Ferhat Elmas 214805

Advanced Databases Hw-7

1-) I think this friendship as a big graph and I try to count distinct paths between person X that is the person I am interested in and other each person A where A is accessible(has friends) and isn't friend with X. Then, I sort the list in terms of the number of the paths and output the greatest 10 people that may connect to X.

We need two map/reduce jobs, one is to compute distinct paths and the other is to sort the list.

First job

Map: I do a cartesian product for all pairs of friends in friend list of person X. This enables us to see whether people who aren't connected directly are connected by indirectly. I also need to eliminate directly connected pairs because they are already friends. However, these pairs are needed for the calculation with the other pairs, map should also output directly connected pairs. Therefore, to keep an order in which directly connected pairs come before others, I use a flag.

- Input: [(person), (friend list)]
- Output: [(person1, person2, flag), 1]

Reduce: Reducer input will be sorted so if first pair is a directly connected pair then they are already friends. Otherwise, reducer sums the counts in the list.

- Input: [(person1, person2, flag), (count list)]
- Output: [(person1, person2), (count)]

Pseudo Code:

```
map(person, friend_list)
    foreach person1 in friend_list
        emit([person, person1, 0])
        foreach person2 > person1 in friend_list
            emit([person1, person2, 1], 1)
```

Here, we need a partitioner not to use flag value while choosing a reducer.

```
reduce([person1, person2, flag], [count_list])
    if person1 not current_person1 or person2 not current_person2
        current_person1 ← person1
        current_person2 ← person2
        skip ← flag is 0
    unless skip
        count_sum = 0
        foreach count in count_list
            count_sum ← count_sum + count
        emit([person1, person2], count_sum)
```

Second Job

Map: Input records are rearranged to go to the reducer in sorted order. This rearrangement is to utilize of the feature of the map/reduce framework, keys are sorted.

- Input: [(person1, person2), (count_sum)]
- Output: [(person1, count_sum), (person2)]

Reduce: All combinations will be list of the reducer in sorted order so reducer only needs to get the greatest 10.

- Input: [(person1, count_sum), (person_list)]
- Output: [(person1), (list_of_[person2, count])]

Pseudo Code:

```
map([person1, person2], count)
    emit([person1, count], person2)
    emit([person2, count], person1)
```

Here, we need a partitioner to care only person part of the key while selecting a reducer.

```
reduce([person1, count], person_list)
    if person1 not current_person or last record
        emit(current_person, top_ten_list)
        current_person ← person1

    if size of top_ten < 10
        foreach person2 in person_list
            add [person2, count] into top_ten
            if size of top_ten = 10
                break
```

2-) a- NLJ is default join algorithm because we have windows and with NLJ, we iterate over the all tuples in each stream. Otherwise, invalidation and insertion require further processing but now we can easily invalidate tuples at the start in the number of arrivals and insert at the end. Moreover, stream speeds can differ and in search heavy workloads, using only NLJ can decrease performance dramatically so it should be used via index such as B+ tree. Tree index is better since hash index can only handle equality joins.

b- In the paper, they suggest asymmetric join algorithms and their cost formulas are independent from the other selected algorithms. Therefore, characteristics of the stream determines the join algorithm. For example is stream update or search load? Moreover, their relative speed also matters. If we have equal window sizes for two streams, there are four tipping points which means 5 winning algorithms according to ratio of the skewness between streams. If upload load is so skew, TN is better. TH is better until arrival rate of stream A is a quarter of the stream B. HH is better when stream speeds are swapped. And, HT and NT can be used in the rest of combinations.

3-) a- When we do the calculations we see HT or TH has the lowest cost since streams are symmetric.

b- In this case, only HT is calculated as the optimum.

c- HT

d- TH
e- HT
f- TH
Since we do comparison and speed are same, we can exclude them from formulas, then only some window calculations are left and when do the calculation, we get above combinations.