Question 1

1) Inputs and Outputs of the Map and Reduce functions.

The map function extracts the domain name and number of bytes from the string and sends the number of bytes keyed by domain name on. There is no need for an input key, though the line number could be used if necessary.

**Map:** (url: string) -> (domain: string, byte\_count: int)

The reduce function recieves a list of byte counts for each domain. It sums up the list for each domain into a combined byte count.

**Reduce:** (domain: string, byte\_counts: List(int)) -> (domain: string, total\_bytes: int)

2) Pseudo-code for Map and Reduce

**def** Map(text):

domain, byte\_count = parse(text)

**yield** domain, byte\_count

**def** Reduce(domain, byte\_counts):

**yield** domain, sum(byte\_counts)

The "sum" function refers to adding up all the byte counts.

The "parse" function refers to extracting the domain name and byte count from the input text line. This could be done in different ways. One relatively easy way to do it in python is with regex, as follows:

pattern = r"https:\/\/([^\/]\*)\/.\*,\s\*(\d+).\*,.\*"

domain, byte\_count = re.match(pattern, text).groups()

Question 2

1) Inputs and Outputs of the Map and Reduce functions.

I'm choosing to work with matrices in sparse form here. (Not that that doesn't mean the matrices have to be sparse, just that they are stored in coordinate/value pairs). This has the advantage that they can be read and processed as streams without needing to be fully stored in one place in memory.

e.g.

the matrix:

Is stored as:

((1, 1), 1)

((1, 2), 2)

((2, 1), 3)

((2, 2), 4)

The algorithm I'm using for this question is loosely based on an article from the GeeksforGeeks website[1].

The map function takes coordinate/value pairs. The coordinates are an i coordinate, a j coordinate, and we need a way of identifying which matrix it came from, since the operation is not symmetric. The output is one *or more* values keyed by the target cell in the matrix M they are relevant to.

**Map:** ((source: string, i: float, j: float), value: float) -> List((i: float, j: float), value: float)

The reduce function recieves the values for a target cell (i,j) in the result matrix M. It does not need to know the source. The output will be a coordinate/value pair for the result matrix.

**Reduce:** ((i: float, j: float), values: List(float)) -> ((i: float, j: float), value: float)

Question 2 (Cont.)

2) Python pseudocode for the map and reduce functions.

N is considered a global constant here. "a" and "b" are used as source for entries in the matrices A and B respectively. Finally, note that Map yields *multiple values per input record.*

**def** Map(key, value):

source, (n, m) = key

**if** source == "a":

i = n

k = m

**for** j **in** range(1, N+1):

**yield** (i, j), (k, value)

**else**:

k = n

j = m

**for** i **in** range(1, N+1):

**yield** (i, j), (k, value)

**def** Reduce(key, values):

terms = [1]\*N # Create a length N list

**for** k, factor **in** values:

terms[k-1] \*= factor

**yield** key, sum(terms)

3) Large matrices

My answer for Q3 is the same as for Q2 - the above algorithm does allow for matrices that are too big to fit in memory. The mappers are O(1) in space and the reducers are O(N) in space as opposed to the O(N^2) required to store the matrix.

However! The reducers being O(N) does mean that there is a significant limit to the horizontal scaling, which is not desirable. I can see a way to keep things O(1) using an extra reduce step - the first one multiplies matching factors, the second sums those factors up. I do not see, as of writing this, a way to combine those two steps into a single reduce.

Question 3

1) Inputs and Outputs of the Map and Reduce functions.

First of all, I'm only considering mapreduce for a *single* iterations of kmeans. I'll include an algorithm for iterating over this at the end. I don't think it is sensible to use a single mapreduce pass for the entire algorithm, a mapreduce pass isn't made for cycles.

With this in mind, one way to look at kmeans is that we are creating groups which are entirely defined by their centroid. Since each point is part of the group with the closest centroid, given the centroids we can determine which point is in each group. Each iteration, we input the points and centroids and get an updated list of centroids.

The algorithm here is based on [2]. It is horizontally scalable in the number of points, but not in the number of groups.

This algorithm is defined for some type "T", for which the following operations have been defined:

* distance: (a: T, b: T) -> float
* mean: List(T) -> T

**Map:** (centroids: List(T), point: T) -> (centroid\_index: int, point: T)

**Reduce:** (index: int points: List(T)) -> (index: int, centroid: T)

The map function needs *all* all the centroids - i.e. each mapper node must maintain a list of centroids that is updated each iteration. Hence, it is scalable in N but not in k.

Question 3 (Cont)

2) Python pseudocode for the Map and Reduce functions.

**def** Map(centroids, point):

min\_distance = MAX\_VALUE

**for** i, centroid **in** enumerate(centroids):

d = distance(point, centroid)

**if** d < min\_distance:

min\_distance = d

closest\_index = i

**yield** closest\_index, point

**def** Reduce(index, values):

**yield** index, mean(values)

Mean and distance are assumed to be defined as described in part 1.

Question 3 (Cont)

Appendix: combining the iterations

The map reduce step above is for a single iteration of kmeans. Below is pseudocode for combining iterations.

**def** kmeans(points, initial\_centroids, max\_iterations):

N = len(points)

k = len(initial\_centroids)

centroids = initial\_centroids.copy()

**for** i **in** range(max\_iterations):

result = map\_reduce(points, Map, Reduce, context=centroids)

any\_changes = **False**

**for** ki, (x, y) **in** result:

**if** centroids[ki] != (x, y):

any\_changes = **True**

centroids[ki] = (x, y)

**if** not any\_changes:

**break**

**return** centroids

"mapreduce" is a made-up function in the above, and the context bit left fairly vague for the sake of keeping things short.

# Bibliography

|  |  |
| --- | --- |
| [1] | "Matrix Multiplication With 1 MapReduce Step," GeeksforGeeks, 12 11 2020. [Online]. Available: https://www.geeksforgeeks.org/matrix-multiplication-with-1-mapreduce-step/. [Accessed 16 10 2021]. |
| [2] | W. H. M. Q. H. Zhao, "Parallel k-means clustering based on mapreduce," *IEEE international conference on cloud computing,* pp. 674-679, 2009. |