Big Data Processing Coursework

Ethereum Analysis

Part A: Time Analysis

This part is implemented in two files. First created the list of the transactions per month of each year. The second file created a bar plot using Excel.

Hadoop JobID: 1637

http://andromeda.student.eecs.qmul.ac.uk:18088/history/application_1575381276332_1637/jobs/

Programming:

```
import pyspark
import time
sc = pyspark.SparkContext()
#we will use this function later in our filter transformation
def is_good_trans(trans):
   #checks if the record has missing columns
   try:
      fields = trans.split(',')#split the records into fields
      if len(fields)!=7:
         return False
      float(fields[6])
      return True
   except:
      return False
trans=sc.textFile('/data/ethereum/transactions')
clean_trans=trans.filter(is_good_trans)#filter the transactions
timestamp=clean_trans.map(lambda t:int(t.split(',')[6]))#split the clean transactions
monthyears=timestamp.map(lambda my: (time.strftime("%B-%Y",time.gmtime(my)),1))#create key value pair for month & year
transactions=monthyears.reduceByKey(lambda a,b: a+b)#add up the no. of transactions per month & year
inmem=transactions.persist()
inmem.saveAsTextFile("/user/mm347/PARTtA_Output")
```

Explanation:

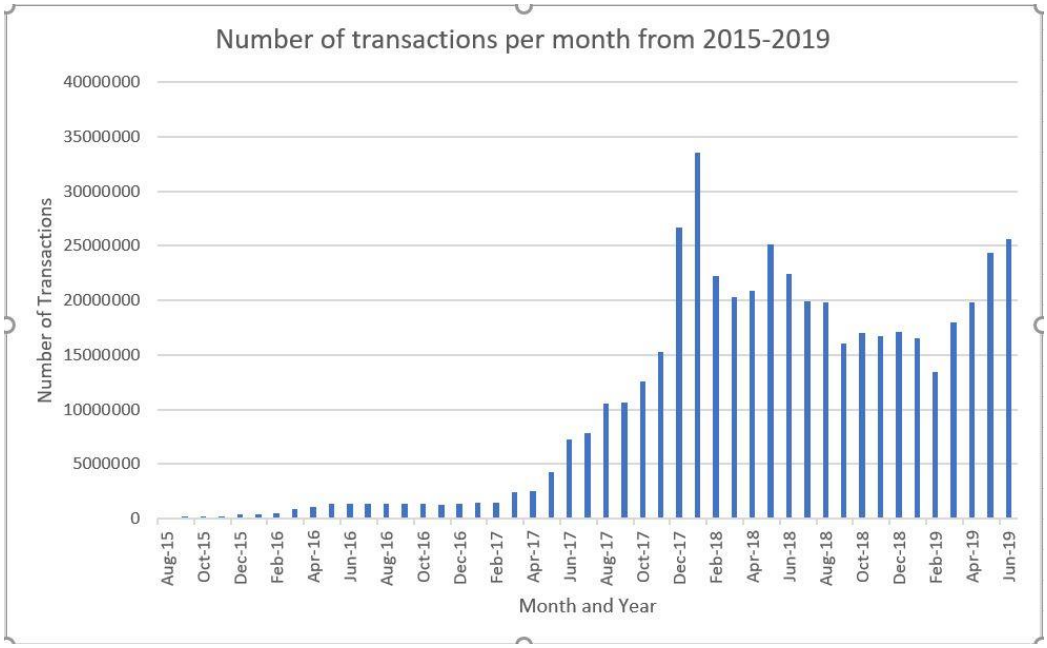
- In this program we use transaction dataset. It contains block_number, value, from_address and to_address fields.
- A named function is defined to be ran on each element(is_good_trans). This function takes one element which will correspond to a transaction.
- Monthyears are obtained. It is formatted in Month- Year form.monthyears are generated as a key with value as 1.
- Transactions are obtained from monthyears using reduceByKey() and adding up all the values for each key.After assigning values to address,added up the values for each address.
- Perform the operation on HDFS and merge the output to get it in local.

PartAOut - Notepad

File Edit Format View Help

```
('October-2015', 205045)
('October-2016', 1329847)
('October-2017', 12570063)
('October-2018', 17056926)
('March-2017', 2426471)
('March-2016', 917170)
('March-2019', 18029582)
('March-2018', 20261862)
('February-2019', 13413899)
('February-2018', 22231978)
('February-2017', 1410048)
('February-2016', 520040)
('August-2016', 1405743)
('August-2017', 10523178)
('August-2015', 85609)
('August-2018', 19842059)
('July-2017', 7835875)
('July-2016', 1356907)
('July-2018', 19937033)
('May-2019', 24332475)
('May-2018', 25105717)
('April-2016', 1023096)
('May-2017', 4245516)
('April-2017', 2539966)
('May-2016', 1346796)
('April-2018', 20876642)
('April-2019', 19830158)
('January-2016', 404816)
```

➢ Below graph is obtained as a result of the program.



It can be seen above there were only few thousand transactions in the year 2015 .Transactions increased in the year 2017 by many folds, however still in the range of thousands. Peak value is observed for the month December 2017. Hence overall, there were only little transaction in the year 2015, then increased 2017.

Part B: TOP TEN MOST POPULAR SERVICES

JOB 1 - INITIAL AGGREGATION

Hadoop Job ID:

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2324

DURATION 39mins, 52sec

```
class PartBJob1(MRJob):
    def mapper(self, _, trans):
        try:
            fields = trans.split(',')
            if len(fields) == 7 :#check if the transactions are clean
                address=fields[2]#assigning transaction values
                value=int(fields[3])
                yield(address,value)# giving the address and its value
        except:
            pass
    def combiner(self, address, value):
        yield(address,sum(value))#Adding up the values for each address
    def reducer(self, address, value):
        yield(address,sum(value))#Adding up the values for each address
#this part of the python script tells to actually run the defined MapReduce job. Note that Lab1 is the name of the class
```

Explanation:

- As given, in order to calculate the aggregation of transactions the value field is aggregated for addresses in the to_address field from transactions file
  • In mapper the lines are split by comma and the key and values are yielded only if the values field is not zero. This will remove all the lines that are not required for computation, thus saving memory and improving the execution performance of the job.
  • A combiner is used to calculate the sum of values for each transaction present in each mapper. Adding a combiner improve the aggregation performance of the job.
  • Finally, in the reducer the same sum operation is used to calculate the aggregate of transaction values.

JOB 2 - JOINING TRANSACTIONS/CONTRACTS AND FILTERING

Hadoop Job ID:

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2555
DURATION 9mins, 4sec

```
class PartB_Job2(MRJob):
  def mapper(self, _, line):
    try:
      if len(line.split(','))==5:#check if the transactions are clean
        #this should be the contracts dataset
        fields = line.split(',')
        jkey = fields[0]
        jvalue = int(fields[3])
        yield (jkey,(jvalue,1))
      #one mapper,we need to first differentiate among both types
      if len(line.split('\t')) ==2:
        #this should be the transactions aggregate dataset. Output from job 1
        fields = line.split('\t')
        jkey = fields[0]
        jkey = jkey[1:-1]
        jvalue = int(fields[1])
        yield (jkey, (jvalue,2))

    except:
      pass
  def reducer(self, address, values):
    block_number = 0
    counts = 0
    for value in values:
      if value[1] == 1:
        block_number = value[0]
      if value[1] == 2:
        counts = value[0]
      if block_number > 0 and counts > 0
   yield(address, counts)
```

Explanation:

- Map reduce job to perform repartition join between contracts dataset and transaction aggregate dataset (output from job 1).

• In the mapper we are differentiating the two input files by checking the number of fields present in the file. If the number of fields is 5 its contracts dataset and of number of fields is 2 its transactions aggregate dataset (output from job 1).

• The first if condition in mapper recognises the contracts dataset where we specify key as address(fields[0]) and value as block_number and '1'. 1 is hardcoded to identify that the value is from contracts dataset at the reducer.

• The second if condition in mapper recognises the transactions aggregate dataset where we specify key as address and value as aggregate count .

• The mapper takes records only if both the keys matches.. By this way we filter smart contract address.Finally, the key which is the smart contract address and value which is the aggregate value is yielded in the reducer which gives us the aggregate values of all smart contracts.

JOB 3 - TOP TEN

Hadoop JobID:

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2605
Duration 29sec

```
class PartB_Job3(MRJob):

    def mapper(self, _, line):
        try:
            fields = line.split('\t')
            #one mapper,we need to first differentiate among both types
            if len(fields)==2:
                #address of a company sector line
                address = fields[0][1:-2]
                count = int(fields[1])
                yield (None, (address, count))
        except:
            pass

    def combiner(self, _,values):
        sorted_values = sorted(values,reverse = True, key = lambda tup:tup[1])
        i=0
        for value in sorted_values:
            yield ("top", value)
            i += 1
            if i >= 10:
                break

    def reducer(self, _,values):
        sorted_values = sorted(values, reverse = True, key = lambda tup:tup[1])
        i = 1
        for value in sorted_values:
            yield (i, ("{} - {}".format(value[0],value[1])))

            i += 1
            if i > 10:
                break
```

Output:

```
1	"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd511644 - 84155100809965865822726776"
2	"0xfa52274dd61e1643d2205169732f29114bc240b - 45787484483189352986478805"
3	"0x7727e5113d1d161373623e5f49fd568b4f543a9 - 45620624001350712557268573"
4	"0x209c4784ab1e8183cf58ca33cb740efbf3fc18e - 43170356092262468919298969"
5	"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f - 27068921582019542499882877"
6	"0xbfc39b6f805a9e40e77291aff27aee3c96915bd - 21104195138093660050000000"
7	"0xe94b04a0fed112f3664e45adb2b8915693dd5ff - 15562398956802112254719409"
8	"0xbb9bc244d798123fde783fcc1c72d3bb8c18941 - 11983608729202893846818681"
9	"0xabbb6bebfa05aa13e908eaa492bd7a834376047 - 11706457177940895521770404"
10	"0x341e790174e3a4d35b65fdc067b6b5634a61cae - 83790007519177556240575000"
```

Explanation:

As given, the output from Job 2 will be taken as input and sort the values based on the total aggregate value to get the top 10 values

• In the mapper the lines are split by tab since the input file is tab separated. Key is None and value is the address and sum of aggregate values. Key is none because we need to sort the values based on the aggregate count.

• A combiner is used to speed up the sorting process. In combiner sort the values in ascending order by giving reverse = True. This will return the top values first. After this condition has been set a for loop is used to iterate between all the records and yield the sort the values in ascending order.

• Finally, in reducer the same combiner operation is performed where key value pairs from all the combiners are sorted once again to obtain the exact result. A for loop is used to iterate between all the records from combiner and the result is displayed in the top 10 order. In order to display only top 10 values the for loop is terminated if the iteration count reaches 11.

Part C: Comparative Evaluation

Part B is implemented in spark to compare

```
import pyspark

sc = pyspark.SparkContext()

def clean_trans(trans):
    try:
        fields = trans.split(',')
        if len(fields)!=7:
            return False
        int(fields[3])
        return True
    except:
        return False


def clean_contracts(contract):
    try:
        fields = contract.split(',')
        if len(fields)!=5:
            return False
        return True
    except:
        return False

trans = sc.textFile("/data/ethereum/transactions")
trans_f = trans.filter(clean_trans)
address=trans_f.map(lambda l: (l.split(',')[2], int(l.split(',')[3]))).persist()
partbjob1output = address.reduceByKey(lambda a,b:(a+b))
partbjob1output_join=partbjob1output.map(lambda f:(f[0], f[1]))

contracts = sc.textFile("/data/ethereum/contracts")
contracts_f = contracts.filter(clean_contracts)
contracts_join = contracts_f.map(lambda f: (f.split(',')[0],f.split(',')[3]))

partbjob2output = partbjob1output_join.join(contracts_join)

top10=partbjob2output.takeOrdered(10, key = lambda x:-x[1][0])
for record in top10:
    print("{}: {}".format(record[0],record[1][0]))
```

Explanation:

To implement Part B with a single spark Job, the operations of all the 3 jobs will be performed by a single spark job. First step is to calculate the aggregate counts for all transactions in the TRANSACTIONS dataset. Second step is to filter out the smart contracts address from user address. Third step is to sort the address and filter out top 10 services.

• First line reads the transaction dataset using pyspark's sparkcontext function. Second line filters any bad lines from transactions dataset using the user defined function clean_transactions.

• In third line using spark's lambda function we map the key as address and value as values from transaction dataset. In fourth line we use spark's reduceByKey function to calculate the aggregate of transaction values. This output is kept in memory .

• In fifth line we map the key as address and value as aggregate values from the output of previous operation. Spark's in-memory processing is made use here. These values are stored in variable job1output_join.

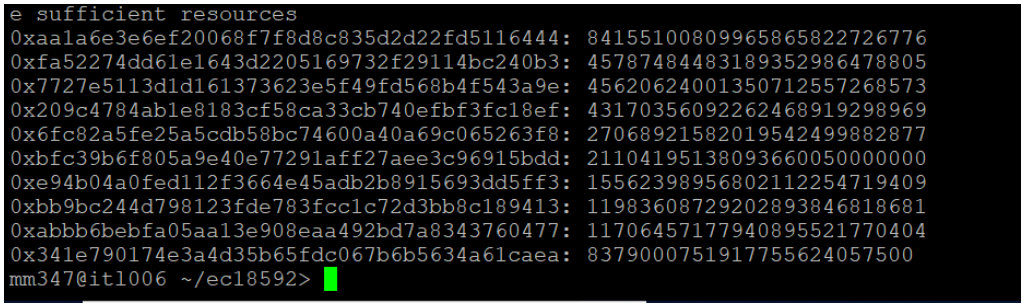
• Sixth line reads the contract dataset. Seventh line filters out the bad lines from contracts dataset.

• Eighth line maps the key as address and value as block_number from contracts dataset using spark's lambda function.

• Ninth line performs the join operation using spark's join operation. Variable joined_data is the result of the joined dataset.

• Tenth line performs the sorting operation and filtering only 10 values using spark's takeOrdered function. The symbol '-' in lambda function x:-x[1][0] sorts the values in descending order leaving top value at first.

Output:



Comparison between Spark and Map/Reduce:

• Hadoop MapReduce has to read from and write to a disk. Whereas spark can perform in-memory. As a result, speed differs.

• Spark performs fast data processing as this program's data should process again and again. Spark RDD's enables multiple map in memory. While Hadoop has to write interim results to disk and process the data again and again.

• In joining dataset, due to its speed, Spark can create all combinations faster. Though Hadoop maybe better if joining of very large sets that requires lots of shuffle and sort.

• As Hadoop MapReduce program should run in HDFS and merged back to disk, this process has done for multiple times. Hence the average speed I observed to finish one whole task is apminutes. While Spark has finished executing around 6 minutes every time as shown below.

Spark JOBID(for which I run multiple times):

Part B spark first round
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_1702
Duration 6 min 14sec

part b spark second round

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_1723
duration 3mins, 8sec

par B spark third round
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_1744
duration 2mins, 47sec

Hadoop JOB ID (for which I run the multiple times:)

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_1507
Duration: 13 min,43 sec
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_1702
Duration:6 min 14 sec
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_1723
Duration 3 min 8 sec
• Number of tasks required for Hadoop to analyse is minimum 4, while spark can perform in 1 task.

Job Id s when the program ran for multiple times:

First time:

Part B job 1
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_1793
Duration: 34mins, 37sec

Part B job 2
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_1944

DURATION 9mins, 5sec

Part B job 3
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2018

DURATION 34sec

Second time:

part job 1

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2041
duration 37mins, 10sec
part job 2
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2228
duration 9mins, 8sec
part B job3

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2288
duration 30sec

Third time:

PART JOB 1
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2324
DURATION 39mins, 52sec
PART JOB 2
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2555
DURATION 9mins, 4sec
PART JOB3
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2605
DURATION 29 sec

PART C SCAM ANALYSIS:

Hadoop JobID

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_3194

- I tried to download the scam.json file ,converted to .csv in python and upload back to the HDFS.

```
import pyspark
import time
sc=pyspark.SparkContext()
#clean_transactions function to filter out bad lines from transactions dataset
def clean_transactions(line):
   try:
      fields = line.split(',')
      if len(fields)!=7:
         return False
      int(fields[6])
      int(fields[3])
      return True

   except:
      return False
#read transactions dataset
transactions = sc.textFile('/data/ethereum/transactions')
#remove bad lines from transactions dataset using clean_transactions function
transactions_f = transactions.filter(clean_transactions)
#map the to_address and value field from transactions dataset
transactions_join = transactions_f.map(lambda l: (l.split(',')[2] , (int(l.split(',')[6]),
int(l.split(',')[3])))).persist()
#read scams dataset
scams = sc.textFile('/user/mm347/scams.csv')

scams_join = scams.map(lambda f: (f.split(',')[0],f.split(',')[6]))

joined_data = transactions_join.join(scams_join)
# joined_data.saveAsTextFile('scamsjoinnew')

category = joined_data.map(lambda a: (a[1][1], a[1][0][1]))
category_sum = category.reduceByKey(lambda a,b: (a+b)).sortByKey()
category_sum.saveAsTextFile('lucrative_scam'
time_series = joined_data.map(lambda b: ((b[1][1], time.strftime("%m-
%Y",time.gmtime(b[1][0][0]))), b[1][0][1]))
time_series_sum = time_series.reduceByKey(lambda a,b: (a+b)).sortByKey()
time_series_sum.saveAsTextFile('timeseries')
```

Explanation:

- In JSON, we need to know the most lucrative form of scam, file include category(scam type) and address to which it belongs.
- Once JSON is converted to csv file and uploaded back to the HDFS ,above program can be used .
- Then ,take the count of scam category over wei value which shows that scamming is the highest in count . Once the category is observed we can join it to the transactions to get the time stamp which is then used to plot the time analysis. (Tried to implement is similar to the part A)

I tried to execute the problem in similar to part A ,but unfortunately I couldn't get the time series analysis.