**Project 1 Hadoop Query**

**Xiang Huang, ZhongZheng Shu**

1. **Creating Datasets**

We wrote two classes, GenerateData and DataGenerateMain to create the desired datasets. The first class defines several methods to generate different types of data, including generateInteger(), generateCharacters(), generateFloat(), and generateName(). This class utilizes the random number generator and ASCII table to generate data according to the requirements. The second class is a driver which invokes methods from the first class to generate datasets we need. Each column in the datasets is separated by a comma, as is specified in the project description. Eventually we created two datasets, customers.txt which is around 1.8Mb and transactions.txt which is 181MB.

1. **Uploading Data into Hadoop**

After uploading two txt files into the oracle VM, we are ready to upload our file to the Hadoop System. We simply used “hadoop fs –put” command to upload our file to /user/hadoop/input folder.

1. **Writing Jobs**
   1. **Query1**

This problem asks us to write a filter for the original customers dataset. We can solve it with a map-only job by only writing out records that meet the criteria. The runtime of our job is around 220ms.

* 1. **Query2**

This problem asks us to write an aggregation of certain attributes for the transactions dataset. We solve it with a map-reduce job. It maps the original record into a key-value tuple, where the key is the customer ID and the value is an ArrayWritable that stores the number of one (representing one transaction) and the field of TransTotal. During the reducer phase, the reducer will sum up all the TransTotal and count the number of transactions for a given customer. With four reducers, the runtime without combiner is 26950ms.

We can improve that by pre-combining the output at each mapper. With a combiner that functions similar to that of reducer, the runtime is lowered to 31800ms.

* 1. **Query3**

This problem asks us to write a join and aggregation job based on customers and transactions. We tried two different implementations: Improved-Repartition Join and a Map-side Join.

The Repartition Join is implemented as follows. First of all, we would need a composite key for storing the joining key and order of join. This is implemented as the FlaggedKey class, which implements WritableComparable to allow for sorting and grouping in the sorting and grouping process. You also have to override the original hash () function for this class to ensure all records with the same joining key are partitioned to the same reducer (to serve aggregation purposes). We also need to write two new classes to store the non-join attributes from each relations. After that, we can start writing the Map-Reduce Job.

The Mapper would produce a FlaggedKey-Generics tuple, and the output record is sorted based on the joining key and the joining order. The reason why it is sorted this way is that we want to ensure the records with a lower joining order will arrive at the reducer first, so that we can load the corresponding records into memory beforehand and joins with the following records, which is from another relation. The runtime of this method is 65400ms.

As for map-side join, we first load customers data into our Mapper, and store the information we need as a HashMap. Its key is a customer ID and the value is a corresponding CustomerInfo object which has three attributes including customer ID, name and salary. For each line mapper read from transactions data, we can join it with customer information by customer ID. Mapper then output a CustomerInfo object as a key and three other transactions fields as value. After receiving the key-value pairs from Mapper, Reducer calculate the number of transactions, TotalSum and minimum number of items in transactions done by each customer. This method takes 66440ms to complete.

* 1. Query 4

The problem asks us to write a job to report the number of customers from each country, and the maximum and minimum transactions done by these customers. Here is how we do it in a single map-reduce job. By using a broadcast join, we can load the customers into our mapper and emits joined records based on customer ID. The output key should be the CountryCode, with values being a FloatWritable storing the Transtotal information. Then in the reducer phase, we just need figure out how many customers there are and the maximum and minimum Transtotal. It takes 22600ms to finish.

* 1. Query 5

The problem asks us to find the customer name with the minimum number of transactions made. There are several options to do after-wards. We first tried a single map-reduce job that is to do a map-side join and output all the records to one single reducer. We keep a HashMap in the reducer which uses the Customers’ name as key and a class named Inverse to store a count-name pair. After processing all the records, we constructed a priority queue of the values from the HashMap, which is sorted based on count. All records that share the same count with the first instance in the Priority Queue are indeed the results. The entire process takes 24040ms to finish.

Then we tried a chain map-reduce job, where the first job is a simple map-side join and we find the minimum in the second job. While this is slower than the first implementation (This one takes 33130ms), but it offers a general solution to this problem as it can handle conditions where a HashMap does not fit in the memory.

Finally we tried another way to run chain map-reduce job. The first job is to group transactions data by customer ID, and calculate the number of transactions for each customer. This job then output customerIDs and numOftransactions, and stores them as a temporary file. In the second job, we first read the temporary data by the Mapper, and output a single number, one, as the key, and one line as the value The Reducer read the customers.txt file at first, and stores the information we need into a HashMap. Then the Reducer finds the customer ID whose number of transactions are minimum, and then get the corresponding name of the customer in our HashMap. This method performs better than previous one, and only takes 23530+1060 = 24590ms