**Big Data Management - CS585 - Project 1**

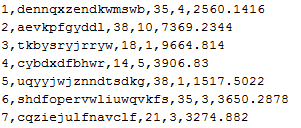
* **Azharuddin Priyotomo**
* **Congyuan Tang**

We are working under the virtual box (Ubuntu OS) and hadoop environment that the Professor provided us.

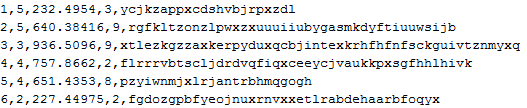
1. **Creating Datasets**

We generate the name of the customers and description of the transactions using random lowercase letters (a-z). For customer data that we created, the file has size of 1.7 MB (50,000 records), while for transaction data, the file has size of 291.68 MB (5 million records).

Sample dataset for Customer:



Sample dataset for Transaction:

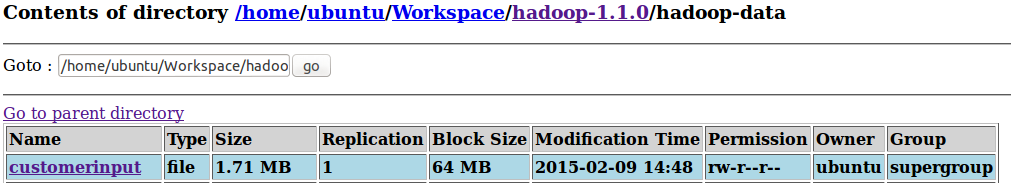


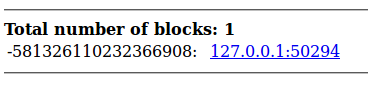
1. **Uploading into Hadoops**

For customers.txt, we use following command to upload:

hadoop fs -put /home/ubuntu/Workspace/examples/customer.txt /home/ubuntu/Workspace/hadoop-1.1.0/hadoop-data/customerinput

The uploaded file is placed into 1 block with 64 MB size:



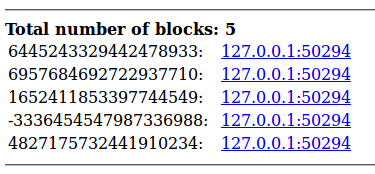


For transaction.txt, we use following command to upload:

hadoop fs -put /home/ubuntu/Workspace/examples/transaction.txt /home/ubuntu/Workspace/hadoop-1.1.0/hadoop-data/transactioninput

The uploaded file is divided into 5 blocks, each block size is 64 MB:

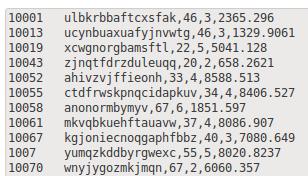




1. **Writing MapReduce Jobs**
   1. **Query 1: Reporting the customer whose CountryCode is between 2 and 6 (inclusive)**

Using our dataset, from 50,000 customers, we get 24,939 customer whose CountryCode is between 2 and 6.

Sample output:



Description: Customer ID, Customer Name, Age, Country Code, Salary

* 1. **Query 2: Reporting for every customer, the number of transactions, and the total sum of that transactions**

We create the query by using the transaction data only as our input in the mapper, since all the desired query result is available from the transaction data. The output of mapper will have the format of <customer ID, (no of transaction, amount of transaction). The number of transaction here is initially set as 1. Then in the reducer, the output of the mapper will be sorted and shuffled so that the input in the reducer will have the format of <customer ID, [(no of transaction, amount of transaction)]>. Inside the reducer, we sum up the number of the transaction and the total amount of the transaction in iteratively.

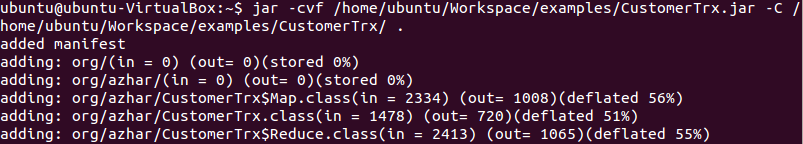
Note: There is a requirement that we should try both using combiner and without combiner. To utilize a combiner, we actually use the same function as the reducer. But one thing we have noticed to make this works is that, **the structure and type of the combiners input and output (both key and values) should always be the same of the structure and type of the mapper's output**. So in this case, since we use the reducer as the combiner as well, we make sure in a way that the reducer input and output have the same type and structure with the mapper's output.

1. First we compile and run the code which does not utilize combiner:

Compiling the source code:



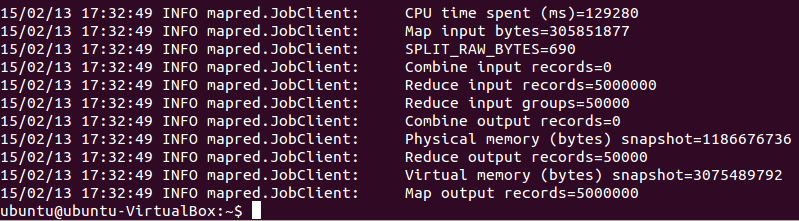
Creating the JAR file:



Running the JAR:



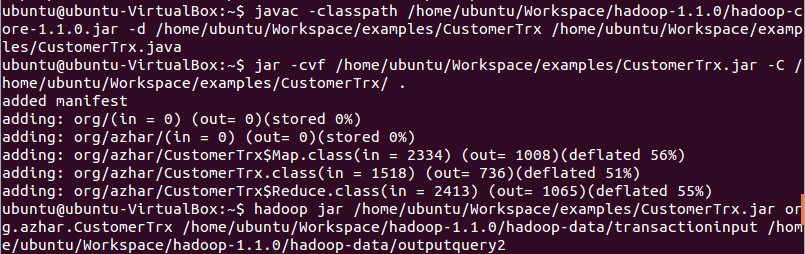
Output end:



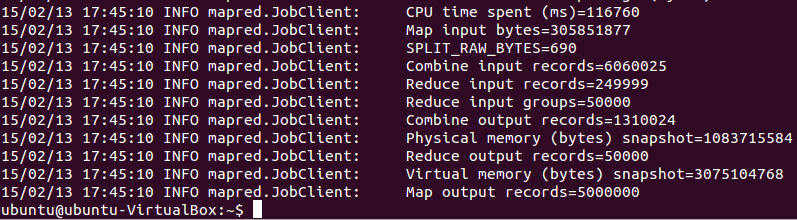
Notice that the input records for the reducer without using combiner are **5,000,000 records** and the CPU time spent is **129280 ms**.

Next we will run the same code but now we uncomment the "conf.setCombinerClass(Reduce.class);" line.

We recompile the source code & recreate the JAR file, and run again:



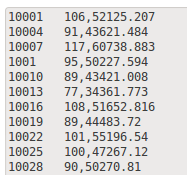
Here is the output end:



Note that using combiner, the input records for the reducer is **249999 records** the CPU time spent now is **116760 ms**.

So we can see that using combiner in this particular case will reduce the CPU time spent to run the query. Using the combiner, CPU time consumed is reduced to **9%**  than before, so we can conclude that the performance of this query is increased when we use the combiner.

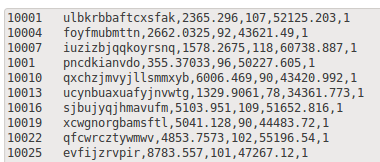
Sample output of the query 2:



Description: Customer ID, No. of transactions, Total Sum

* 1. **Query 3: Reporting joins of customers and transactions based on customer ID**

Sample output:



Description: Customer ID, Customer Name, Salary, No. of Transactions, Total Sum, Min Items

* 1. **Query 4: Reporting for every country code, the number of customers having this code as well as the min and max of transaction total for the transactions done by those customers**

We perform this query by creating map side join which will join the customer data set and the transaction data based on the customer ID. We join both of the data set and then write the output in the format of <country\_code, cust\_id, transaction\_amount>. This mapper output will be sorted, shuffled and transformed into input at the reducer in the format of: (country\_code, [(cust\_id, transaction\_amount)]. So the reducer will have 10 output since there are only 10 country code. Inside the reducer, we sum up the transaction amount of each customer. We use hashmap to store the customer and transaction amount because it will help us to quickly search and compare value, as well as searching the minimum and the maximum transaction.

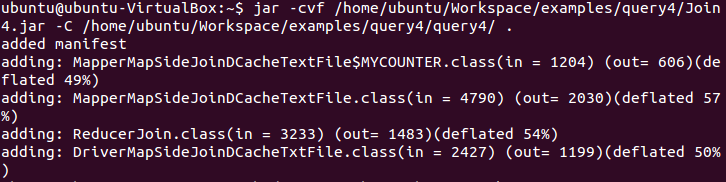
There are 3 java source files for this query:

1. MapperMapSideJoinDCacheTextFile.java: the map side join code
2. ReducerJoin.java: the reducer code
3. DriverMapSideJoinDCacheTxtFile.java: the code that contains the main function

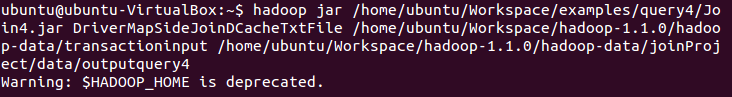
Compile all those 3 java files:



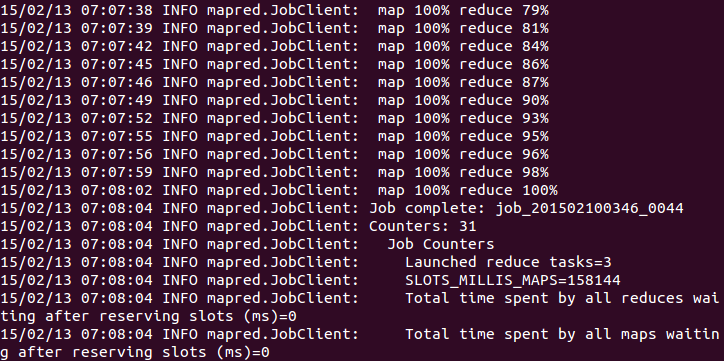
Create the JAR:

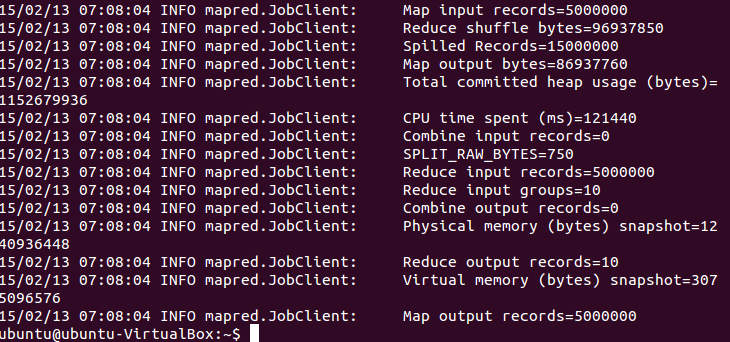


Command to run the query:

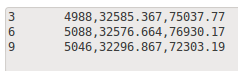
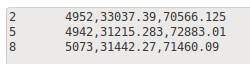


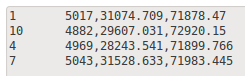
Job execution process:





Output:





Description: Country Code, Number of Customers, Minimum Transaction, Maximum Transaction