CS561 - Project 1

Group 3

Aniket Prafulla Kadam

Xinyu Dai

Documentation

1. Creating Dataset:

a. Developed java code, CreateDataset.java, to write two files - Customer.txt and Transaction.txt

b. Used ‘java.util.concurrent.ThreadLocalRandom’ to create random numeric values

c. Used ‘org.apache.commons.lang.RandomStringUtils’ to create random String values

d. Checking average number of transactions for each customer in Transaction.txt:

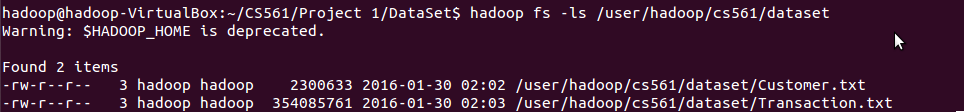




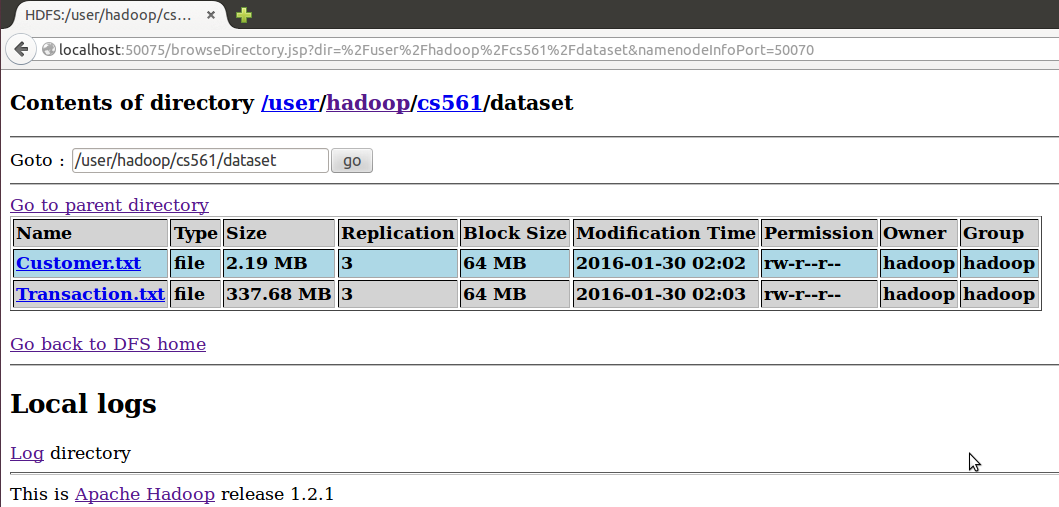
2. Loading the data on HDFS:







Viewing Block Details and Replication Through Web Interface:

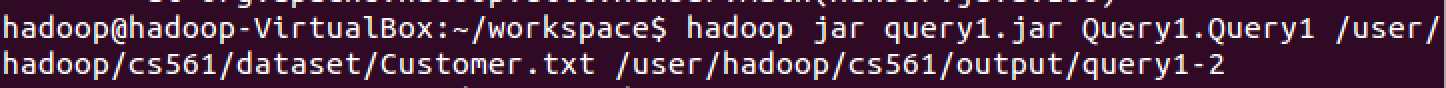


3. Design decisions and Project issues:

**Task 3.1**

The Query 1 is a map-only job which uses the CountryCode as the key. The job would output all information for customers whose CountryCode are between 2 and 6(inclusive).

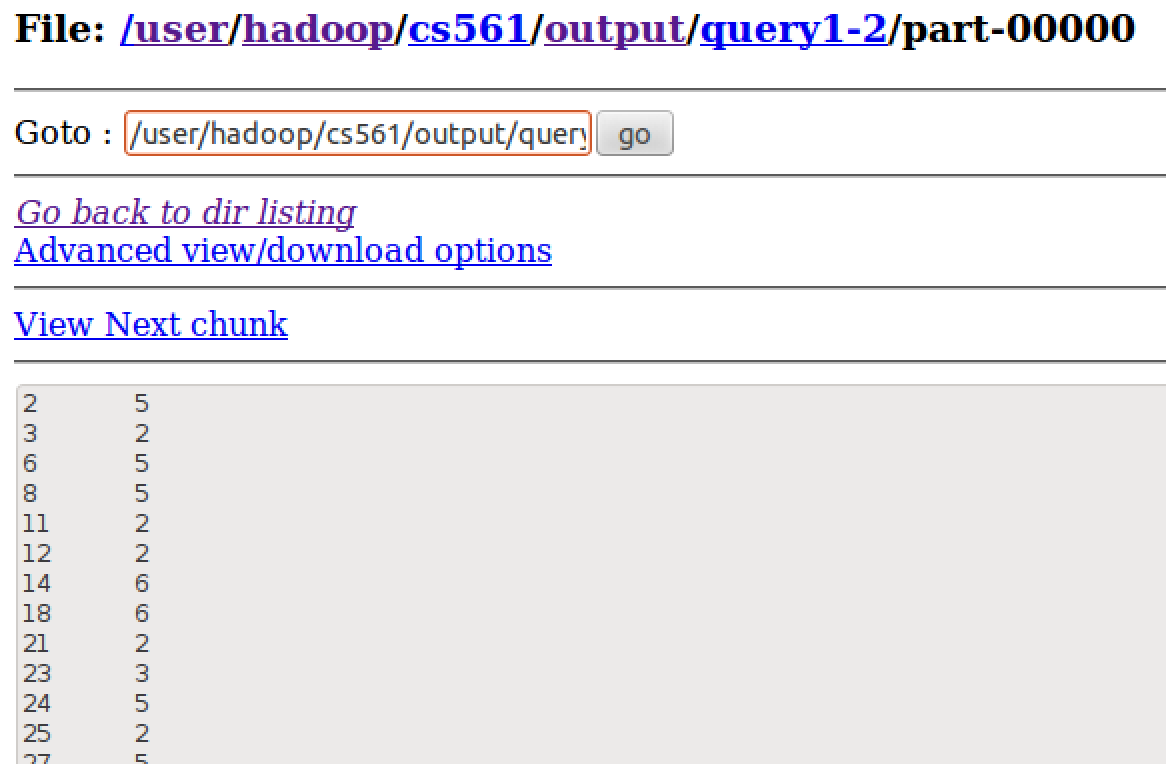
This is the running command line to run the query1.jar in hadoop:



The result is that there are 25057 customers whose CountryCode are between 2 and 6(inclusive).



The output stored on HDFS is as follows(snapshot): (with the ID as key and the attributes as values)

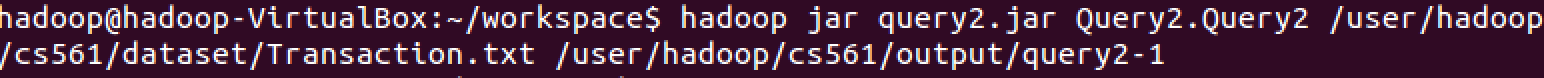


**Task 3.2**

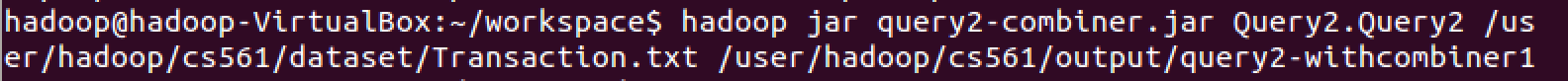
The Query 2 is a map-reduce job and the main goal is to sum TransTotal based on CustomerID. The mapper is responsible for selecting the two columns from the raw dataset ‘Transaction.txt” and outputs “CustomerID-<1, TransTotal>” key-value pairs which is the input for the reducer. As in the shuffling and sort phase it has already put the key-value pairs with the same key into one reducer, we sum up the first and the second column of the values for each record separately as the output value “NumTransactions” and“TotalSum” for each key.

To figure out the different performance with and without the combiner, we set up two jar file. The first design is the query 2 without the combiner and the running command line to run query2(without combiner) is as follows:

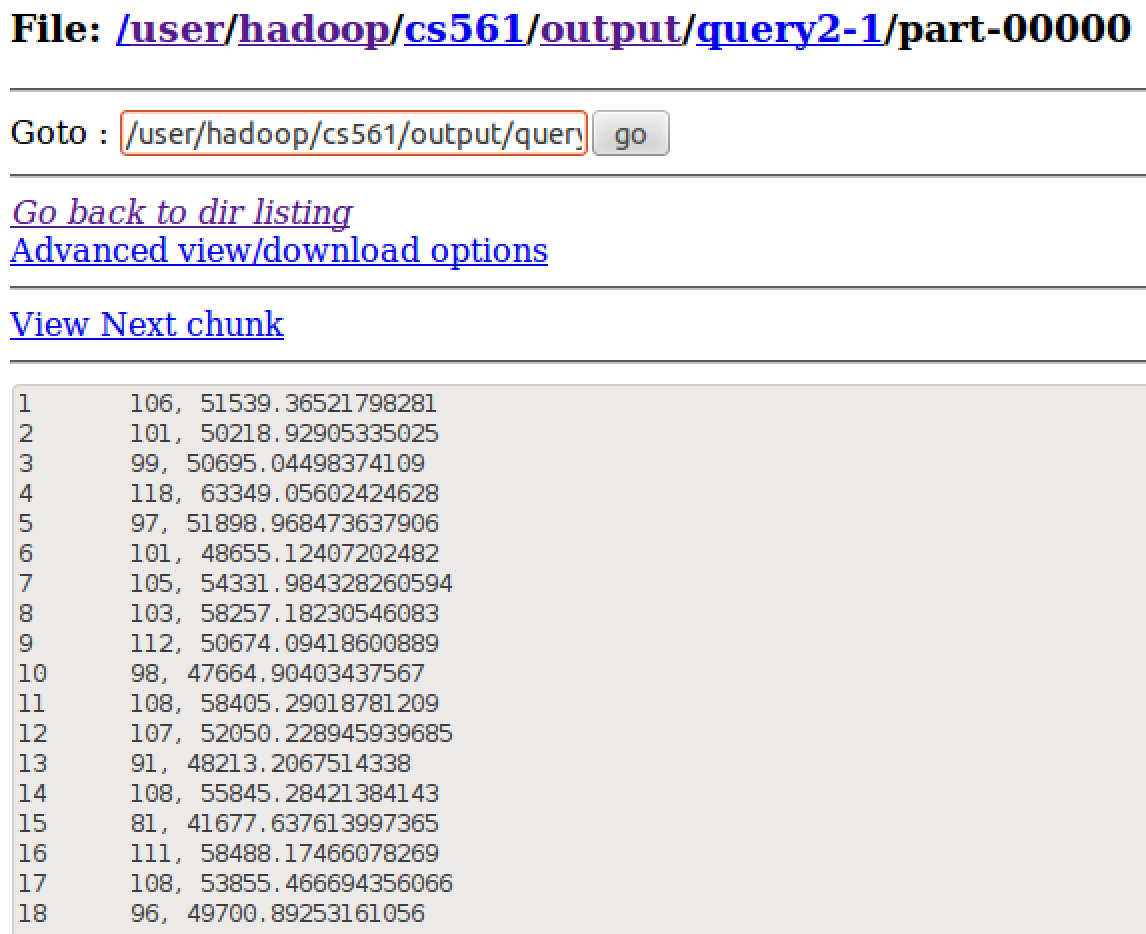
The running command line to run query 2(with combiner) is as follows:



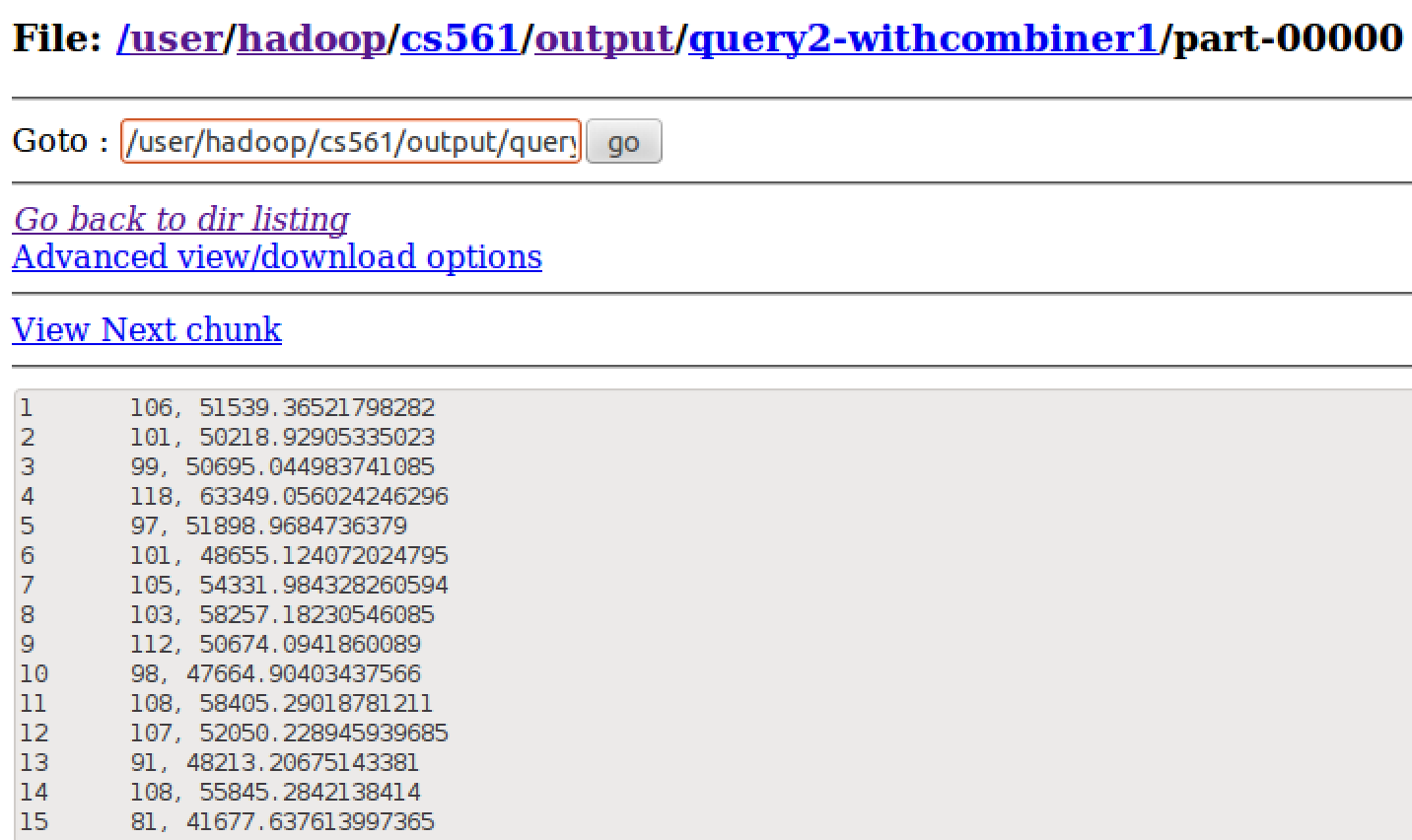
The running command line to run query 2(with combiner) is as follows:



The outputs of query2 with and without a combiner are the same, however, the first program **without combiner took 45 seconds** to finish the mapreduce job which the second program **with the combiner took 39 seconds**. We could tell from this task that we can save the process time while using the combiner if possible. The following figures are two snapshots of the output of query2(without and with combiner in order):



(Snapshot for query 2 without a combiner)



(Snapshot for query 2 with a combiner)

**Task 3.3**

Approach 1:

* 2 Mappers: 1 for each file
* 1 Reducer with cleanup: Reducer performs aggregation tasks and writes the data to the sorted Maps
* Cleanup: Adds the datasets to the sorted sets which are sorted w.r.t the keys. Joins the two datasets based on the Customer IDs. Outputs the data.
* CPU time spent (ms) = 49170

Approach 2: Using Distributed Cache in Reducer

* 1 Mapper for Transaction dataset
* Customer dataset is cached as it is small in size
* 1 Reducer with setup and cleanup functions
  + Setup:
    - Cached Customer file is read
    - Required customer data is stored in dataset
  + Cleanup:
    - Loads the 2 datasets in sorted sets that are sorted w.r.t keys
    - Datasets are joined by customer ID
    - Output is written to HDFS
* CPU time spent (ms) = 49900

Note: Cleanup functions and overall join method is same in both the approaches

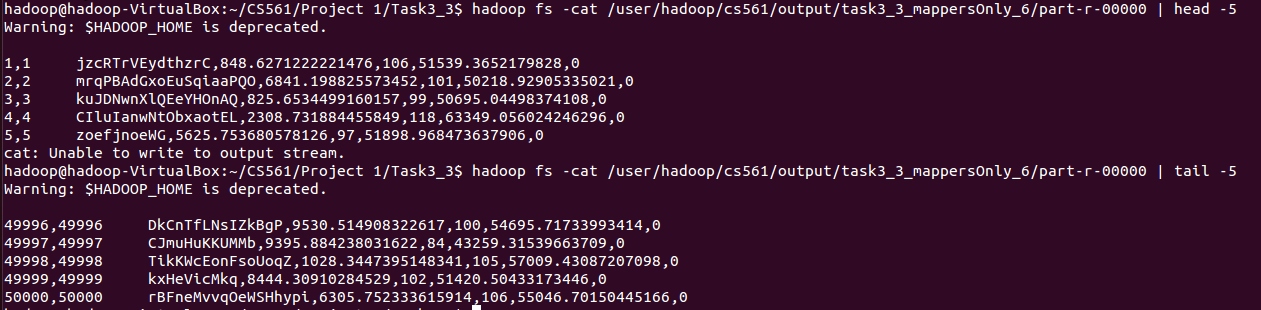
Observations:

Approach 2 with Distributed Cache took marginally more CPU time as compared to the the approach 1. This could mean that the time required to read the file through Mapper is more than the time required to cache the file. However it is also evident that if the cached file is used multiple times in the code, then the caching can be beneficial instead of reading the file through a separate mapper.

Important Optimization Technique:

* Normally join would require two loops, one for each dataset
* In this approach the information regarding the dataset is utilized to optimize the execution
* We know that the aggregated transaction dataset is w.r.t customer ID. We also know the number of unique customer IDs (50000 in this case). Thus both the sorted sets would have the same number of records. Moreover both the datasets are sorted on keys. Thus we know that the records in both the datasets are in the order of the customer ID. Thus 1st record in Transaction sortedset would have key = 1 and the 1st record in Customer sortedset would also have key = 1. Same way 10th records in both the sets would also have same key values = 10 and so on for all the records.
* Using the above mentioned approach, we have avoided 2 loops. We are using just 1 loop which iterates 50000 times and prints the outputs in sequence.
* Usage of This Technique:
  + When we know that the keys in the datasets to be joined, are equal in number ( also when the same key is used for joining) we can sort the datasets on the keys and use this technique. Note that when the number of keys in the two datasets is not known or known to be unequal this method would not work and we would need to use 2 loops, 1 for each dataset.
* Performance Analysis:
  + If we use two loops then the complexity of the join operation would be O(n^2). But as we know using sorting through TreeSet is inexpensive, O(log n), this kind of sorting combined with just 1 loop operation, O(n), would give better performance over O(n^2).

Verifying that the used approach gives correct output, i.e. joins correctly:

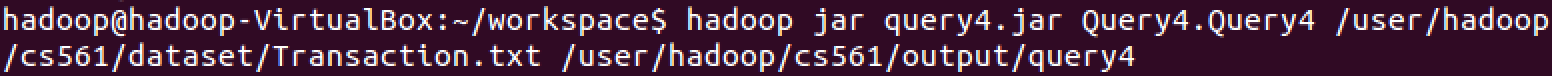


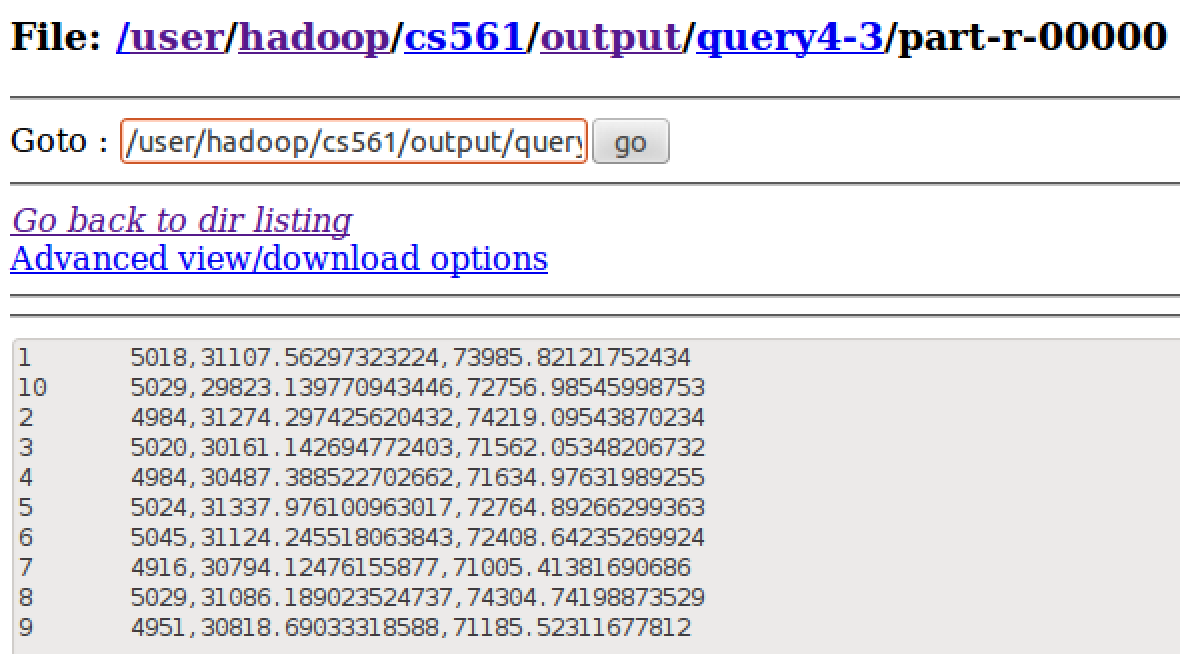
As can be seen in the above screenshot, the 1st 2 columns are the Customer IDs from Transaction and Customer dataset respectively. Both are equal for all the records (checked all output chunks as well). Thus we confirm that our approach works correctly.

**Task 3.4**

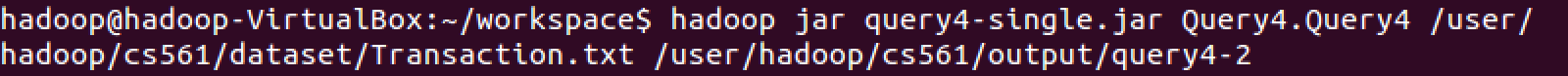
The query 4 is a map-reduce job to do the join based on Customer ID and sum up the number of customers based on the CountryCode while finding the min and max amount of TransTotal in each CountryCode group. In order to finish the query in one mapreduce job, we could do the join in the mapper. Because the Customer.txt file is only 2.19MB which is far less than the size of a block, we put the Customer.txt file in the DistributedCache and in the map we store the Customer.txt file into the hashmap. After joining by the Customer ID, we output the map result with (CountryCode, <CustID, TransTotal>) as key-value pairs into the reducer. In the reducer, we created a hashmap for each CountryCode key. If the hashmap contains the key CustID, we sum up the value of TransTotal of that key. If the hashmap doesn’t contain the CustID, we add the CustID and the value of TransTotal to the hashmap and add the number of customers. After reading in all the records for the reducer key CountryCode, we find the maximum and minimum amount of the sum of TransTotal of each CustomerID for the key.

The running command line for query 4 is as follows:

The output of the reducer in query 4 is as follows:



If we consider the min and max of TransTotal as each transaction for each customer, we would compare the amount of TransTotal with the current Min and Max under the same CountryCode. If it is smaller or greater than the current value, we would renew the value of Min and Max. The command line for single transaction comparison is as follow:



The output of this single transaction comparison is as follow:



**Task 3.5**

As observed in Task 3.3, Separate Mapper works better in our specific join case rather than using distributed cache, in this task only Separate Mapper method is implemented

* Design:

1. 2 Mappers, 1 for each input file: Customer.txt and Transaction.txt and 1 Reducer with cleanUp method
2. TransMapClass (Mapper for Transaction.txt) appends the Customer ID from Transaction.txt with ‘T’ to form the key: custID+"T"
3. CustMapClass (Mapper for Customer.txt) appends the customer ID from Customer.txt with ‘C’ to form the key: custID+"C"
4. Reducer along with cleanUp method performs the following tasks:
   1. For the data coming from Transaction.txt, it calculates the sum of the number of transactions for each customer ID
   2. Sorted set, sortedTransSet, is used to hold the customer ID and number of transactions for each customer in the sorted order w.r.t the number of transactions
   3. Sorted set, sortedCustSet, is used to hold the customer ID and customer name from the Customer.txt file in sorted order w.r.t the customer ID
   4. In cleanup method the customer ID with the least number of sum of transactions is calculated and stored in a SortedMap
   5. Using iterators over the sorted set of customer ID and Name and over the map containing the least transaction count customer records, we join the 2 datasets and output the customer name and count of transactions when the customer IDs from both the datasets match.
5. Optimization:

As the sortedCustSet is sorted w.r.t the customer ID and as we also know that the Customer file has only 1 record per customer, we don’t need to compare the entire data set and we can stop once the customer IDs match for any record. Further possible optimization could be to use binary search to optimize the join process

* Stats:

1. Number of Mappers: 2
2. Number of Reducers: 1
3. CPU time spent (ms)=34400

* Output: (there is only 1 customer with least number of transactions, hence the below o/p. The code supports if more than one customers have same number of least transactions

HILkdndFWe 60

Contact information:

If you have any question, please feel free to contact at:

Xinyu Dai [xdai4@wpi.edu](mailto:xdai4@wpi.edu)

Aniket Prafulla Kadam [apkadam@wpi.edu](mailto:apkadam@wpi.edu)