DS 503 - Project 1

By :
Abdulaziz Alajaji - asalajaji@wpi.edu
Yousef Fadila - yousef@fadila.net

Question 1 : Creating Datasets

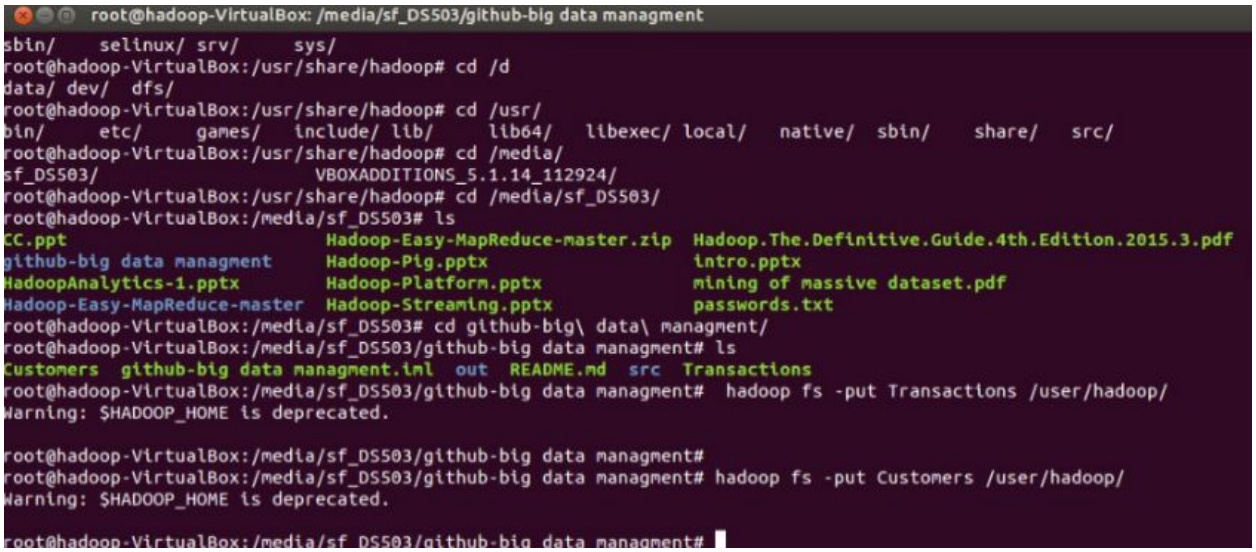
The main class to create the Datasets is DataGeneratorMain.java.
The code is very short and straightforward

```
package edu.wpi.ds503;

public class DataGeneratorMain {

    public static void main(String[] args) {

        DataGenerator[] dataGenerators = new DataGenerator[]{ new CustomersGenerator(50000),
                new TransactionsGenerator(5000000) };

        for (DataGenerator dataGenerator : dataGenerators) {
            dataGenerator.generate();
        }
    }
}
```

Both CustomersGenerator and TransactionsGenerator classes extend DataGenerator abstract class which has abstract method named: “generate()”.
The Generator constructor receive number of records as an argument in order to allow creating small samples.

Question 2- uploading Data into Hadoop

Uploading the data from local file system to HDFS was done using the put command:

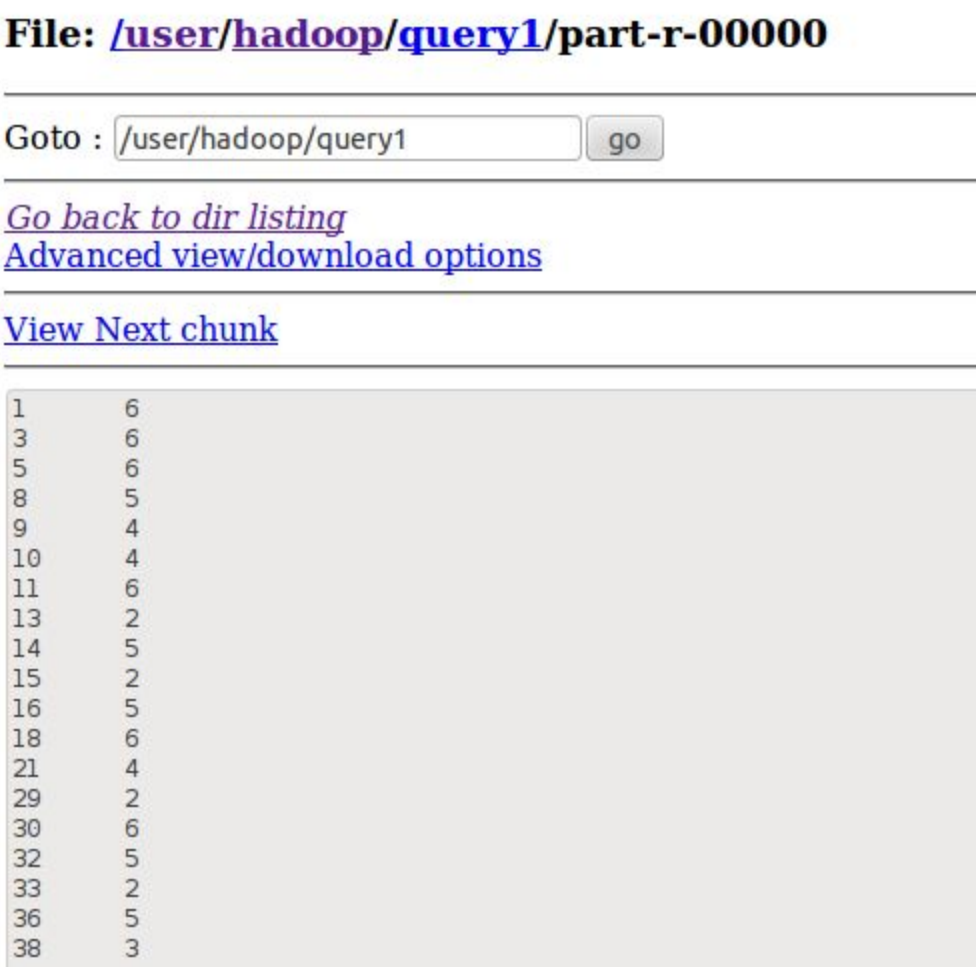


3.1) Query 1

Query 1 was implemented using map only job
The command to run the job:

hadoop jar hw1.jar edu.wpi.ds503.hadoop.Query1 /user/hadoop/Customers /user/hadoop/query1

Screenshot of the output:

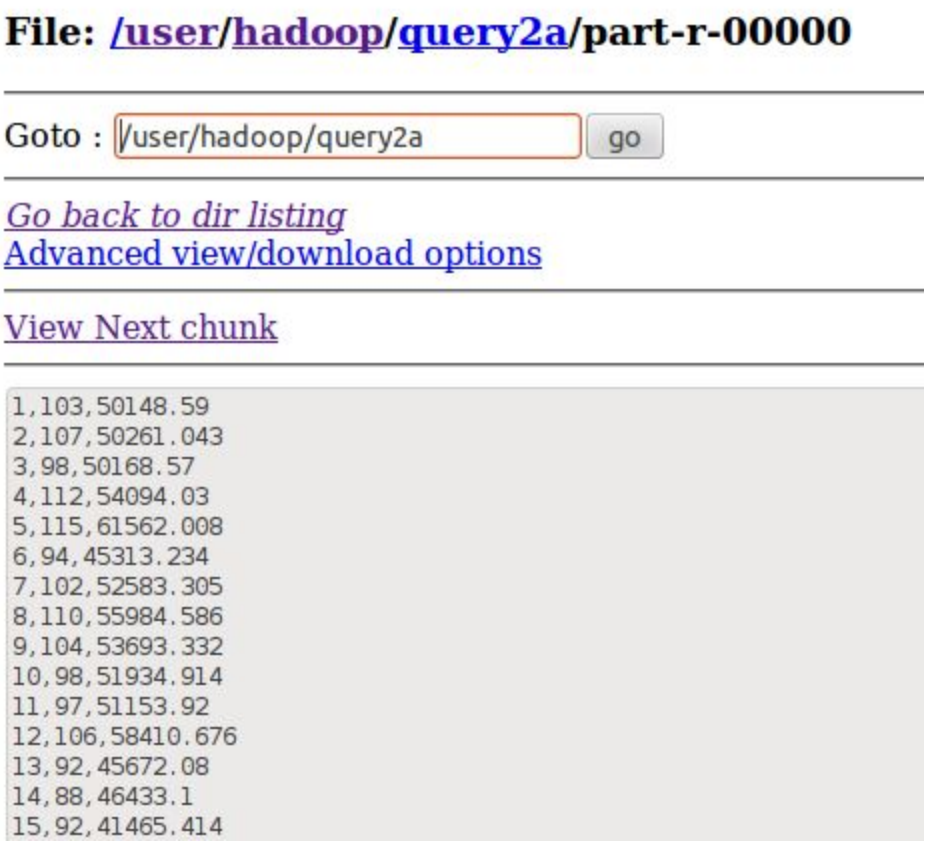


3.2) Query 2

Query2 was implemented with single map-reduce job with combiner.

The command to run the job:
hadoop jar hw1.jar edu.wpi.ds503.hadoop.Query2 /user/hadoop/Transactions /user/hadoop/query2

Screenshot of the output:



We define new writeable object **"CustomerQuery2Data"** to be passed between Map, Combiner and reducer.

CustomerQuery2Data {
    Count;
    Sum;
}

The Mapper will set count to 1 and sum to transTotal for each single transaction
The Combiner will set count and sum to the aggregated values for each customer.
The reducer has the same functionality like the Combiner except that it produces Text, not CustomerQuery2Data

---

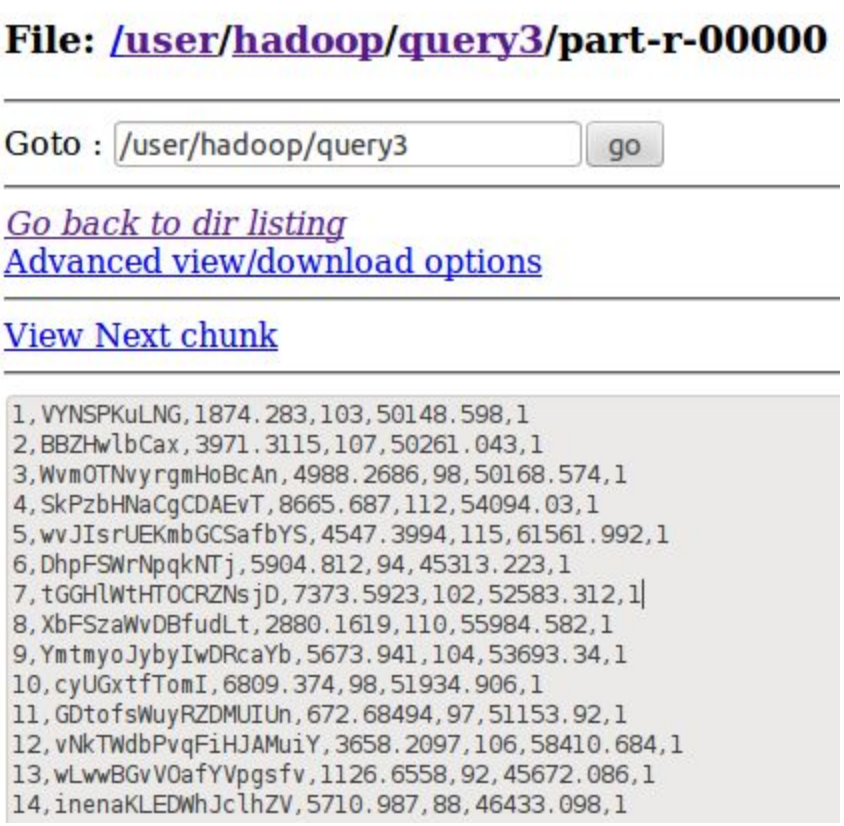
3.3) Query 3

Query3 was implemented with 2 mapers with 1 reducer job.

The command to run the job:

hadoop jar hw1.jar edu.wpi.ds503.hadoop.Query3 /user/hadoop/Customers /user/hadoop/Transactions /user/hadoop/query3

Screenshot of the output:



We define new writeable object **“Query3Data”** to be passed between the 2 Mapers and the reducer.

Query3Data {
- **name**
- **salary**
- **transTotal**
- **transItems**

}

Customers Mapper will fill in name and salary and writes the object with CustomerId key.
Transaction Mapper will fill in transTotal and transItems and writes the object with CustomerId key

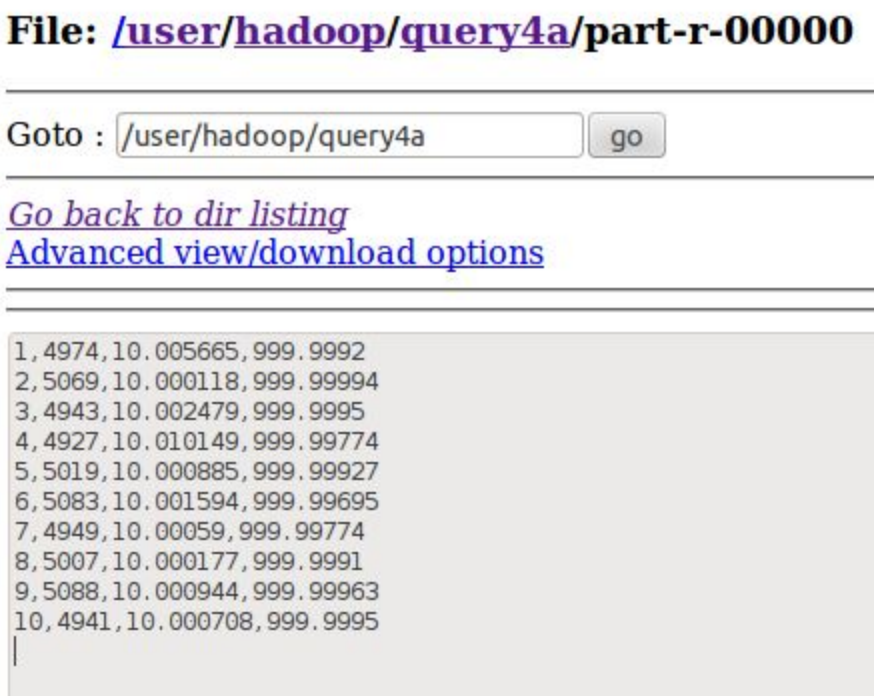
In the reducer, the indicator whether the data comes from Transactions or Customers is **if name is defined or no**t! There is no need to define another flag.

3.4) Query 4

Query4 was implemented with single map-reduce job.

The command to run the job:
hadoop jar hw1.jar edu.wpi.ds503.hadoop.Query4 /user/hadoop/Customers /user/hadoop/Transactions /user/hadoop/query4

Screenshot of the output:



In order to implement the query with single map-reducer job, motivated by the fact that the customers files is relatively small. We decide to build a lookup table in the setup that maps each customer to country code.

```
protected void setup(Context context) throws IOException, InterruptedException {
    Configuration conf = context.getConfiguration();
    String customerPath = conf.get(CUSTOMER_PATH);
    try {
        Path pt = new Path(customerPath);
        FileSystem fs = FileSystem.get(new Configuration());
        BufferedReader br = new BufferedReader(new InputStreamReader(fs.open(pt)));
        String line;
        line = br.readLine();
        while (line != null) {
            String[] fields = line.toString().split(",");
            int customerId=Integer.parseInt(fields[CUST_CUSTOMER_ID]);
            int countryCode=Integer.parseInt(fields[CUST_CONTRY]);
            custToCountryMap.put(customerId, countryCode);
            countryCustomers[countryCode - 1] = countryCustomers[countryCode - 1] + 1 ;
            line = br.readLine();
        }

    } catch (Exception e) {
        throw new IOException("UNEXPECTED,error",e);
    }
}
```

The disadvantage of this approach is that each mapper has to read the customer's file to build this lookup table. The advantage is that it saves another map-reducer job to do the join based on customer Id.

Based on the fact that customers file size is less than 2MB, we favor to pay the penalty of each mapper has to read this 2 mb file instead of doing double map-reduce job to do the join.
Since we already read the file, line by line, and the number of countries is limited to 10. We choose to count customers per country during the setup as the cost of counting the customer is negligible in both space and time.(done in the same loop that make the look-up table)

3.5) Query 5

Query5 was implemented with 2 map-reduce jobs followed by map only job.

The command to run the job:

hadoop jar hw1.jar edu.wpi.ds503.hadoop.Query5 /user/hadoop/Customers /user/hadoop/Transactions /user/hadoop/query5

Screenshot of the output:

Contents of directory /user/hadoop/query5

Goto : /user/hadoop/query5 go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time
final	dir				2017-02-10 20:33
step1	dir				2017-02-10 20:32
step2	dir				2017-02-10 20:32

Go back to DFS home

Final result screenshot
The output is the name, Trans number; we print the Trans number as validation to manually check if all reported customers will be higher than output from step2

File: /user/hadoop/query5/final/part-r-00000

Goto : /user/hadoop/query5/final go

Go back to dir listing
Advanced view/download options

View Next chunk

```
VYNSPKuLNG,103
BBZHwlbCax,107
SkPzbHNaCgCDAEvT,112
wvJIsrUEKmbGCSafbYS,115
tGGHlWtHTOCRZNsjD,102
XbFSzaWvDBfudLt,110
YmtmyoJybyIwDRcaYb,104
vNkTWdbPvqFiHJAMuiY,106
VjmTKtrozk,105
aqqpfImCwOToPBwddb,105
aZsmOulVoNLwRes,105
sbOCQHmCuqAwJyx,104
```

The first map-reduce job of this question is running query 2.

```
Job job = Job.getInstance(conf, "Project1, Query5 - step1");
job.setJarByClass(Query5.class);

// first step is running Query2.
job.setMapperClass(Query2.CustomersQuery2Mapper.class);
job.setCombinerClass(Query2.CustomersQuery2Combiner.class);
job.setReducerClass(Query2.CustomersQuery2Reducer.class);

job.setOutputKeyClass(NullWritable.class);
job.setOutputValueClass(Text.class);

job.setMapOutputKeyClass(IntWritable.class);
job.setMapOutputValueClass(Query2.CustomerQuery2Data.class);
FileInputFormat.addInputPath(job, new Path(args[1]));
FileOutputFormat.setOutputPath(job, new Path(args[2] + STEP1_OUTPUT));
```

That because in query 2 we count the transaction per customer.

The 2nd step is calculating the average; done by map-combiner-reducer job. We define new writeable object:

Query5Data {
Count;
Average;
}

The mapper will go over the result from query 2. And fill in this object with count =1 ; average=total trans

```
public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    String[] fields = value.toString().split(",");
    // 19,105,55984.484 (query2 output is CustomerId, TotalTrans,Sum
    float transNum=Float.parseFloat(fields[TRANS_NUM]);
    FloatWritable transNumFloatWriteable = new FloatWritable(transNum);
    context.write(NullWritable.get(), new Step2Query5Data().set(one, transNumFloatWriteable));
}
```

The combiner will aggregate the count and calculate the new average.

```
public static class Step2Query5Combiner
        extends Reducer<NullWritable,Step2Query5Data, NullWritable,Step2Query5Data> {

    public void reduce(NullWritable key, Iterable<Step2Query5Data> values, Context context)
            throws IOException, InterruptedException {
        float sum = 0;
        int count = 0;

        for (Step2Query5Data val : values) {
            float average = val.getAverage().get();
            int _count = val.getCount().get();

            sum += average * _count;
            count +=val.getCount().get();
        }

        float newAverage = sum/count;
        context.write(key, new Step2Query5Data().set(new IntWritable(count),
                new FloatWritable(newAverage)));
    }
}
```

And finally, the reducer will calculate the global average

```
public static class Step2Query5Reducer
        extends Reducer<NullWritable,Step2Query5Data,NullWritable,Text> {

    public void reduce(NullWritable key, Iterable<Step2Query5Data> values, Context context)
            throws IOException, InterruptedException {
        float sum = 0;
        int count = 0;

        for (Step2Query5Data val : values) {
            float average = val.getAverage().get();
            int _count = val.getCount().get();

            sum += average * _count;
            count +=val.getCount().get();
        }

        float newAverage = sum/count;
        // print count for validation
        Text result = new Text(count + "," + newAverage);
        context.write(NullWritable.get(), result);
    }
}
```

The 3rd step is map only task.
In the setup of the map we create a lookup table to map customerId to customer name.
And we read the global average from step 2

```
protected void setup(Context context) throws IOException, InterruptedException {
    Configuration conf = context.getConfiguration();
    String customerPath = conf.get(CUSTOMER_PATH);
    String step2OutputPath = conf.get(STEP2_PATH);
    try {
        Path pt = new Path(customerPath);
        FileSystem fs = FileSystem.get(new Configuration());
        BufferedReader br = new BufferedReader(new InputStreamReader(fs.open(pt)));
        String line;
        line = br.readLine();
        while (line != null) {
            String[] fields = line.toString().split(",");
            int customerId=Integer.parseInt(fields[CUST_CUSTOMER_ID]);
            custIdToNameMap.put(customerId, fields[CUST_NAME]);
            line = br.readLine();
        }
        // read the one line of the putput of step 2. format COUNT,AVERAGE.
        pt= new Path(step2OutputPath);
        br = new BufferedReader(new InputStreamReader(fs.open(pt)));
        line = br.readLine();
        String[] fields = line.toString().split(",");
        global_average = Float.parseFloat(fields[1]);
    } catch (Exception e) {
        throw new IOException("UNEXPECTED,error",e);
    }
}
```

The mapper of this step is very simple, it goes over the step1 (query 2 result) and filter out all customers have less or equal to the global average

```
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String[] fields = value.toString().split(",");

        // 19,105,55984.484 (query2 output is CustomerId, TotalTrans,Sum
        int customerId=Integer.parseInt(fields[CUST_CUSTOMER_ID]);
        int transNum=Integer.parseInt(fields[TRANS_NUM]);

        if (transNum > global_average) {
            String name = custIdToNameMap.get(customerId);
            if (name == null) {
                throw new IOException("UNEXPECTED, customer is not fount");
            }
            // print the transNum to validate that all of customers will be higher than output from step2
            context.write(NullWritable.get(), new Text(name + "," + transNum));
        }
    }
}
```

4-Writing Apache-Pig Jobs

4.1) Query 1

We first load the customers/transactions files using LOAD command, and then join them on ID & CustID fields using JOIN command. To get aggregation statistics, we have to perform GROUP over the JOINED relation. And To get number of Transactions for each customer we used Count(<NAME_OF_JOINED_RELATION>). This is similar to COUNT(*) in SQL.

And In order to get the customer with the least number of transactions. We have to first store the minimum number of transactions, and we have done this by (ordering the results by number of transactions, and taking the first row. We tried to find a way to do this in the same operation, but we couldn't find one. Now In order to get the needed result, we just need to FILTER records, and evaluate "the number of transactions" to be equal the minimum number c.

4.1) Query 2

To perform the replicated join, we only needed to add the phrase "replicated" after the join operation. We put the larger relation first, e.g. when we write JOIN A by $1 , B by $1. A is the large relation.

To get the needed aggregations :

E = foreach D generate group,C.NAME,C.SALARY,COUNT(C) AS NumOfTransactions,SUM(C.TransTotal) as TotalSum, MIN(C.TransNumItems) as MinItems;

It's very straightforward.

4.1) Query 3

To solve this problem, we basically group customers on Country_Code, and then find how many customers associated with that country code by looking at COUNT(customers) in each group (each country code). Then we filter by the required condition < 5000 or < 2000. When we tried to run the query, we didn't get any country code with less than 2000. We might justify this to the huge number of customers (50,000), and assigning country codes with a uniform randomness.