Exam Solutions

Ahmed Alhasan 'ahmal787'

2020-06-02

Q11

- one such a application is search engine where the system need to take care of concurrent requests per second, Network saturation, read and write scalability and data scalability

Q12

-a) we iterate over all key idecies and for each key we look for first value that match "Alice"

-b) by adding the birth dates of all users in every key, we dont need to iterate but it will add a lot of redundancy

Q13

Q14

- Yes, because it is done locally it reduces communication time between nodes and it will improve load balancing where if a node finishes aggreagating locally it can take another task

Q15

-a) because we can break down every problem into map-reduce tasks that can be run independently on local nodes.

-b) in Spark the map-reduce pattern is done through RDD system which do lazy evaluation of transformations where they are just added to the graph and not materialize until an Action is reached and that gives more flexibility to the scheduler with better data locality and persisting the data in memory

Q16

- in MapReduce the data have to written/stored to disk every time we perform a MapReduce operation and that means we need to maintain a number of replicas to improve fault tolerance.
- while in Spark because we can recompute a task from available, earlier computed data blocks it improve fault tolerance by reducing the need to have the data stored on disk

Q17

let T_r is the time needed by the reducer and T_m is the time needed by a mapper and M is the number of mappers

$$\text{Time } T = \ T_r + \frac{T_m}{M}$$

the relative speedup S_{rel} is:

$$S_{rel} = \frac{T(1)}{T} = \frac{T(1)}{T_r + T_m(M)}$$

where T(1) is the time to excute single map or reduce operation

and if we let β as the ratio between work done by the reducer over the total work

$$\beta = \frac{w_r}{w} \leq 1$$

then relative speed up of M mappers is limited by:

$$S_{rel} = \frac{T(1)}{\beta T(1) + (1-\beta)T(1)/M} = \frac{M}{\beta M + (1-\beta)} < \frac{1}{\beta}$$

Q18

```
# mapped in the following format (x1, t1),..., (xN, tN)
# and persisting in memory for faster access
data = sc.textFile(...).map(...).persist()

# Intial random weights
w = np.random.ranf(size = D)
alpha = Constant

# Sum the partial gradients and update w accordingly by Reduce
for i in range(iterations):
  w -= data.map(lambda x: x[0] * x[1]) \ # take the product of x_n*t_n
           .reduce(lambda a,b: a+b)) \   # take the sumation of all misclassified instances in one
           .map(lambda x: alpha * x)     # multiply the summation with alpha for one iteration

# -= ensures that we are moving towards the minima by sustracting from the old w
```