Working with MapReduce

DS730

In this project, you will be working with input, output, Python and Hadoop framework. You will be writing multiple mappers and reducers to solve a few different problems.

If you recall, the map and reduce functions are **stateless** and this is especially important when dealing with Hadoop and distributed work. We can’t guarantee that any 1 mapper will read in all of the data. Nor can we guarantee that certain inputs will end up on the same machine for mapping. Rather, 1 mapper will likely read in a small portion of the data. The output that your mapper produces must only depend on the current input value. For the reducer, you can only guarantee that (key,value) pairs with the same key will end up on the same reducer.

Your mapper and reducer cannot be trivial. For example, do not have all of your mappers map use the same key and then solve everything in the reducer. Such a solution defeats the purpose of MapReduce because all (key,value) pairs will end up on the same reducer. If you are unsure if your keys are trivial, post a private message to the message board for the instructors and we will let you know if your keys are trivial. A couple of very important things:

1. **Make sure your key is separated by your value using a tab. Hadoop will only work if this is the case. Otherwise, Hadoop has no idea what your “key” is nor will it know what your “value” is.**
2. **Make sure that this is the first line of your mapper and reducer:  
   #!/usr/bin/env python**

You must write 1 mapper file and 1 reducer file to solve each of the following problems. Make sure you name your files **mapperX.py** and **reducerX.py** where X is the problem number. For each problem, Hadoop will define what your input files are so there is no need to read in from any file. Simply read in from the command line. You are encouraged to use the “starter” mapper and reducer as shown in the activity.

Discovering Contacts

On many social media websites, it is common for the company to provide a list of suggested contacts for you to connect with. Many of these suggestions come from your own list of current contacts. The basic idea behind this concept being: I am connected with person A and person B but not person C. Person A and person B are both connected to person C. None of my contacts are connected to person D. It is more likely that I know person C than some other random person D who is connected to no one I know. For this problem, all connections are mutual (i.e. if A is connected to B, then B is connected to A). In this problem, you will read in an input file that is delimited in the following manner:

PersonA : PersonX PersonY PersonZ PersonQ

PersonB : PersonF PersonY PersonX PersonG PersonM

…

For example, the person to the left of the colon will be the current person. All people to the right of the colon are the people that the current person is connected to. All people will be separated by a single space. In the example above, PersonA is connected to PersonX, Y, Z and Q. In all inputs, all people will be replaced with positive integer ids to keep things simple. The following is a sample input file:

6 : 2 9 8 10

1 : 3 5 8 9 10 12

4 : 2 5 7 8 9

2 : 3 4 7 6 13

12 : 1 7 5 9

3 : 9 11 10 1 2 13

10 : 1 3 6 11

5 : 4 1 7 11 12

13 : 2 3

8 : 1 6 4 11

7 : 5 2 4 9 12

11 : 3 5 10 8

9 : 12 1 3 6 4 7

The ordering of people on the right hand side of the input can be in any order. Your goal is this: you must output *potential contacts* based on the following 2 criteria:

1. Someone who **might** be someone you know. For someone to be suggested here, the person must not currently be a connection of yours and that person must be a connection of exactly 2 or 3 of your current connections. For example, consider person 2 in the above example. Person 2 is connected with 3, 4, 6, 7 and 13. Person 4 is connected to 8, person 6 is connected to 8, person 3 is not connected to 8, person 7 is not connected to 8 and person 13 is not connected to 8. Therefore, person 2 has two connections (4 and 6) that are connected to 8 and person 2 is not currently connected to 8. Therefore, person 2 might know person 8.
2. Someone you **probably** know. For someone to be suggested here, the person must not currently be a connection of yours and that person must be a connection of 4 or more of your current connections. For example, consider person 2 in the above example. Person 2 is connected with 3, 4, 6, 7 and 13. Person 4 is connected to 9, person 6 is connected to 9, person 3 is connected to 9 and person 7 is connected to 9. Therefore, person 2 has at least four connections that are connected to 9 and person 2 is not currently connected to 9. Therefore, person 2 probably knows person 9.

Your output must be formatted in the following fashion:

personID:Might(personA,…,personX) Probably(personA,…personX)

For each line you have a personID following by a colon. The colon is followed by the list of Might’s separated by commas (but no space). If a person has no one they might be connected to, this list is not printed at all (see person 13 below for example). The Might list is followed by a single space and then followed by the Probably list separated by commas (but no space). If a person has no one they probably are connected to, this list is not printed at all (see person 3 for example). If a person has neither a might list nor a probably list, that person only has their id along with a colon (see person 13 for example). The *Might* list must appear before the *Probably* list. If there is no *Might* list but there is a *Probably* list, there is no space between the colon and the *Probably* list. The integers within each list must appear in increasing order. However, the order the rows appear in the output need not be in any specific order. For example, the row for 5 might appear before the row for 3. As a concrete example from the above sample input, this would be a potential sample output:

1:Might(4,6,7) Probably(11)

2:Might(5,8,10) Probably(9)

3:Might(4,5,6,7,8,12)

4:Might(1,3,6,11,12)

5:Might(2,3,8,10) Probably(9)

6:Might(1,3,4,7,11)

7:Might(1,3,6)

8:Might(2,3,5,9,10)

9:Might(8,10) Probably(2,5)

10:Might(2,5,8,9)

11:Might(4,6) Probably(1)

12:Might(3,4)

13:

For each question, the rows do not have to be in any specific order. The following is also a valid output for number 3:

9:Might(8,10) Probably(2,5)

1:Might(4,6,7) Probably(11)

3:Might(4,5,6,7,8,12)

4:Might(1,3,6,11,12)

5:Might(2,3,8,10) Probably(9)

11:Might(4,6) Probably(1)

6:Might(1,3,4,7,11)

7:Might(1,3,6)

2:Might(5,8,10) Probably(9)

8:Might(2,3,5,9,10)

10:Might(2,5,8,9)

12:Might(3,4)

13:

Other Important Information

1. You can use any Python libraries as long as they are installed by default on the Hortonworks machine and your code works on Hadoop. **All projects will be tested using Hadoop on Hortonworks. You should ensure that your code executes correctly on that platform before submitting.**
2. The order of the rows in the output does not matter. This is shown in the last part of problem 3. For problem 2, for example, the **:1** row could come after the **eo:2** row.
3. The format must be:

mapper.py

#!/usr/bin/python3.6

#Be sure the indentation is identical and also be sure the line above this is on #the first line

import sys

import re

line = sys.stdin.readline()

pattern = re.compile("[a-zA-Z0-9]+")

while line:

for word in pattern.findall(line):

print(word+"\t"+"1")

line = sys.stdin.readline()

reducer.py

#!/usr/bin/python3.6

#Be sure the indentation is correct and also be sure the line above this is on the first line

import sys

current\_word = None

current\_count = 0

word = None

for line in sys.stdin:

line = line.strip()

word, count = line.split('\t', 1)

count = int(count)

if current\_word == word:

current\_count += count

else:

if current\_word:

print('%s\t%s' % (current\_word, current\_count))

current\_count = count

current\_word = word

if current\_word == word:

print('%s\t%s' % (current\_word, current\_count))