Project 4

Bohao Tang

April 15, 2018

1. Describe your map/reduce algorithm for solving the three's company problem.

a. Describe the operation of the mapper and reducer. How does this combination solve the three's company problem?

Answer: My mapper will iteratly read each line(file). For every single read (that's a line or a file), the mapper will first split the string it read to identify the IDs for friends and host in this line(file), then do looping to find all possible triples of IDs such that it satisfies following requirement:

1. **A, B, C are all in the same line(file). Therefore we only need data from each line separatly to run the program**
2. **A, B, C are all different and B < C. These triples are all possible friends triangles in the sense of having correct format**
3. **One of A, B, C is the first id in the line(or in the file), that's the file name (or the host for this friends list). This means for every legal , two of people here must be friends to the rest one**

If <A, B, C> (in file with host X) satisfies these three requirements. We call <A, B, C> a legal triangle (suggest by X).

Those legal triangles are the keys and their values are all one. Then the mapper emit all these key-value pairs. Notice that since IDs in one file are always different from each other, the mapper will emit all possible `<A, B, C> - one` pair at most once.

My reducer does an aggregating job. It simply sums all values that have the same key and reduces these pairs to one with the same key and sum for value. And whenever this value of the reduced pair is bigger than one, it prints out the key(in format of `A B C`). In the streaming version, since the key-value emit by mapper will be sorted. All replicated keys will appear continuously. Therefore we only need to add serialy and don't need to introduce big aux data structure (see the code).

Here we prove the combination of this two will solve the problem.

First. The format of output is right. Because of reducer reducing all pairs with same keys to one and the second requirement. Output `A B C` will not be duplicated, A,B,C all different and `B < C`.

Second. For every true friends triangle `<A, B, C>`, since they are all friends of each other. `<A, B, C>` is a legal triangle(satisfies all three requirements) suggest by all `A` and `B` and `C`. Therefore `<A, B, C> - one` pair will be emit at least three times in the mapper, which means the reducer will output this triangle.

Third. For every triple `<A, B, C>` with the right format but not a true friends triangle. There must exists one from this three that the other two are not all his friends("mutual friends" just means "for every people, the others are his friends"). For example we suppose `A`, `B` are friends but `A`, `C` are not (other situations can be discussed in the same way and get the same result). Then `A` and `C` will not suggest `<A, B, C>` triple because of **requirement 1**. Also, since **requirement 3**, files other than `A, B, C` will not suggest `<A, B, C>` either. Therefore in reducer, `<A, B, C>` will at most appear once(suggest by `B`), which means the map-reduce algorithm will not output `A B C`.

Combine these three arguments, we proved that the algorithm will and will only output true friends triangles in the right format.

b. What is the potential parallelism? How many mappers does you implementation allow for? Reducers?

Answer: The mapper work only needs one line(file) input in a run, and the reducer work only need all pairs with same key separatly. Therefore you can run multiple mappers to deal with different lines(files) and multiple reducers to deal with different keys simultaneously. That's the potential parallelism. Therefore we can have at most M mappers and N reducers in this implementation, where M is the numbers of lines(files) and N is the numbers of all legal triangles. (Recall that we defined **legal triangle** to be the triple `<A, B, C>` that satisfies: **(1) A, B, C are all in the same line(file), (2) A, B, C are all different and B < C, (3) One of A, B, C is the first id(host) in the line(in the file)**)

2. On combiners

a. Why do we leave the combiner class undefined?

Answer: Because a combiner is actually a reducer that will only accept key-value pairs emit from each single mapper separatly. But in my implementation, we only need to count the times that a key emit by different mappers. Since one mapper will at most emit a key once, you will never gain any information unless you get output from more than one mappers. Therefore the combiner is useless in this situation.

3. Let's gently analyze the total work done by the algorithm.

a. How many messages do your mappers output? Assuming the Hadoop runtime has to sort these messages how much total work does the algorithm do (in Big-O notation)? You should assume that there are n friends list each of which are of length O(n).

Answer: Mappers will output all legal(possible) friends triangle defined above. For every file(each mapper), this amount will be $O(F^2)$, where $F = O(N)$ is the number of friends the host have. Therefore for the whole data of N people, the mappers will output $O(N * F^2) = O(N^3)$ messages. The sorting method in Hadoop runtime can only be based on comparison because in general Hadoop will never know at beginning what kind of keys we will emit. (Actually the sorting method used is Quicksort in Mapper and Mergesort in reducer) Therefore the sorting will introduce at least $O(M \log(M)) = O(N^3 \log(N^3) + N^3) = O(N^3 \log(N))$ work, where M is the amount of messages output by mappers. Then in reducers, they just do an aggregate counting job, which can be done by a single scan. That is $O(M) = O(N^3)$ amount of work. Therefore for the whole M/R algorithm, total work is $O(N^3 \log(N))$.

b. How does this compare with the performance of serial implementations linked here? Describe the tradeoff between complexity and parallelism (Note: it is more reasonable to compare your implementation with the naive $O(n^3)$ algorithm than the optimized $O(n^{2.37})$ algorithm.)

Answer: For the serial implementation of $O(N^3)$, you can just emit keys like in my solution, then **don't sort** the output and finally scan it to aggregate the keys to find friends triangles. Since you don't do sort, you will do less job compared to my implementation, which is $O(N^3 \log(N))$. Therefore our implementation have higher complexity. But if you don't sort the keys output by mappers, M/R architecture will never know how to distribute work(those key-value pairs) to multiple different reducers (therefore no parallelism). Since now a key can appear in any place of the mapper messages, but we need to send pairs with same keys to same reducer to have the correct output. This is a tradeoff between complexity and parallelism.