Tutoring Worksheet Week 6

MapReduce Overview:

* Simple data-parallel programming model, designed for scalability and fault-tolerance

MapReduce with Spark:

* map(*func*): return a new distributed dataset formed by passing each element of the source through a function func.
  + e.g: map(lambda (k, v): (k, v+1)) increments the value by 1 in each of the k, v pair.
* flapMap(*func*): similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).
  + e.g: flapMap(lambda (k, v): [(k, v + 1) for i in range(5)]) map each k, v pair to 5 pairs, specifically those that have the same key as the original pair, but have value + 0, value + 1, … , value + 4 as the new value for the generated pairs.
* reduce(*func*): aggregate the elements of the dataset regardless of keys using a function func.
  + e.g: reduce(lambda kv1, kv2: kv1[1] + kv2[1]) aggregate all the k, v pairs into one single value, which is the sum of all the values in all pairs.
* reduceByKey(*func*): when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V, V) => V.
  + e.g: reduceByKey(lambda x, y: x \* y) multiplies all the values that correspond to each specific key and uses this as the new value for that key.

**Question 1.** Speaking the Spark language.

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​.

Here is a sample input and output:

#Input: [1​, 2, 1​, 2, 3, 4, 5, 6, 4, 2, 1,​ 3, 3, 1​, 1​, 2, 2, 1​]

#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):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#values = list (numbers)

modeData = sc.parallelize (values)

modeData. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_( \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_( \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_( \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

**Question 2.** Map and Reduce are 2nd degree friends.

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

**Question 3.** Cache Revisited!

Assume we have a single L1 data cache having the following characteristics:

- 4 KiB cache size

- 16 byte blocks

- Direct Mapped

Assume the following piece of code is run in a 32-bit address space with sizeof(int) = 4 :

#define SIZE 8192 // 2^13

int ARRAY[SIZE];

int main() {

ARRAY[0] = ARRAY[4] + ARRAY[8]; // This happens before Loop 1

for (int i = 0; i < SIZE - 16; i += 4) { // Loop 1

ARRAY[i] += ARRAY[i + 4] + ARRAY[i + 8] + ARRAY[i + 12];

}

for (int i = SIZE - 1; i >= 0; i -= 32) { // Loop 2

ARRAY[i] += 10;

}

}

1) What is the T:I:O breakdown?

2) Assume that we start with a cold cache from the start of main(), what data does the cache contain after the first line (the line before Loop 1) is executed? What is the hit rate for this line?

3) Now consider the loop. How many memory accesses occur per iteration of Loop 1?

4) During each iteration of Loop 1 when we access ARRAY[i], what other data is loaded in? Do we get ever hits because of it?

5) What is the hit rate for Loop 1? Keep in mind that the cache is no longer cold when we enter Loop 1.

6) What is the hit rate for Loop 2 given that the cache is NOT reset after Loop 1?

7) Assume that accessing memory takes 100 cycles, accessing data that is in the cache takes 5 cycles, Also assume for this part that Loop 1’s hit rate is 60% and Loop 2’s hit rate is 75%, which may or may not be the correct hit rates. What is the average memory access time (AMAT) in cycles for (Please reduce fractions):

i) Loop 1:

ii) Loop 2:

iii) Overall AMAT of Loop 1 2 (an expression of REDUCED fractions is alright. You may use “T1” as the Loop 1 AMAT and “T2” as the Loop 2 AMAT in your calculation of this value):