ENSF 612 – Fall 2021

Midterm – Wednesday, November 25

Student Name: Bhavyai Gupta

Submission date: November 25, 2021

Question 1

a. True.

b. False.

c. During the reduce task, the workers might fail. The master node keeping the track would know which worker node has failed. Then, the master node tries to find another standby worker node that is supposed to do the same reduce task. If there is no standby worker that is available to that same reduce task, then the master node keeps that failed task waiting. Once the master node finds some available worker, it assigns that failed task to completed to that available worker.

d. Reducer 1 will get that key, as `31 % 5 = 1`.

e. **Implementation**

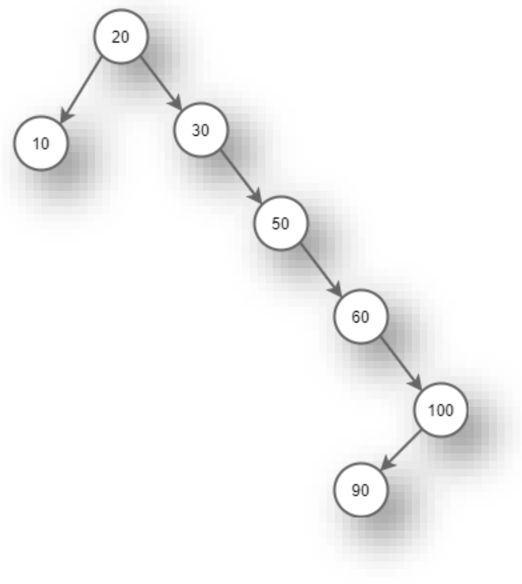
In the map task, we supply the root node as the key, and (distance from root, [list of children directly pointed by root]) as value.

Then for all nodes directly pointed by their root (represented by key), the map task outputs that root (represented by key) as key and the distance+1 as value. This value is fed to reduce task.

The reduce task gathers all possible distances of the node to be searched and selects where the distance has minimum value.

Explanation for given data, to search for value 60

First, we create a BST from the given data nodes - [20, 30, 10, 50, 60, 100, 90].



To implement the BFS algorithm –

We supply 20 as key and (0, [10, 30]) as value to the map task. The map emits (20,1) for both the nodes 10 and 30. Here in the output 20 is the root node of 10 and 30, and 1 is the distance of 10 and 30 from 20.

Then for the new root 30, we supply 30 as key and (1,[50]) as value to the map task. The map emits (30,2).

We repeat this process until all nodes are processed, so that we can find all possible ways to reach the node 60. The below table summarizes the input and outputs to the map task for the given nodes.

Root Node	Map Input	Target Nodes	Map Output
20	K=20, V=(0, [10, 30])	10, 30	K=20, V=1
10	K=10, V=(1, [])	-	
30	K=30, V=(1, [50])	50	K=30, V=2
50	K=50, V=(2, [60])	60	K=50, V=3
60	K=60, V=(3, [100])	100	K=60, V=4
100	K=100, V=(4, [90])	90	K=100, V=5
90	K=90, V=(5, [])	-	

The reduce task will look at the map outputs for the node to be searched, i.e., 60, and selects the minimum distance. Since we have only one possible value of (50, 3) for the target node 60, the reducer outputs **3** as BFS answer.

Question 2

Assume that records data is stored at `dbfs:/FileStore/midterm/q2_data.csv`. We first load the data from the file into a pyspark dataframe.

```
def read_CSV_to_DF(filepath, isHeader):
  """
  Read a csv file into a spark dataframe
  """
  df = (spark.read
        .option("multiline", "true")
        .option("quote", '"')
        .option("header", isHeader)
        .option("escape", "\\")
        .option("escape", '"')
        .csv(filepath)
        )

  return df

# creating the dataframe
df = read_CSV_to_DF('/FileStore/midterm/q2_data.csv', True)

# updating the datatype of columns of dataframe
df = df.withColumn('IssuePriority', df['IssuePriority'].cast('int'))
df = df.withColumn('NumberOfComponentsAffected',
df['NumberOfComponentsAffected'].cast('int'))
```

Also below are the functions that emulate the functionality we must assume.

```
import random
from datetime import datetime

@udf
def getIssueType(IssueDescription):
  """
  returns 'b' for bug, 'f' for new feature, and
  'e' for feature enhancement
  """

  issue_type = ['b', 'f', 'e']
  random.seed(len(IssueDescription))
  return random.choice(issue_type)
```

```
@udf
def getYear(CreationTime):
  """
  returns year of the CreationTime
  """

  CreationTime = int(CreationTime)
  creationYear = datetime.fromtimestamp(CreationTime).strftime('%Y')
  return creationYear


@udf
def getMonth(CreationTime):
  """
  returns month of the CreationTime
  """

  CreationTime = int(CreationTime)
  creationMonth = datetime.fromtimestamp(CreationTime).strftime('%m')
  return creationMonth


@udf
def getDay(CreationTime):
  """
  returns day of a week like Monday, Tuesday, Sunday
  """

  CreationTime = int(CreationTime)
  creationDay = datetime.fromtimestamp(CreationTime).strftime('%A')
  return creationDay
```

Task 2.1

```
# adding additional columns
df = df.select("*", getIssueType("IssueDescription").alias("IssueType"))
df = df.select("*", getYear("CreationTime").alias("IssueYear"))
df = df.select("*", getMonth("CreationTime").alias("IssueMonth"))
df = df.select("*", getDay("CreationTime").alias("IssueDay"))

# showing the results
df.toPandas().head()
```

Task 2.2

```
# total number of components
numberOfComponents = df.select('NumberOfComponentsAffected').rdd.flatMap(lambda
x: x).reduce(lambda x, y: x + y)

# printing the results
print("Total number of components affected by all the issues =
{}".format(numberOfComponents))
```

Task 2.3

Subtask 1

```
# sum all the priorities
rdd_sumPriority = df.select(['IssueType', 'IssuePriority']).rdd.map(lambda x:
(x['IssueType'], (x['IssuePriority'], 1))).reduceByKey(lambda a, b: (a[0]+b[0],
a[1]+b[1]))

# divide by total to get average
rdd_avgPriority = rdd_sumPriority.map(lambda x: (x[0], x[1][0]/x[1][1]))

# printing the average IssuePriority per IssueType
spark.createDataFrame(rdd_avgPriority, ['IssueType', 'Average
Priority']).show(n=100)
```

Subtask 2

```
# total number of reviews by IssueType
rdd_issuesByType = df.select(['IssueType']).rdd.map(lambda x: (x['IssueType'],
1)).reduceByKey(lambda a, b: a+b).sortByKey()

# pretty print into table using df.show()
spark.createDataFrame(rdd_issuesByType, ['IssueType', 'Total
Issues']).show(n=100)
```

Subtask 3.a

```
# total number of issues by year
rdd_issuesByYear = df.select(['IssueYear']).rdd.map(lambda x: (x['IssueYear'],
1)).reduceByKey(lambda a, b: a+b).sortByKey()

# pretty print into table using df.show()
spark.createDataFrame(rdd_issuesByYear, ['IssueYear', 'Reported
Issues']).show(n=100)
```

Subtask 3.b

```
# total number of issues by month
rdd_issuesByMonth = df.select(['IssueMonth']).rdd.map(lambda x: (x['IssueMonth'],
1)).reduceByKey(lambda a, b: a+b).sortByKey()

# pretty print into table using df.show()
spark.createDataFrame(rdd_issuesByMonth, ['IssueMonth', 'Reported
Issues']).show(n=100)
```

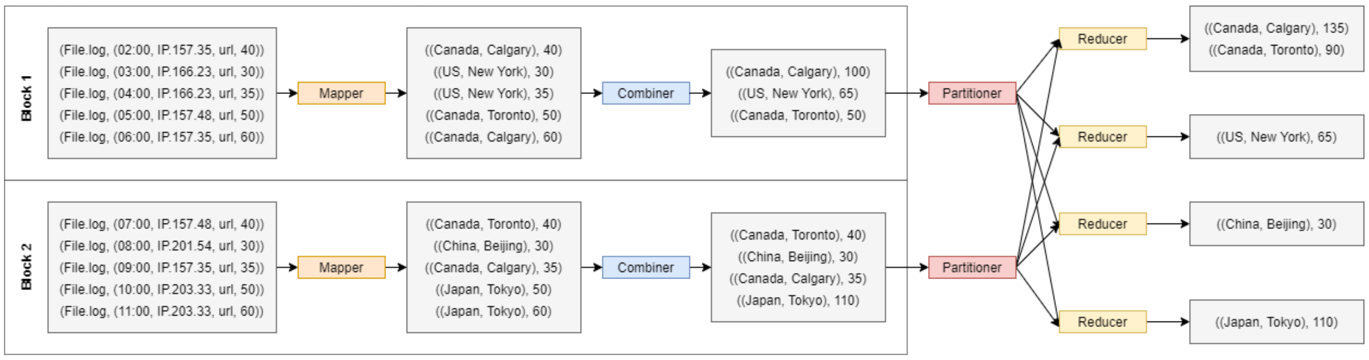
Subtask 3.c

```
# total number of issues by day
rdd_issuesByDay = df.select(['IssueDay']).rdd.map(lambda x: (x['IssueDay'],
1)).reduceByKey(lambda a, b: a+b).sortByKey()

# pretty print into table using df.show()
spark.createDataFrame(rdd_issuesByDay, ['IssueDay', 'Reported
Issues']).show(n=100)
```

Question 3

MapReduce analytic example diagram –



MapReduce analytic explanation –

Mapper
Map function takes key=log file and value=text of the log file as inputs, then uses given functions to extract country and city from the Client_IP. Finally, the map function emits a key-value pair where key is a tuple (Client_country, Client_city) and values is the Size_of_data_transferred.

Input = (web server log file name, text of log file)
Output = ((Country, City), Size_of_data_transferred)

pseudo code for map -

```
def map(key, value):
    """
    key = web server log file
    value = text of document
    """

    for each line l in value:
        '''
        l = 'Access_time, Client_IP, URL_requested, Size_of_data_transferred'
        '''

        # get the Client_IP and Size_of_data_transferred by splitting each line
        client_ip_add = l.split(',')[1]
        size_of_data = l.split(',')[3]


        # create a key, value like below where key is a tuple of Country and
City, and value is size of data
```

```
        # ((Country, City), Size_of_data)
        emit((getCountry(client_ip_add), getCity(client_ip_add)), size_of_data)
```

Combiner
The combiner placed in the mapper nodes aggregates the output of each mapper and combines all keys in each single mapper node by aggregating their values. It just behaves like a mini reducer for each mapper.

pseudo code for combiner is going to be like that of a reducer -

```
def combiner(key, values):
    """
    key = (Country, City)
    values = an iterator over size_of_data
    """

    result = 0

    for each size_of_data s in value:
        result += s

    emit(key, result)
```

Partitioner
Partitioner shuffles the output of combiner in such a way that all keys with same country goes to the same reducer as input. For this, first we get the number of distinct countries. Let's say that we store the count of countries in R. Then we override the hash function of the partitioner which ensures a particular Client_country ends up in the same reducer.

pseudo code for overriding hash function -

```
R = getCountryCount()
hash(Client_country) mod R
```

Reducer
Reduce function takes key=(Country,City) and value=Sum of Size_of_data_transferred from each combiner. Here every reducer gets a specific Country due to the Partitioner used. The reducer then further aggregates the Sum of Size_of_data_transferred according to the City. The final output of the every reducer is the Total_size_of_data_transferred for a specific country but grouped by cities.

Input = ((Country, City), Size_of_data_transferred)
Output = ((Country, City), Total_size_of_data_transferred)

pseudo code for reducer -

```
def reduce(key, values):
```

```
"""
key = (Country, City)
values = an iterator over size_of_data
"""

result = 0

for each size_of_data s in value:
    result += s

emit(key, result)
```

Question 4

Assume that records data is stored at `dbfs:/FileStore/midterm/q4_data.csv`. We first load the data from the file into a pyspark dataframe.

```
# creating the dataframe
df = read_CSV_to_DF('/FileStore/midterm/q4_data.csv', True)
```

Task 1

Subtask a

Creating a helper function that splits the list of friends and explode them into rows of the dataframe

```
from pyspark.sql.functions import col, explode, regexp_replace, split

def list_splitter(pyspark_df):
  """
  Splits the list of friends and explode it into rows
  """

  return pyspark_df.withColumn("Friend_profile_id_list",
explode(split(regexp_replace(col("Friend_profile_id_list"), "(^\[)|(\]$)", ""),
",")))
```

Function that counts the friends of a given profile "profile_id".

```
def friend_count(pyspark_df, profile_id):
  """
  Returns the count of friends of the given
  profile_id in the dataframe pyspark_df
  """

  # filter the profile_id from the dataframe
  pyspark_df = pyspark_df.filter(pyspark_df['Profile_id'] == profile_id)

  # explode the friend list
  pyspark_df = list_splitter(pyspark_df)

  # compute the friend count and return it
  return pyspark_df.rdd.map(lambda x: (x['Profile_id'], 1)).reduceByKey(lambda a,
b: a+b).collect()[0][1]
```

Call to the function for profile_id '416'.

```
friend_count(df, '416')
```

Subtask b

Function that returns the dataframe of common friends for the given profiles "profile_1" and "profile_2".

```
def common_friend_list(pyspark_df, profile_1, profile_2):
  """
  Returns the dataframe of common friends of given profiles
  profile_1 and profile_2
  """

  # filter profiles from the dataframe
  pyspark_df = pyspark_df.filter((df['Profile_id'] == profile_1) |
(df['Profile_id'] == profile_2))

  # explode the friend list
  pyspark_df = list_splitter(pyspark_df)

  # count the key=common friends, so that the corresponding value of common
friends is 2
  df_counts = pyspark_df.rdd.map(lambda x: (x['Friend_profile_id_list'],
1)).reduceByKey(lambda a, b: a+b)

  # reverse the key and value
  df_counts = df_counts.map(lambda x: (x[1], x[0]))
  df_list = spark.createDataFrame(df_counts, ['Count', 'Common Friend'])

  # print all values where key=2
  return df_list.filter(df_list['Count'] == 2).select('Common Friend')
```

Call to the function for profiles '416' and '501'.

```
common_friend_list(df, '501', '416').show(n=100)
```

Task 2

MapReduce analytic explanation –

Mapper
Map function takes key=profile_id and value=comma separated friend list. It is analogous to BFS using mapreduce, where the key is the root node and the friend list is the adjacency_list.

In this map step, we iteratively visit each friend for every profile_id and emit records corresponding to each friend node, basically emitting root profile_id as key and 1 as value.

Input = (profile_id, comma separated friend list)
Output = (profile_id, 1), with one record of every friend for a given profile_id

Reducer
Reduce function takes key=Profile_id and value=1 from the mapper. It then aggregates the value based on the key, so that output of the reducer produces profile_id as key and the sum of values as value. This sum of value is basically the count of friends for every profile_id.

Input = (Profile_id, 1), with one record of every friend for a given profile_id
Output = (Profile_id, count of friends)