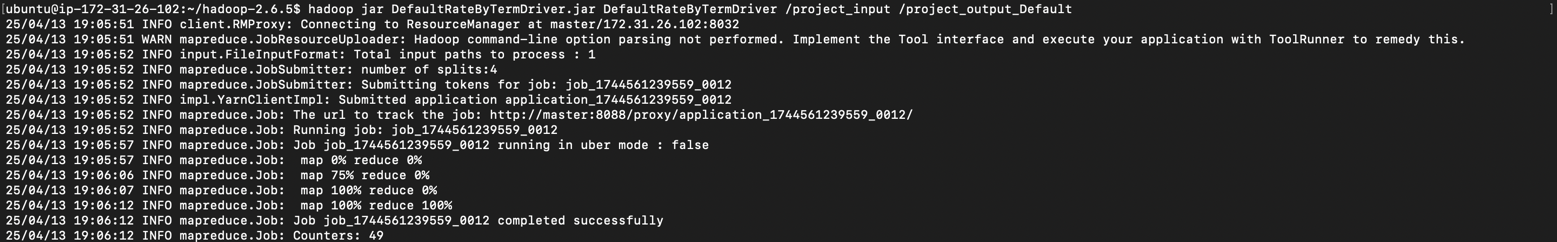
System.exit(job.waitForCompletion(true) ? 0 : 1);

}

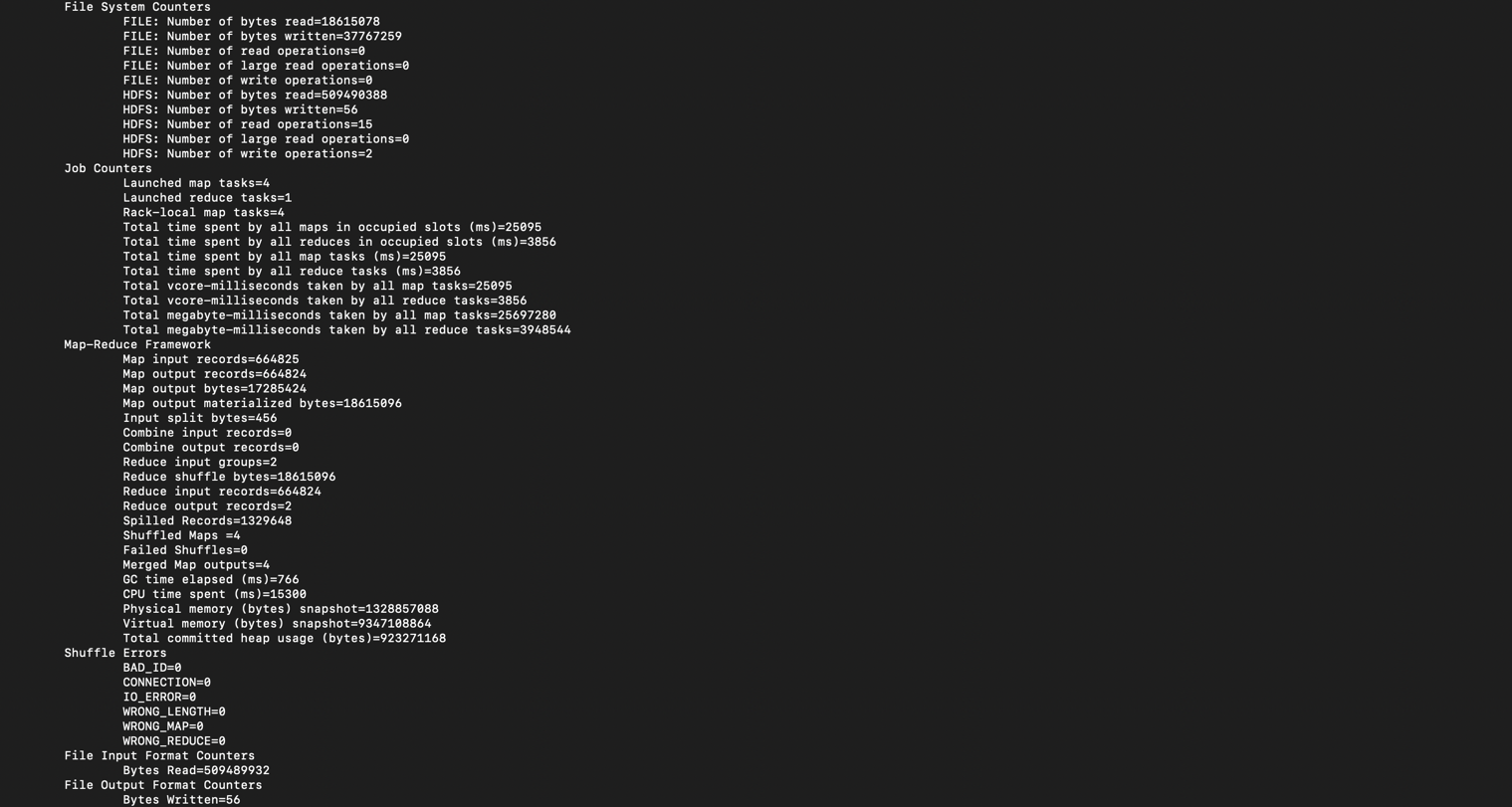
}

**Code Execution & Output Interpretation:**

1. The screen of running your code command and its result.



1. Process of MapReduce without any error until the end.

****

1. The screen of getting the output command and its result.

****

1. **Loan Amount by State**

In this MapReduce job, we tried to Aggregates the total loan amount for each US state.

**Mapper (DefaultRateMapper):**

* Each CSV line is read, its fields are separated, and the loan amount from column 3 (index 2) and the state (addr\_state) from column 23 (index 22) are extracted. The values are trimmed and validated using a regular expression to ensure they are valid two-letter US state codes. Key-value pairs in the format (state, loan\_amount) are emitted.

**Reducer (DefaultRateReducer):**

* Receives key-value pairs, where the key is the state and the value is the associated loan amount. Sum the loan amounts for each state to obtain the total. Output the final result as (state, total\_loan\_amount).

**Driver (DefaultRateByTermDriver):**

* Configures and initializes the Hadoop MapReduce job, setting the Mapper, Reducer, and Combiner classes, specifying output key and value types, reading input and output paths from command-line arguments, and executing the job.

**Java Code:**

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.DoubleWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

import java.util.regex.Pattern;

/\*\*

\* Hadoop MapReduce program to calculate the total loan amount by state.

\*/

public class LoanDriver {

/\*\*

\* Mapper class to extract state and loan amount from the input dataset.

\*/

public static class LoanMapper extends Mapper<Object, Text, Text, DoubleWritable> {

private Text state = new Text();

private DoubleWritable loanAmount = new DoubleWritable();

private static final Pattern STATE\_PATTERN = Pattern.compile("^[A-Z]{2}$"); // Valid 2-letter state codes

@Override

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

String[] fields = value.toString().split(","); // Assuming CSV format

try {

if (fields.length > 22) { // Checking if column exists

String loanState = fields[22].trim(); // Extract state (addr\_state)

double loanAmountValue = Double.parseDouble(fields[2].trim()); // Extract loan amount (loan\_amnt)

// Process only if the state is a valid US state abbreviation

if (STATE\_PATTERN.matcher(loanState).matches()) {

state.set(loanState);

loanAmount.set(loanAmountValue);

context.write(state, loanAmount); // Emit state and loan amount

}

}

} catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {

System.err.println("Skipping invalid line: " + value.toString());

}

}

}

/\*\*

\* Reducer class to sum up loan amounts for each state.

\*/

public static class LoanReducer extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {

private DoubleWritable result = new DoubleWritable();

@Override

public void reduce(Text key, Iterable<DoubleWritable> values, Context context)

throws IOException, InterruptedException {

double sum = 0.0;

// Sum up loan amounts for the given state

for (DoubleWritable val : values) {

sum += val.get();

}

result.set(sum);

context.write(key, result); // Emit state and total loan amount

}

}

/\*\*

\* Driver class to configure and run the Hadoop MapReduce job.

\*/

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Loan Amount by State");

job.setJarByClass(LoanDriver.class);

job.setMapperClass(LoanMapper.class);

job.setCombinerClass(LoanReducer.class); // Combiner optimizes performance

job.setReducerClass(LoanReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(DoubleWritable.class);

// Input and output paths from command-line arguments

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

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

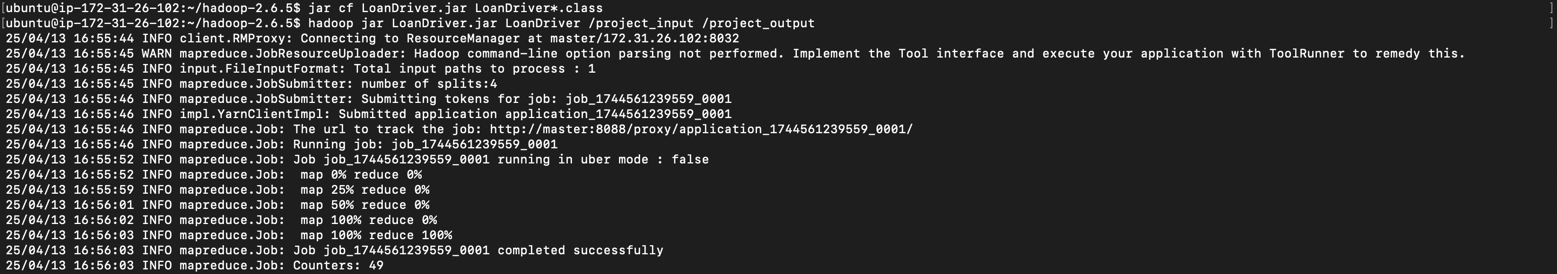
System.exit(job.waitForCompletion(true) ? 0 : 1);

}

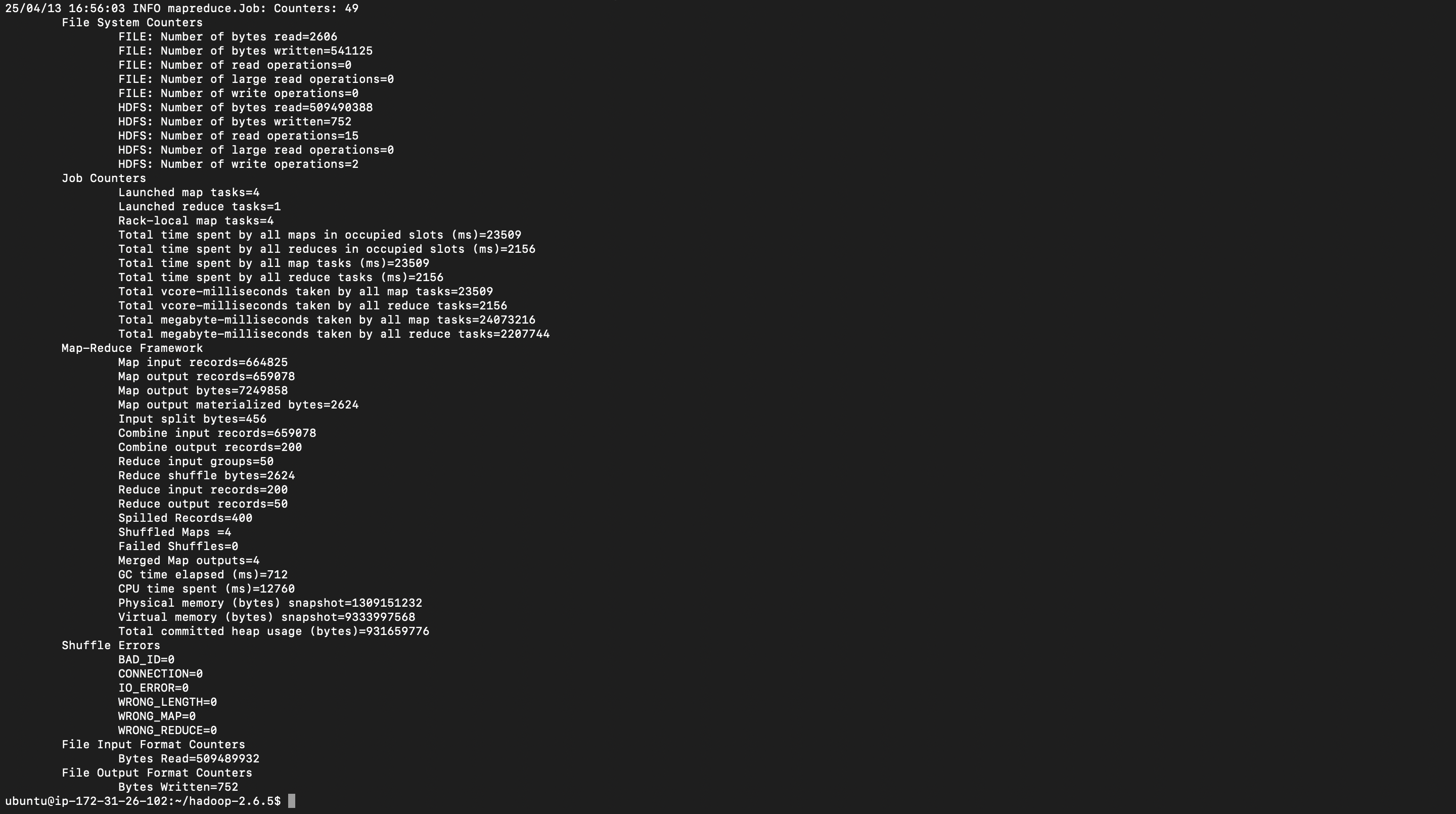
}

**Code Execution & Output Interpretation:**

1. The screen of running your code command and its result.



1. Process of MapReduce without any error until the end.

****

1. The screen of getting the output command and its result.

****

1. **Average Interest Rate by Grade**

In this MapReduce job, we tried to Calculates the average interest rate for each loan grade.

**Mapper (DefaultRateMapper):**

* Read each CSV line, skip the header, split into fields, extract loan grade (column 7) and interest rate (column 11), clean the interest rate (e.g., remove quotes), and emit key-value pairs (loan grade, interest rate) for valid records.

**Reducer (DefaultRateReducer):**

* Receives key-value pairs, where loan grades are keys and corresponding interest rates are values. Calculates the sum of interest rates for each grade, counts the number of entries, and computes the average interest rate. Outputs (loan grade, average interest rate).

**Driver (DefaultRateByTermDriver):**

* Configures and initializes the Hadoop MapReduce job, specifying mapper and reducer classes, defining output key/value types, setting input and output HDFS paths from command-line parameters, and executing the job, terminating it based on its success.

**Java Code:**

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.DoubleWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;

/\*\*

\* Hadoop MapReduce program to calculate the average interest rate by loan grade.

\*/

public class InterestRateByGradeDriver {

/\*\*

\* Mapper class to extract grade and interest rate from the input dataset.

\*/

public static class InterestRateMapper extends Mapper<Object, Text, Text, DoubleWritable> {

private Text grade = new Text();

private DoubleWritable interestRate = new DoubleWritable();

@Override

public void map(Object key, Text value, Context context) throws IOException, InterruptedException {

String[] fields = value.toString().split(","); // Assuming CSV format

try {

// Skip header row (first field is "id")

if (fields.length > 0 && fields[0].trim().equals("id")) {

return; // Skip header

}

if (fields.length > 11) { // Ensure required columns exist

String loanGrade = fields[6].trim(); // Extract grade (column 7, 0-based index 6)

String intRateStr = fields[10].trim(); // Extract int\_rate (column 11, 0-based index 10)

// Remove quotes if present (e.g., "7.97" -> 7.97)

intRateStr = intRateStr.replaceAll("\"", "");

// Parse interest rate

double intRate = Double.parseDouble(intRateStr);

// Process only if grade is not empty and int\_rate is valid

if (!loanGrade.isEmpty()) {

grade.set(loanGrade);

interestRate.set(intRate);

context.write(grade, interestRate); // Emit (grade, int\_rate)

} else {

System.err.println("Skipping record due to empty grade: " + value.toString());

}

} else {

System.err.println("Skipping record due to insufficient columns: " + value.toString());

}

} catch (NumberFormatException e) {

System.err.println("Skipping record due to invalid int\_rate format: " + value.toString());

} catch (ArrayIndexOutOfBoundsException e) {

System.err.println("Skipping record due to array index error: " + value.toString());

}

}

}

/\*\*

\* Reducer class to compute the average interest rate for each grade.

\*/

public static class InterestRateReducer extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {

private DoubleWritable result = new DoubleWritable();

@Override

public void reduce(Text key, Iterable<DoubleWritable> values, Context context)

throws IOException, InterruptedException {

double sum = 0.0;

int count = 0;

// Sum up interest rates and count the number of loans for each grade

for (DoubleWritable val : values) {

sum += val.get();

count++;

}

// Calculate average interest rate

double average = (count > 0) ? sum / count : 0.0;

result.set(average);

context.write(key, result); // Emit (grade, average int\_rate)

}

}

/\*\*

\* Driver class to configure and run the Hadoop MapReduce job.

\*/

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = Job.getInstance(conf, "Average Interest Rate by Grade");

// Set the jar, mapper, and reducer classes

job.setJarByClass(InterestRateByGradeDriver.class);

job.setMapperClass(InterestRateMapper.class);

job.setReducerClass(InterestRateReducer.class);

// Set output key and value types

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(DoubleWritable.class);

// Set input and output paths from command-line arguments

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

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

// Run the job and exit

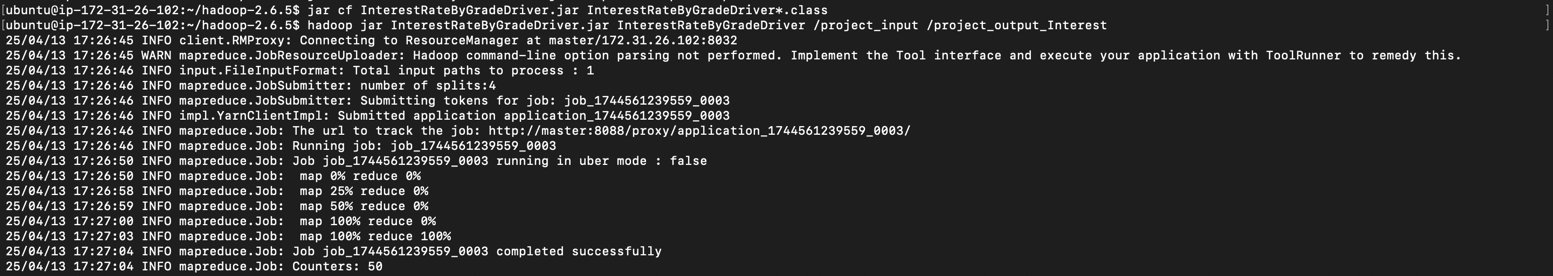
System.exit(job.waitForCompletion(true) ? 0 : 1);

}

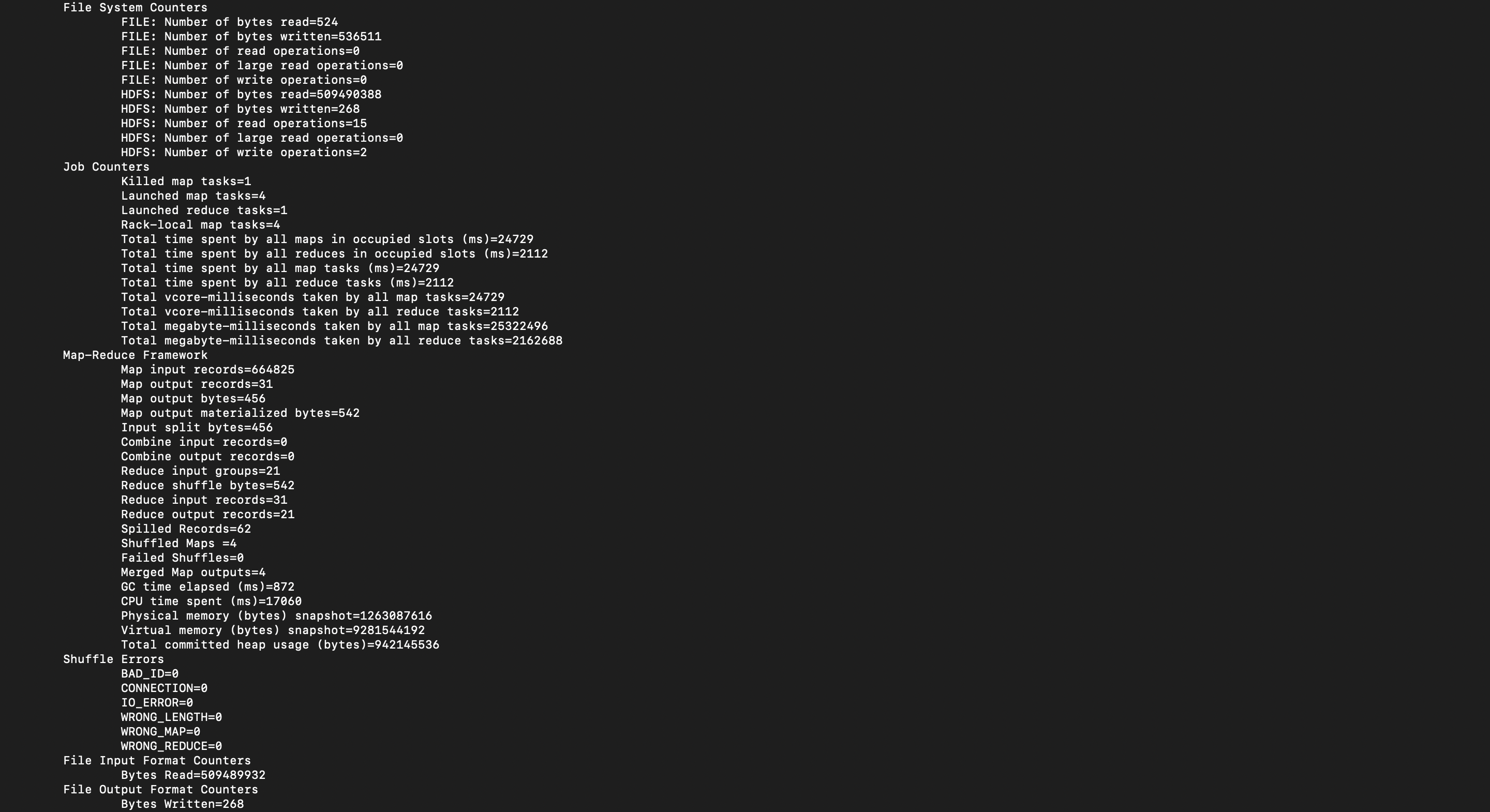
}

**Code Execution & Output Interpretation:**

1. The screen of running your code command and its result.



1. Process of MapReduce without any error until the end.

****

1. The screen of getting the output command and its result.

****

**Step for Compilation & Execution Commands for each and every MapReduce jobs.**

1. **Create a Java File or Upload Java File from Local:**

* Create a new Java file for the MapReduce WordCount program: **vim DefaultRateByTermDriver.jar**

(Write the Java code for the MapReduce WordCount program. You can find code examples online.)

Once completed, press `ESC`, type `:wq` to save, and hit Enter to exit the editor.

To upload a Java file, use the following command: **scp -i <path of your key> <path of the Java file> ubuntu@54.172.163.69:/home/<username>/**

1. **Compile the Java Program**

* Compile the Java program using the following command:

**javac DefaultRateByTermDriver.java -cp $(hadoop classpath)**

(Instead of “DefaultRateByTermDriver.jar,” use the name of your code file.)

1. **Create a JAR File**

* Create a JAR file using the following command:

**jar cf DefaultRateByTermDriver.jar DefaultRateByTermDriver\*.class**

1. Run the MapReduce Job

* Run the MapReduce job using the following command:

**hadoop jar DefaultRateByTermDriver.jar DefaultRateByTermDriver /project\_input /project\_output\_Default**

• Replace “/project\_input” with the name of your input directory.

• Replace “/project\_output\_Default” with the desired name of your output directory.

1. **Retrieve and view the results:**

* Retrieve the results from HDFS using the following command:

**bin/hdfs dfs -get /project\_output\_Default**

* Change the directory to “project\_output\_Default” and view the results using the following command:

**cd project\_output\_Default**

**cat part-r-00000**

**CHALLENGES & TROUBLESHOOTING**

**Challenges we faced throughout the project milestones:**

* **Dealing with Large Data Volume:**

Handling a sizable dataset (~510 MB) in CSV format was quite challenging, especially on local systems where standard tools slowed down and ran out of memory quickly.

* **Designing Effective MapReduce Logic:**

Creating efficient Map and Reduce functions initially required a solid grasp of the dataset’s schema and an understanding of Hadoop data flow to accurately calculate insights like default rates and total loan amounts by state.

* **Setting Up Hadoop Cluster on AWS:**

Encountered several technical hiccups while configuring Hadoop on AWS EC2:

* + Issues with mismatched SSH keys affecting communication between nodes.
  + Misconfigured environment variables for Hadoop.
  + Incorrect settings in core-site.xml and hdfs-site.xml, causing problems accessing the file system.
* **Frequent Job Failures:**

Several MapReduce jobs initially failed due to:

* Incorrect HDFS file permissions.
* When I tried to run Hadoop, I got a data node running failure.
* Misconfigured input/output paths.
* Java heap space runs out during large-scale computations.

**Troubleshooting Steps:**

* **Data Sampling and Testing:**

Used smaller subsets of the data initially to manage memory issues and confirm that the logic was working correctly before scaling up.

* **Step-by-Step Debugging:**

Independently tested Map and Reduce functions with mock data to pinpoint and fix logic issues early on, ensuring smoother integration later.

* **Resolving AWS Configuration Issues:**
* Set up passphrase-less SSH logins for smooth communication between nodes.
* Regularly checked daemon statuses using jps to ensure Hadoop services ran correctly on all nodes.
* **Managing Permissions and Heap Size:**
* Fixed file permissions using the **hdfs dfs -chmod** command.
* Also, for re-run data node, using the command **hadoop-daemon.sh start datanode** command.
* Increased the Java heap size using suitable JVM flags to prevent memory exhaustion during heavy computations.

**Performance Observations:**

* **Identified Bottlenecks:**
* Initial Delays During Data Transfer to HDFS
* Initial delays may occur during the transfer of data to HDFS.
* High memory usage may be experienced during intensive computation phases, particularly on nodes with limited RAM.
* **Effective Optimization Approaches:**
  + Implemented Combiner classes to substantially minimize data shuffling between the Map and Reduce phases.
  + Optimized the number of reducers to evenly distribute workloads across nodes.
  + Reduced unnecessary intermediate writes to HDFS to enhance overall performance.

**SUMMARY & KEY LEARNINGS**

**Project Reflection:**

* Working on this project was a great learning experience. I became comfortable handling large CSV datasets and typical big data challenges like memory overloads and process crashes. This practical experience gave me insights beyond theoretical knowledge.
* Optimizing the pipeline using Hadoop MapReduce was rewarding. Combining combiners and distributed computation enhanced processing efficiency and speed.
* I gained a clear understanding of Hadoop’s scalability solutions for managing massive data across nodes.

**Real-World Application:**

* Technologies such as MapReduce and Hadoop are widely utilized in industries including finance, e-commerce, healthcare, and telecommunications. In the financial sector, these tools facilitate credit risk analysis, fraud detection, and the management of extensive customer data.
* This project provided a clear demonstration of how big data tools can effectively manage vast, structured datasets, empowering businesses to make informed decisions based on comprehensive analytical insights.

**Suggestions for Future Improvements:**

* **Optimization Opportunities:**
  + Experiment with advanced MapReduce techniques, including secondary sorting and strategic data partitioning, to enhance computational efficiency.
  + Explore the utilization of additional Hadoop ecosystem tools, such as Apache Spark, for expedited, in-memory data processing.
* **Possible Extensions:**
  + Enhance the analysis by employing predictive models integrated with machine learning libraries such as Apache Mahout or Spark MLlib to enhance loan default prediction accuracy.
  + Perform comprehensive studies to analyze data trends over extended timeframes, thereby offering more profound insights into borrower behavior and economic conditions.
  + Exploring real-time data processing tools like Apache Kafka could provide quicker insights and enhance decision-making, especially valuable in scenarios requiring immediate actions.