Washington State University
CPT_S 415 – Big Data
Online

Srinivasulu Badri

Assignment 5

Name: Nam Jun Lee

Student Number: 11606459

1. **[MapReduce]** This set of questions test the understanding and application of MapReduce framework.

a. Facebook updates the "common friends" of you and response to hundreds of millions of requests every day. The friendship information is stored as a pair (Person, [List of Friends]) for every user in the social network. Write a MapReduce program to return a dictionary of common friends of the form ((User i, User j), [List of Common Friends of User i and User j]) for all pairs of i and j who are friends. The order if i and j you returned should be the same as the lexicographical order of their names. You need to give the pseudo-code of a main function, and both Map() and Reduce() function. Specify the key/value pair and their semantics (what are they referring to?).

Pseudo-code:

Main()

{

Input: (Person, [List of Friends]) for every user

Map: (Person,[List of Friends]) -> ((User i, User j), [List of Common friends of each other])

Reduce: ((User i, User j), [List of Common friends of each other]) for all -> ((User i, User j), [List of Common friends of each other]) for each pair

Sort: Combine all keys using sort to return the lexicographical order of their names.

Output: Common friends of User i and User j who are friends order by lexicographical.

}

Map()

{

Applied to each pair, computes key-values pairs of input data

Intermediate key-value pairs are hash-partitioned based on key

Each partition ((User i, User j), [List of Common friends of each other]) is sent to a reducer
}

Reduce()

{

Takes a partition as input and computes key-values pairs ((User i, User j), [List of Common friends of each other]) for each pair

}

The Main() function has friendship information as a pair (Person, [List of Friends]) for every user. And find the dictionary of common friends for all pairs of User i and User j who are friends (Map()). (Person, [List of Friends]) -> ((User i, User j), [List of Common friends of each other]) for all. Next, find the key-value pair for each pair of User i and User j who are friends (Reduce()). ((User i, User j), [List of Common friends of each other]) for all -> ((User i, User j), [List of Common friends of each other]) for each pair. Finally, combine all keys using sort to return the lexicographical order of their names.

b. Top-10 Keywords. Search engine companies like Google maintains hot webpages in a set R for keyword search. Each record $r \in R$ is an article, stored as a sequence of keywords. Write a MapReduce program to report the top 10 most frequent keywords appeared in the webpages in R. Give the pseudo-code of your MR program.

Pseudo-code:

Main():

{

Input: article R

Splitting set record r in article R

For all words in r, count the word in each word

Make a list key-value: key = word, value = number of word

Map(): List(word, num of word) -> (word, [num of word]) for word # shuffling

Reduce(): (word, [num of word]) for word -> (word, count of word)

Sort: count of each number order by descending

Output: return the most 10 frequent keywords

}

Map():

{

Applied to each pair, computes key-values pairs of input data

Intermediate key-value pairs are hash-partitioned based on key

Each partition ((Word, [number of word]) is sent to a reducer # ex: (Cat, (1,1,1)), (My, (1,1,1,1,1,1)), (River, (1,1))

}

Reduce():

{

Initialize sum = 0
Count of all number of word in [number of each word: w1,w2,…,wx] # sum = sum + wx

Takes a partition as input and computes key-values pairs (Word, Count of the word) for each pair # ex: (Cat, 3), (My, 6), (River, 2)

}

The Main() function has article R. First splitting record in article R. And make key-value list (word, number of word). And find the dictionary of all pairs of words (Map()). List(word, num of word)-> ((Word), [number of word]). Next, find the key-value pair for each pair of words and compute each word count (Reduce()). ((Word), [number of word]) -> (Word, Count of the word). Finally, sort the count of each number in order by descending and return the top 10 most frequent keywords.

2. **[Graph Parallel Models]** This sets of questions relate to MR for graph processing.

a. Consider the common friends problem in Problem 1.a. We study a "2-hop common contact problem", where a list should be returned for any pair of friends i and j, such that the list contains all the users that can reach both i and j within 2 hops. Write a MR algorithm to solve the problem and give the pseudo code.

Pseudo-code:

Main()

{

Input: (Person, [List of Friends]) for every user

Map: (Person,[List of Friends]) -> ((User i, User j), [List of Common friends of each other]) for all

Sort and shuffle to find groups distances by reachable

Reduce: ((User i, User j), [List of Common friends of each other]) for all -> ((User i, User j), [List of Common friends of each other]) for each pair within only 2 hops.

Output: Return common friends of User i and User j who are friends within only 2 hops.

}

Map()

{

Input: Person, List of friend

Applied to each pair, computes key-values pairs of input data

Intermediate key-value pairs are hash-partitioned based on key

Each partition ((User i, User j), [List of Common friends of each other]) is sent to a reducer
}

Reduce()

{

Input: ((User i, User j), [List of Common friends of each other])

Find 2 hops h for each reachable and track the actual path. If h is more than 2 then not included.

Takes a partition as input and computes key-values pairs ((User i, User j), [List of Common friends of each other]) for each pair within h.

hops <- 2

$\forall_h \in [h_1, h_2]$

If h >= hops then None

Else if h <= hops then hops <- h

H.hop <- hops

Emit(Users, H)

}

This problem can be solved by slightly modifying the shortest path MR algorithm.

The Main() function has friendship information as a pair (Person, [List of Friends]) for every user. First, And find the dictionary of common friends for all pairs of User i and User j who are friends (Map()). (Person, [List of Friends]) -> ((User i, User j), [List of Common friends of each other]) for all. And for each reachable and track of the actual path within 2 hops. Finally, returns the value within 2 hops (ex: (User i, User j), [List of Common friend of a friend (it means 2 hops)]).

b. We described how to compute distances with mapReduce. Consider a class of d-bounded reachability queries as follows. Given a graph G, two nodes u and v and an integer d, it returns a Boolean answer YES, if the two nodes can be connected by a path of length no greater than d. Otherwise, it returns NO. Write an MR program to compute the query $Q(G, u, v, d)$ and give the pseudo code. Provide necessary correctness and complexity analysis.

Pseudo-code:

Main():

{

Input: Graph G, represented by adjacency lists u and v, and length d

Key: node ID

Value of node N:

Find N.distance (from start node s to N) and N.adjList [(u, w(v,u))], node id and weight of edge (u, v)

Initialization for all n, N.distance = infinity

Map(): $\forall_u \in N.AdjList: Emit\ (u, d + w(v, u))$

Sort and shuffle to find groups distances by reachable nodes

Reduce(): selects d (user input) distance path for each reachable node and track of actual path. If a path of length no greater than d (user input): Yes, else: No. Then, it returns (value of node N, Boolean answer)

Output: return Boolean answer YES, if the two nodes can be connected by a path of length no greater than d. Otherwise, it returns NO.

}

Map():

{

Input: nid v, nvalue N

all nodes are processed in parallel

d <- N.distance

emit(v, N)

for each (u,w) in N.AdjList

emit(u, d+w(v,u)) # for each node u adjacent to v, emit a revised distance via v; Each partition ((nid u, [distance]) is sent to a reducer

}

Reduce():

{

Input nid u, list[distance]

d_userInput <- user_input

$\forall_{d} \in [d_1, d_2, \dots, d_{user_{input}}]$

If d is Node then Node U <- d

Else if d <= d_userInput then d_userInput <- “Yes”

Else if d > d_userInput then d_userInput <- “No”

U.Boolean <- d_userInput

Emit(u, U)

}

The Main() function has Graph G, represented by adjacency lists u and v, and length d. First, find the distance and adjacent list. Next, for each node u adjacent to v, emit a revised distance via v and sent it to a reducer (Each partition ((nid u, [distance])). Now, select a d (user-input) distance path for each reachable node and track of the actual path. If a path of length no greater than d (user-input): Yes, else: No. Returns value of node N, Boolean answer.

3. **[Hadoop]**

a. Hadoop Program:

The attached CSV file contains hourly normal recordings for temperature and dew point temperature at Asheville Regional Airport, NC, USA. The unit of measurement is tenth of a degree Fahrenheit. So, 344 is 34.4 F.

Write a program using Hadoop to compute and output daily average measurements for temperature and dew point temperature. The daily average measurements should include measurements for 24-hour period, for example from 20100101 00:00 (2010, January 1st, 00:00) to 20100101 23:00 (2010, January 1st, 23:00). Output the result in the format shown below - the columns are date and the combined result (separated by comma) of daily temperature and daily dew point temperature:

20100101	377.04, 285.58
20100102	378.67, 286.92
....	...., ...

You may write the application in Java, C/C++ or Python language. Provide both source code and compiled code, if applicable, for your program.

Write the application in Python:

mapper.py:

```
#!/usr/bin/env python3
"""mapper.py"""

import sys

# input from sys.stdin (normal_hly_sample_temperature.csv)
for line in sys.stdin:
    # remove whitespace
    line = line.strip()
    # split the line ','
    line = line.split(',')
    # extract only the information needed
    (date, temp, dewp) = (line[5][0:9], line[6], line[7])
    # ready to send to reducer
    print ('%s\t%s\t%s' % (date, temp, dewp))
```

reducer.py

```
#!/usr/bin/env python3
"""reducer.py"""

import sys
# day of hours
DAYHOURS = 24
(date, temp, dewp) = (None, 0, 0)
```

```
# Avg temperature
avg_temp = 0.0
# Avg dew point temperature
avg_dewp = 0.0

# input from sys.stdin (normal_hly_sample_temperature.csv)
for line in sys.stdin:
    # remove whitespace and paser the input from mapper
    (key, val, val1) = line.strip().split("\t")
    try:
        # convert temperature and dew point temperature (str to int)
        val, val1 = int(val), int(val1)
    except ValueError:
        # ignore error
        continue
    # hadoop sorts map output by key before passed to reducer
    if date == key:
        # sum daily temperature
        avg_temp = (avg_temp + val)
        # sum daily dew point temperature
        avg_dewp = (avg_dewp + val1)
    else:
        if date:
            # write the output
            # divide the average of temperature and dew point temperature
using daily hours
            # show only 2 decimal points
            print('%s\t%s, %s' % (date, round(avg_temp/DAYHOURS,2),
round(avg_dewp / DAYHOURS, 2)))
        avg_temp = val
        avg_dewp = val1
        date = key
# write the all date the output
if date == key:
    print('%s\t%s, %s' % (date, round(avg_temp/DAYHOURS,2), round(avg_dewp
/ DAYHOURS, 2)))
```

Run Hadoop to Terminal:

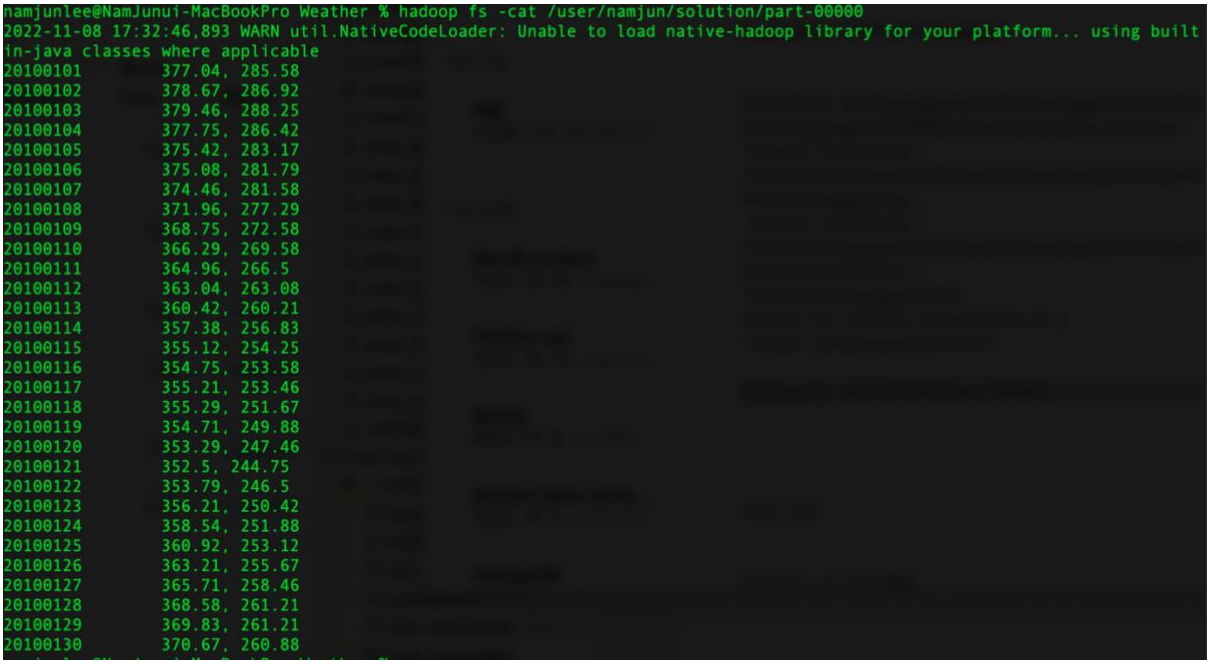
create the directory
Hadoop fs -mkdir /user
Hadoop fs -mkdir /user/namjun
Hadoop fs -mkdir /user/namjun/input

specificy execution permissions
chmod +x mapper.py
chmod +x reducer.py

run hadoop program using streaming
Hadoop jar /opt/homebrew/Cellar/hadoop/3.3.4/libexec/share/hadoop/tools/lib/hadoop-streaming-3.3.4.jar \
-mapper mapper.py \
-file /Users/namjunlee/PycharmProjects/pythonProject1/Weather/mapper.py \
-reducer reducer.py \
-file /Users/namjunlee/PycharmProjects/pythonProject1/Weather/reducer.py \
-input /user/namjun/input/normal_hly_sample_temperature.csv \
-output /user/namjun/solution

show result to terminal
Hadoop fs -cat solution/part-00000

Result to terminal:



```
namjunlee@NamJunui-MacBookPro Weather % hadoop fs -cat /user/namjun/solution/part-00000
2022-11-08 17:32:46,893 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using built
in-java classes where applicable
20100101        377.04, 285.58
20100102        378.67, 286.92
20100103        379.46, 288.25
20100104        377.75, 286.42
20100105        375.42, 283.17
20100106        375.08, 281.79
20100107        374.46, 281.58
20100108        371.96, 277.29
20100109        368.75, 272.58
20100110        366.29, 269.58
20100111        364.96, 266.5
20100112        363.04, 263.08
20100113        360.42, 260.21
20100114        357.38, 256.83
20100115        355.12, 254.25
20100116        354.75, 253.58
20100117        355.21, 253.46
20100118        355.29, 251.67
20100119        354.71, 249.88
20100120        353.29, 247.46
20100121        352.5, 244.75
20100122        353.79, 246.5
20100123        356.21, 250.42
20100124        358.54, 251.88
20100125        360.92, 253.12
20100126        363.21, 255.67
20100127        365.71, 258.46
20100128        368.58, 261.21
20100129        369.83, 261.21
20100130        370.67, 260.88
```

Result to Hadoop browse directory: