Big Data Processing

Ethereum trasaction analysis with MapReduce on Hadoop

By
Areeba Kamil

PART A. TIME ANALYSIS

For time analysis, the Blocks table was used as the input for the job. The blocks table consists of the fields transaction_count and timestamp. This job consists of a mapper, a combiner and a reducer. The output of this job is the sum of transacton_count for each month-year.

Mapper:

```
def mapper(self, _, line):
        fields = line.split(",")
        try:
            if(len(fields) == 9):
                time_epoch = int(   fields[7])
                month_year = time.strftime("%B-%Y",time.gmtime(time_epoch)) #returns month and
year
                transaction_count = int(fields[8])
                yield (month_year, transaction_count)
        except:
            Pass
```

In the mapper, first, the line is split by commas and added to an array called fields. The remaining code is surrounded by a try and except block. In order to discard malformed lines, there is a check inside the try block to ensure that the length of fields is exactly 9. If the length is 9, then it will proceed to the next line of code, otherwise, it will move into the except block where this line is ignored.
If the length of the fields is exactly 9, then we extract the month-year and transaction count. The month-year is extracted from the time stamp using the functions from the time library for python. The timestamp is the 7th field of the array named fields. The transaction count is the 8th field of the array named fields. This transaction count is then converted into an integer. Finally, the month year and transaction count is then yielded as a key-value pair. These key-value pairs will be the input for the combiner

Combiner:

```
 def combiner(self, month_y, transaction_no):
        yield(month_y, sum(transaction_no))
```

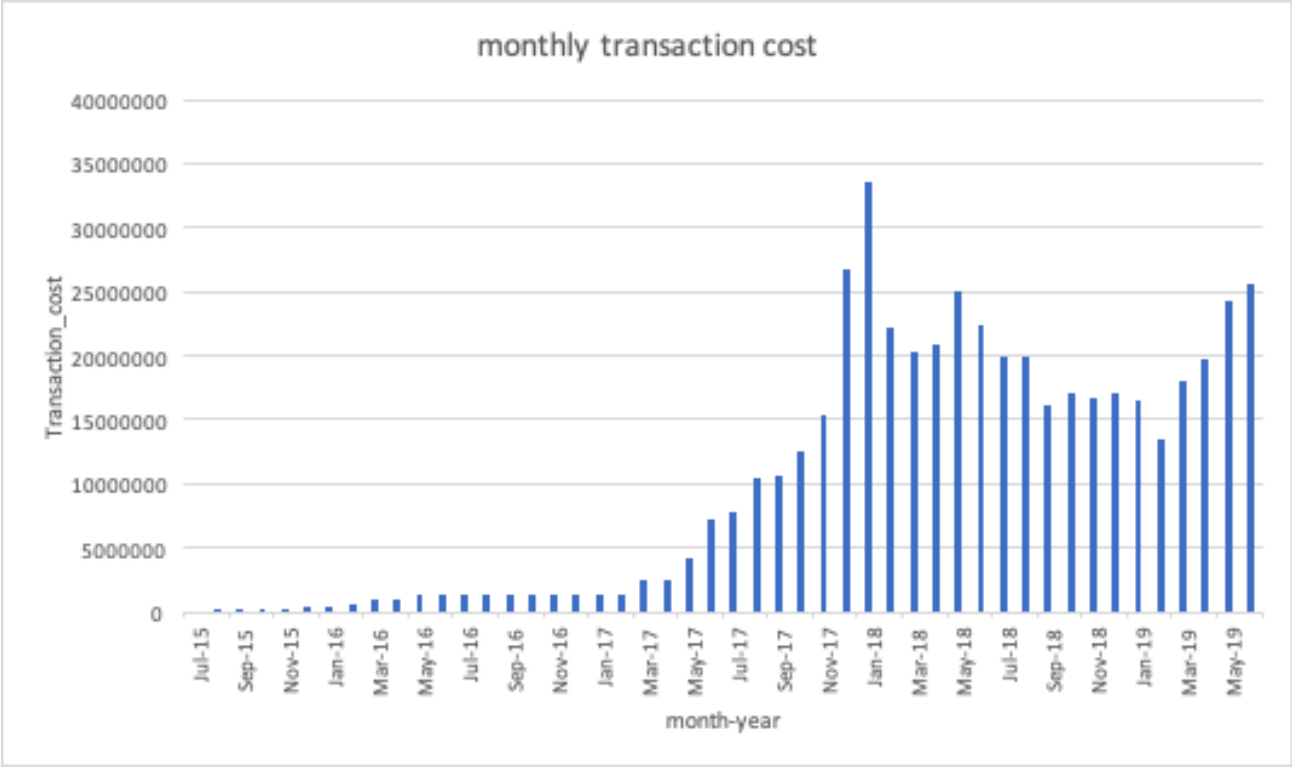
The combiner aggregates transactions_counts for each month_year yielded by each mapper. This is then passed as the input to the reducer

Reducer:

```
def reducer(self, month_y, transaction_no):
    yield(month_y, sum(transaction_no))
```

The reducer aggregated the transaction_count for each month-year and outputs it.

Graph of the output:



PART B. TOP TEN MOST POPULAR SERVICES

For this part, the top 10 addresses containing the highest number of Wei received is obtained. The contracts and transactions table was used as the input. The transactions table consists of the to_address and value transferred in Wei. The contracts table consists of addresses. In this job, the value transferred in Wei is aggregated for each to_address from the transactions table and then the addresses in the contracts table are joined with the to_addresses from the transactions table. This job consists of a mapper and 2 reducers. Since more than one reducer is being used, the steps for MrJob were defined as follows:

```
def steps(self):
        return [MRStep(mapper=self.mapper_1,
                       reducer=self.reducer_1),
                MRStep(reducer=self.reducer_2)]
```

Mapper:

```
    def mapper_1(self, _, line):
        fields = line.split(",")
        try:
            if len(fields) == 7:
                to_address = fields[2]
                values = int(fields[3])
                yield (to_address, [1, values])
            elif len(fields) == 5:
                address = fields[0]
                yield (address, [2])
```

```
        except:
            pass
```

Since both the data sets - contracts and transaction tables - are very large, we are using repartition join in which both the data sets are processed in the mapper. In the mapper, first, the lines are split by commas and added to an array called fields. The remaining code is surrounded by a try and catches block.

Then the length of the fields is checked to differentiate between which table the line is coming from. If the length of the array is exactly 7 then it corresponds to the transactions table else if the length is exactly 5, then it corresponds to the contracts table otherwise, the line is ignored by the except block. From the transactions table, the to_address and values are extracted. Then the key-value pairs are yielded where the key is the to_address and the value is an array with two elements: [1, value]. Here the first index contains digit 1 which acts as an identifier to help the reducer know that this data is coming from the transactions table. The second index contains the value transferred in Wei. From the contracts table, the address is extracted and the key-value pairs are yielded. Here the key is the address and value is an array that consists of a single element : [2]. This is used as an identifier for the reducer to know that this data came from the contracts table

Reducer 1

```
    def reducer_1(self, address, values):
        exists = False
        summation = 0
        for value in values:
            if value[0] == 1:
                summation += value[1]
            elif value[0] == 2:
                exists = True
        if exists and summation > 0:
            yield (None, [address, summation])
```

The first reducer checks whether the values are coming from the contracts table or the transactions table. Since values is an array of arrays, a for loop runs over values. Inside the for loop, the first index of each array inside values is checked. If the first index is 1, then the array is

coming from the transactions table. If the first index is equal to 2, then the array is coming from the contracts table. If the data is coming from the transactions table then the array contains a second element which is the wei transferred. The wei transferred is then aggregated. If an array is found in which the first index is 2, then it means there is a match of address between the transactions table and the contracts table.

If a match exists and the total aggregate is greater than 0, then it is yielded as follows:

```
yield (None, [address, summation])
```

Here the key is None and the value is an array [address, the sum of Wei transferred]. This is fed into the second reducer. Since we have to perform sort on the summation of Wei transferred and get the addresses corresponding to the sorted summations in the second reducer, the key is set to None.

Reducer 2

```
  def reducer_2(self, key, value):
      sorted_vals = sorted(value, reverse=True, key=lambda t: t[1])
      i = 0
      for val in sorted_vals:
          address = val[0]
          total = val[1]
          yield ("{}: {}".format(address, total), None)
          i = i + 1
          if i >= 10:
              Break
```

The second reducer recieved the arrays for [address, summation of weis] inside the variable values. Values is sorted and then the top 10 among sort values is reterieved and yeilded

PART C. DATA EXPLORATION

Gas guzzlers

To check how the gas price has changed over time, the maximum gas prices for each month-year is calculated and sorted. Then the top ten are picked out. For this, the transactions table is used as the input. For this job, one mapper and 2 reducers are used. Since more than one reducer is being used, the steps are defined as follows:

```
def steps(self):
    return [MRStep(mapper=self.mapper_1,
                   reducer=self.reducer_1),
            MRStep(reducer=self.reducer_2)]
```

Mapper

```
def mapper_1(self, _, line):
    fields = line.split(",")
    try:
        if(len(fields) == 7):
            time_epoch = int(fields[6])
            month_year = time.strftime("%B-%Y",time.gmtime(time_epoch)) #returns month and year
            gas_price = int(fields[5])
            yield (month_year, gas_price)
    except:
        pass
```

The lines are split by commas and the array obtained is saved in the variable called fields. The remaining code is surrounded by a try and except block Since the transactions table consists of 7 fields, the if statement checks to see if the length of the fields is exactly 7 otherwise the line is ignored in the except block. The month-year is extracted from the time stamp using python's time library. The gas price is also extracted. The month-year and gas price are then yielded as a key-value pair

Reducer 1

```
def reducer_1(self, month_y, gas_p):
    yield (None, [month_y, max(gas_p)])
```

In the reducer the maximum has prices is obtained for each month-year and yielded as follows

yield(None, [month-year, maximum gas price])

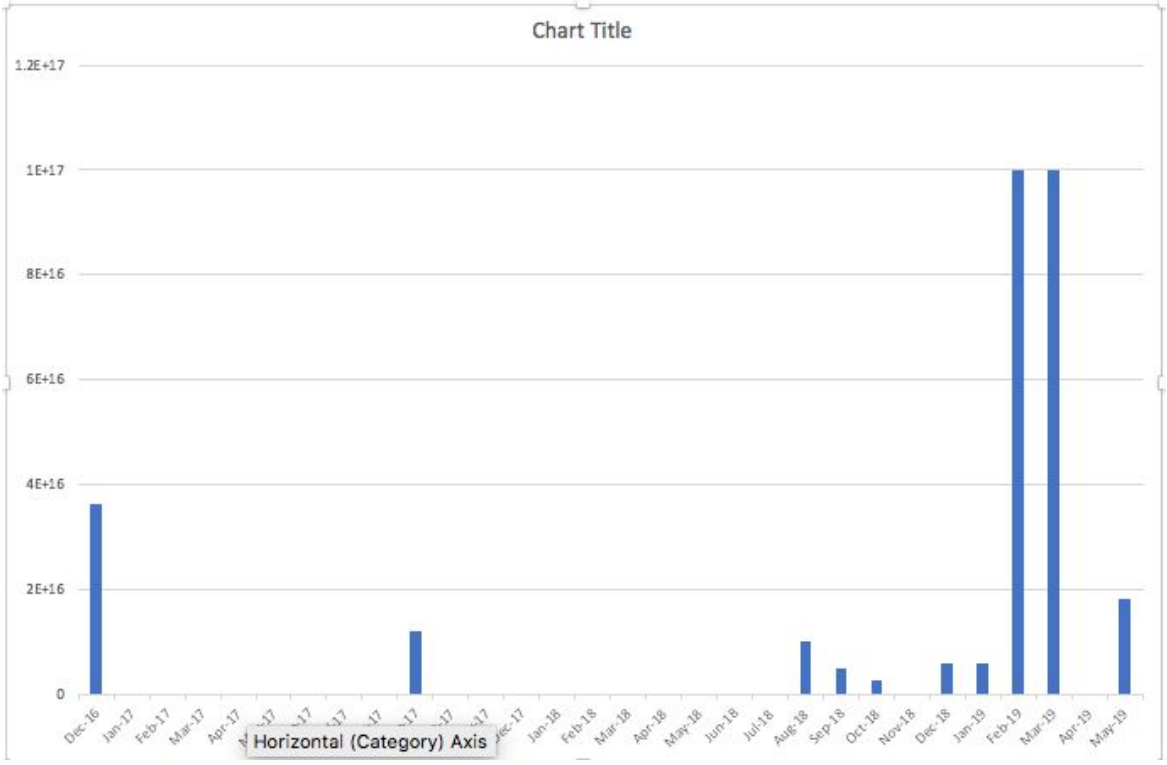
Here the key is set to none so that the sort can be performed in the second reducer on month-year according to gas price

Reducer 2

```
def reducer_2(self, key, values):
    #values = [month-year : max(gas_price)]
    sorted_values = sorted(values, reverse=True, key=lambda t: t[1])
    i = 0
    for val in sorted_values:
        month_year = val[0]
        gas_price = val[1]
        yield ("{}: {}".format(month_year, gas_price), None)
        i = i + 1
        if i >= 10:
            break
```

The second reducer performs sort on the array month-year and the gas price. Then the top 10 are picked out from the sorted values.

According to the data, the data below, the gas price was the highest in February and March 2019 and then it decreased again by May 2019



Comparative Evaluation

Job ID
Application_1575381276332_3088

Part B was implemented in spark. The job ran much faster in spark than in MrJob. The results obtained with spark and MrJob are exactly the same. For this job, spark seems appropriate since it took much less amount of time.

mrJob took around 25 minutes whereas Spark took around 4 minutes to run.

Spark saves time by reducing the number of read and write cycles to disk and storing intermediate data in-memory.