CS236: DATABASE MANAGEMENT SYSTEMS
PROJECT REPORT

Team Members:

- **Devasheesh Vaid**
  SID: 862395097
  Email: dvaid003@ucr.edu

- **Aryan Singh**
  SID: 862394968
  Email: asing301@ucr.edu

How to Execute the Jar file:

Store the *DBMS_MAPRED.jar* file and the *DBMS_dataset.csv* dataset file on local device. Start the hadoop cluster with the datanode and namenode running on it, and type the below commands:
hadoop fs –mkdir /input
hadoop fs –put <local_path_of_DBMS_dataset.csv> /input
hadoop jar DBMS_MAPRED.jar MonthRevenueRanking /input/DBMS_dataset.csv /output
*hadoop dfs -cat /output/**

Contributions:

Equal contributions have been made by both the members of the team for its completion. We have worked on this project with detailed discussions. Co-ordinated effort has been made by us to work on the data preprocessing, coding, errors and debugging. **For extra credit we have accomplished point 2** (*A clever way to achieve faster execution time*). The detailed contributions are as follows:

Devasheesh:

- Set up the infrastructure of Hadoop Map-reduce cluster for local execution.
- Created the python script (Link here) for data preprocessing and joining the two datasets.
- Contributed to the development of Hadoop Map-Reduce java code.
- Contributed to the project documentation.

Aryan:

- Contributed to the python script for data preprocessing and joining the two datasets.
- Contributed to the development of Hadoop Map-Reduce java code.
- Prepared the project documentation.

Description:

Initially we thought of implementing a brute force approach using 2 Map reduce jobs. Where first job will just calculate the sum for each month and the second job would take those summed <year_month,revenue> pairs and emit the sorted output by swapping the key and values in between in the process. But to accomplish **point 2 of the extra credit** and to design a clever way to achieve faster execution time, we modified the code so that the two map-reduce jobs can be merged into one. This resulted in one less pass on the MR job, hence faster execution time. The detailed description the map and reduce steps is provided below-

1) Mapper:

The MonthRevenueMapper class is in charge of analyzing incoming data and extracting important information. Here's a detailed explanation of what the mapper does:

a. Input:

The mapper is fed data in the form of lines, with each line representing a record. Each record is considered to have three columns of comma-separated values: arrival year, arrival month, and total income.

b. Parsing and Validation:

The mapper starts by dividing the input line using a comma as a delimiter. It verifies that the line contains three columns as intended. If the validation fails, the record is skipped by the mapper. The split function, which acts on the length of the string, is used to split the input line into columns during the parsing process. The temporal complexity of this operation is O(n), where n is the length of the input line. The validation phase, which is a constant-time process, examines whether the line has three columns.

c. Data Extraction:

Assuming the record is legitimate, the mapper extracts the pertinent information from the columns. It transforms strings to integers for the arrival year and month, and doubles for the total income. During the extraction process, strings are converted to integers and doubles using methods like Integer.parseInt and Double.parseDouble. The temporal complexity of these approaches is typically O(k), where k is the number of characters in the input string being processed.

d. Emitting Key-Value Pairs:

The mapper then merges the arrival year and month into a single key, expressed as an integer (e.g., 202301 for January 2023). The entire revenue becomes the key-value pair's value. Because it requires establishing the yearMonth and revenue variables and invoking the context, emitting key-value pairs is a constant-time operation.Method should be written.

e. Output:

The MapReduce framework receives the key-value pair (yearMonth, revenue) from the mapper. As a result, the mapper's overall time complexity may be roughly calculated as O(n), where n is the length of the input line.

2) Reducer:

The MonthRevenueReducer class takes the intermediate key-value pairs that the mapper produces and aggregates and ranks them according to the revenue for that particular month. The steps of the reduction are broken down in detail below:

a. Initialization:

The reducer initializes a TreeMap named revenueMap with a comparator that arranges entries in reverse according to the revenue (that is, from highest to lowest) before processing the key-value pairs.

b. Key-Value Pair Processing:

The reducer gets a list of values (revenues) related to each distinct key (yearMonth). To get the overall income for that month, it iterates over the values and adds them up. Each distinct key (yearMonth) receives an iterable of data that the reducer must process. The'm' indicates how many values there are for that key, and this determines how time-consuming it is to iterate over the values. As a result, the processing of the values related to a key may be said to have an approximate O(m) time complexity.

c. Sorting and Ranking:

The reducer adds the month (as an IntWritable) and the total revenue (as the key) to the revenueMap. The elements will be arranged in decreasing order according to revenue because the TreeMap is configured with reverse ordering. The reducer stores the overall revenue as the key and the month as the value in a TreeMap it calls revenueMap. The complexity of the insertion operation into the TreeMap is O(log n), where 'n' is the number of elements already existing in the map.

d. Cleanup and Output:

The cleanup procedure is invoked once all the key-value pairs have been processed. It outputs each month (as an IntWritable) and its accompanying income (as a DoubleWritable) to the context by iterating over the sorted entries in the revenueMap. The end result is a list of months in descending order of revenue. In the cleanup stage, the entries in the revenueMap are iterated over in order to produce the months and associated revenues. The amount of entries in the map, indicated by the letter "k," determines how time-consuming this phase is. The time complexity for cleanup and output is therefore roughly O(k).

The overall time complexity of the reducer can be approximated as O(m log n + k), where'm' denotes the number of values for a key, 'n' denotes the number of entries in the revenueMap, and 'k' denotes the number of entries in the map during cleanup.

Description of Join:

The python script for data preprocessing and joining the two datasets can be [found here.]()
Here, we loaded the two datasets, removed the null values (if any) and any rows corresponding to the cancelled booking. We calculated the total_revenue for each booking by multiplying the ‘total_nights_stayed’ to the ‘avg_price_per_room’ and removed all columns from the datasets except for ‘arrival_year’, ‘arrival_month’ and ‘total_revenue’. Finally, the join was performed by using the pandas’ concatenation method on both the datasets:

data_final=pd.concat([data_hbooking,data_custresv])

Code Snippets:

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Month Revenue Ranking");
    job.setJarByClass(MonthRevenueRanking.class);
    job.setMapperClass(MonthRevenueMapper.class);
    job.setReducerClass(MonthRevenueReducer.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(DoubleWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Fig 1: main()function

```java
ublic class MonthRevenueRanking {
    public static class MonthRevenueMapper extends Mapper<Object, Text, IntWritable, DoubleWritable> {
        IntWritable yearMonth = new IntWritable();
        DoubleWritable revenue = new DoubleWritable();

        public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            String[] columns = value.toString().split(",");
            if (columns.length == 3) {
                int arrivalYear = Integer.parseInt(columns[0]);
                int arrivalMonth = Integer.parseInt(columns[1]);
                double totalRevenue = Double.parseDouble(columns[2]);

                yearMonth.set(arrivalYear * 100 + arrivalMonth);
                revenue.set(totalRevenue);

                context.write(yearMonth, revenue);
            }
        }
    }
```

Fig 2: MonthRevenueMapper Map class

```java
    public static class MonthRevenueReducer extends Reducer<IntWritable, DoubleWritable, IntWritable, DoubleWritable> {
        TreeMap<Double, IntWritable> revenueMap;

        protected void setup(Context context) {
            revenueMap = new TreeMap<>(Collections.reverseOrder());
        }

        public void reduce(IntWritable key, Iterable<DoubleWritable> values, Context context)
                throws IOException, InterruptedException {
            Double totalRevenue = 0.0;
            for (DoubleWritable value : values) {
                totalRevenue += value.get();
            }
            revenueMap.put(totalRevenue, new IntWritable(key.get()));
            totalRevenue = 0.0;
        }

        protected void cleanup(Context context) throws IOException, InterruptedException {
            for (Map.Entry<Double, IntWritable> entry : revenueMap.entrySet()) {
                double revenue = entry.getKey();
                IntWritable month = entry.getValue();
                context.write(month, new DoubleWritable(revenue));
            }
        }
    }
```

Fig 3: MonthRevenueReducer Reduce class.

```
[maria_dev@sandbox newtest]$ hadoop dfs -cat /WordCountTutorial/Output_finale2/*
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

201608  1809324.750000001
201607  1525019.0500000007
201609  1289642.69
201606  1144800.3799999987
201508  1137652.7100000002
201605  1073277.6300000027
201610  1072101.8399999994
201509  1054620.67
201604  896591.3800000026
201510  784714.8799999994
201603  767337.4200000009
201507  758339.790000001
201611  688843.47
201612  657870.7199999997
201810  575239.0300000007
201808  553001.7400000001
201806  550271.8499999996
201809  545198.2899999998
201812  532540.3099999997
201807  527771.6699999993
201805  520029.64000000054
201804  507850.0199999999
201602  484170.7200000003
201803  439599.3000000003
201811  430904.9700000002
201512  429521.5700000004
201710  409479.4400000002
201709  404397.08000000054
201511  346709.4900000003
201802  280068.17999999964
201601  264521.3800000001
201712  213519.2700000002
201801  206380.39000000013
201708  204236.5300000001
201711  122542.08000000005
201707  28074.879999999997
```

Fig 4: Final sorted output