CSM 61C Fall 2019

MapReduce, Spark and IO

Mentoring 11: November 25, 2019

1 Speaking Spark

1.1 In this question let's write Spark to find the mode of a list of values and how often it occurs. As a refresher, the mode is the number that appears most often. If there is a tie, select any of the options. Fill in the blanks for the Python code below. Use the following Spark Python functions when necessary: map , flatmap , reduce , reduceByKey .

```
#Sample input:  [1 , 2, 1 , 2, 3, 4, 5, 6, 4, 2, 1, 3, 3, 1 , 1
, 2, 2, 1 ]
#Sample output: (1, 6) => 1 is the mode, and it occurs 6 times

def output_data (val):
    return __________

def compute_count (a, b):
    return __________

def find_max_occurrence (a, b):
    if a[1] > b[1]:
        return a
    return b

#values = list (numbers)
modeData = sc.parallelize (values)
 .__________(____________________)
            .__________(____________________)
            .__________(____________________)
```

```
def output_data (val):
    return (val, 1)

def compute_count (a, b):
    return a + b

modeData.map(output_data)
         .reduceByKey(compute_count)
         .reduce(find_max_occurrence)
```

2 More MapReduce

2.1 Imagine we're looking at Facebook's friendship graph, which we model as having a vertex for each user, and an undirected edge between friends. Facebook stores this graph as an adjacency list, with each vertex associated with the list of its neighbors, who are its friends. This representation can be viewed as a list of degree 1 friendships, since each user is associated with their direct friends. We're interested in finding the list of degree 2 friendships, that is, an association between each user and the friends of their direct friends.

You are given a list of associations of the form (user_id, list(friend_id)), where the user_id is 1st degree friends with all the users in the list.

Your output should be another list of associations of the same form, where the first item of the pair is a user_id, and the second item is a list of that user's 2nd degree friends. Note: a user is not their own 2nd degree friend, so the list of second degree friends must not include the user themselves.

Write pseudocode for the mapper and reducer to get the desired output from the input. Assume you have a set data structure. You can call set(list) to create a set, and use .remove(value), and .union(set) methods.

```
def flatMapFunc(person, friendIDs):
    return ____________________________________________

def reduceFunc(_________________, _________________):
    return ____________________________________________

def mapFunc(_________________, _________________):
    ________________________________________________
    return ____________________________________________

# persons = list((person, list(friendIDs))
secondDegree = sc.parallelize(persons)
secondDegree.flatMap(lambda (k, v): flatMapFunc(k, v))
        .reduceByKey(lambda (v, v): reduceFunc(v, v))
        .map(lambda (k, v): mapFunc(k, v))
return secondDegree
```

```
def flatMapFunc(person, friendIDs):
    return  [(friend, set(friendIDs)) for friend in friendIDs]

def reduceFunc(friendIDs_1, friendIDs_2):
    return  friendIDs_1.union(friendIDs_2)

def mapFunc(person, friendSet):
    friendSet = friendSet.remove(person)
    return  (person, friendSet)
```

Detailed solution: In the map phrase, we want to somehow associate a user with their second degree friends. The key here is to recognize that every friend in person A's friendIDs is a second degree friend to every other friend in that list (linked through person A). Thus, we want a function that takes in a person and his/her friendIDs and outputs a list of tuples containing each friend and a set of the second degree friends. We call flapMap on this function because each input could be mapped to more than one output.
In reduce, we simply combine (using union) all of the second degree friends lists which correspond to a specific user by calling reduceByKey.
The last step is to remove the user itself from the list of second degree friend. We call map in this case because each input is mapped to exactly one output.
Meta:
It might be useful to draw the graph for a toy example for this problem! Represent the relationships visually so you can work through what each "friend"/ID means.

3 Some Miscellaneous Quest(IO)ns

An important advantage of interrupts over polling is the ability of the processor to perform other tasks while waiting for communication from an I/O device. Suppose that a 1 GHz processor needs to read 1000 bytes of data from a particular I/O device. The I/O device supplies 1 byte of data every 0.02 ms. The time to process the data and store it in a buffer is negligible.

3.1 Assume a polling iteration takes 60 cycles. If the processor detects that a byte of data is ready through polling:

(a) How many cycles does it take for the I/O device to supply 1 byte of data?

20000

(b) How many polling iterations does it take to read 1 byte of data? (round up to an integer)

334

(c) How many cycles does it take to read the 1000 bytes of data?

20,040,000

3.2 If instead, the processor is interrupted when a byte is ready, and the processor spends the time between interrupts on another task, how many cycles of this other task can the processor complete while the I/O communication is taking place? The overhead for handling an interrupt is 2000 cycles.

18,000,000

3.3 The advantage of polling however arises when data rates become very large so that the interrupt overhead becomes substantial and at some point the system simply can't keep up. What is the data arrival time (in ms) at which point an interrupt-driven I/O scheme on this computer can't keep up with the data coming in? The overhead for handling an interrupt is 2000 cycles.

2000 / 1 GHz = 0.002ms

3.4 Solve for the maximum controller overhead to meet the following specifications: We need disk latency under 18 ms while reading 800 B of data. The hard drive spins at 6000 rev/min with a seek time of 2.5 ms and transfer rate of 80 KB/s (SI prefix). Don't forget units!

0.5 ms

controller overhead $\leq$ disk latency – seek time – rotation time – transfer time

rotation time = 0.5/rpm * (60 sec/1 min) = 0.005 s = 5 ms

transfer time = data size/transfer rate = 0.01 s = 10 ms

controller overhead $\leq$ 18 – 2.5 – 5 – 10 ms = 0.5 ms

3.5 To support interrupts, the CPU should be able to save and restore the current state. Which of the following should be saved before handling interrupts to ensure correct execution?

a. Program Counter b. User Registers c. TLB d. Caches

b

The PC gets saved to track where to return to after handling the interrupt. Similarly, the User Registers save information about the state before the interrupt. These are necessary to ensure correct execution. However, the TLB and Caches are implemented for efficiency, so execution would still be correct without them.

3.6 Consider the following three devices. For which device is Direct Memory Access (DMA) most beneficial?

Device	Data Rate	Transfer Block Size
A	80 B/s	4 B
B	400 MB/s	4 B
C	400 MB/s	2 KB

Device C.

DMA allows us to transfer data to/from memory independent of the CPU. This will be most useful when we have to write or read a lot of data from memory, as that will free up the most time for the processor to perform other tasks. Since C has the largest data block, it will receive the most benefit from DMA.