828G Final Exam

Joshua G. Bradley

May 16, 2013

I pledge on my honor that I have not given or received any unauthorized assistance on this examination.

1

1.A

Algorithm 1.A.1 : Job 1 - Compute frequency counts

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Combiner
2:     method Reduce(term t, counts [c_1, c_2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c_1, c_2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)

1: class Reducer
2:     method Reduce(term t, counts [c_1, c_2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c_1, c_2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

Algorithm 1.A.2 : Job 2 - Assign unique integer mapping

```
1: class Mapper
2:     method Map(term t, count c)
3:         Emit(pair < 1, count c >, term t)

1: class Partitioner
2:     method Partition(pair key<id, count>, term t, int NumReducers)
3:         Return key.id % NumReducers

1: class Reducer
2:     method Init
3:         uniqueCounter ← GET(TotalUniqueTerms)
4:     method Reduce(pair key<id, count c>, terms [t_1, t_2, ...])
5:         for all term t ∈ terms [t_1, t_2, ...] do
6:             Emit(term t, count uniqueCounter)
7:             uniqueCounter ← uniqueCounter − 1
```

This MapReduce algorithm requires two jobs. The first job computes the frequency counts of every term and the second job relies on the execution framework to sort the frequency counts.

As can be seen, the first job is essentially a variant of the word count algorithm. It is possible for a combiner to be used in the first job, in order to reduce the amount of intermediate data generated and shuffled/sorted. Because we are only aggregating frequency counts (simple addition) in this job, the Combiner and Reducer are exactly the same (in terms of code). For the first job, there are no apparent bottlenecks in terms of memory. A lot of intermediate data will however be generated. Use of the combiner on both the Map side and Reduce side of the job will greatly reduce (no pun intended ☺) this amount.

In the second job, value-to-key conversion is used to enable secondary sorting on the key. The frequency count of each term is placed into the key in order to allow the execution framework to handle the sorting for us, essentially ordering all terms by increasing frequency count. Since the first job aggregates all frequency counts together, it is guaranteed in the second job that the mappers will only see one unique instance of each term (no two mappers will emit a key-value pair with the same term). It is assumed that *TotalUniqueTerms* is a global parameter that would be set in the job configuration of the second job after the first job finished. This number could

easily be obtained from the "number of records written to output by reducers" counter that is provided by default. A partitioner is used to send all key-value pairs to the same reducer based on the left element of the key. It is important to initialize the *uniqueCounter* in the Init() method so that the reducer can preserve the state of the counter over multiple keys. There are no apparent memory bottlenecks in this job, however there is a runtime bottleneck that occurs because we are sending all keys to the same reducer. The runtime of the reducer will thus be dependent on the size of the vocabulary in the collection. This implementation is still better than a non-MR implementation though since a non-MR implementation would require an in-memory sort of all terms and their frequency counts (which would not be possible for large datasets) before assigning a unique integer to each term.

1.B

Algorithm 1.B.1 : Job 1 - Find all unique terms

```
1: class Mapper
2:     method Init
3:         H ← ∅
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             if t ∉ H then
7:                 Emit(term t, count 1)
8:                 H.add(t)

1: class Combiner
2:     method Reduce(term t, counts [c_1, c_2, ...])
3:         Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c_1, c_2, ...])
3:         Emit(term t, count 1)
```

Algorithm 1.B.2 : Job 2 - Assign unique integer mapping

```
1: class Mapper
2:     method Map(term t, count c)
3:         Emit(value 1, term t)

1: class Reducer
2:     method Init
3:         uniqueCounter ← 1
4:     method Reduce(value key, terms [t_1, t_2, ...])
5:         for all term t ∈ terms [t_1, t_2, ...] do
6:             Emit(term t, count uniqueCounter)
7:             uniqueCounter ← uniqueCounter + 1
```

This MapReduce algorithm requires only one job, however on very large amounts of data, it would be better to break the algorithm down into two jobs.

We are not worried about assigning an integer mapping based on frequency distribution in this algorithm, thus the problem becomes much easier. In the first job, we just want to find all unique terms. Based on the algorithm outlined above, it is easy to notice that there is a potential bottleneck in the mapper. In an attempt to decrease the amount of intermediate data generated by the mappers, a variant of in-mapper combining can be used to simply store every unique term encountered in a document and emit only when a new term has been found in the document. In the worst case scenario of scalability, this means the mapper will be limited by the size of the vocabulary space. One approach that could be used in practice (which was left out of the pseudocode above for clarity's sake) would be to clear out the set H when the amount of memory required to hold it becomes too great. This would present the potential that duplicate terms over several documents could be emitted (remember, H is preserved over state), however it would not greatly effect the overall performance of the algorithm or its final results. The only thing to be effected in this case would be the generation of extra intermediate data. I would consider this a reasonable assumption in practice though, thus it would likely not be a problem. This will greatly decrease the amount of intermediate data generated. Also, a combiner can be employed in a second attempt at *uniquefying* the data. Lastly, in a third attempt to produce a list of unique terms, the reducer will only emit one instance of the term. At this point, it is guaranteed that the output from all reducers will have no

duplicate terms.

In the second job, we want to assign an integer to every unique term. Just like the previous problem, at this point we are guaranteed that the incoming data has no duplicate terms (so the use of a Combiner would be pointless). We are not worried about assignment order, thus there is no need for a complex key or special partitioner to be defined. We then force all terms to be sent to the same reducer by emitting a static value of 1 as the key in the mapper. In the reducer, a *uniqueCounter* is initialized in the Init() method and an integer is assigned to every term. There is no apparent memory bottleneck in this job, however there is a runtime bottleneck where the reducer must process all terms. Multiple mappers may be used in this job, but only 1 reducer will do the actual integer assignment. Thus runtime scalability in the reducer is limited by the size of the vocabulary.

1.C

There are several advantages to representing a text collection by ordering term ids via decreasing frequency. With the most frequently occurring term having an id of 1, it would be quite easy to build a list of k stop words by grabbing all ids in the range $[1, k]$. Stop word lists are used in many NLP applications to improve prediction/classification accuracy. Representation of the corpus as unique integers serves two purposes then ... to provide a compact representation of distinct words and the corpus as a whole and to represent their *approximate* relative frequency in relation to other words in the corpus. With the compact form, processing text on HDFS becomes less of a problem in terms of memory requirements and scalability.

2

2.A

Algorithm 2.A.1 : Calculating the mode

```
1: class Mapper
2:     method Map(string s, value v)
3:         Emit(string s, pair < v, count 1 >)

1: class Combiner
2:     method Reduce(string s, pairs [<value_1, count_1>,<value_2, count_2>, ...])
3:         H ← new AssociativeArray
4:         for all M ∈ [< value_1, count_1 >, < value_2, count_2 >, ...] do
5:             H{M.value} ← H{M.value} + M.count
6:         for all keys k ∈ H do
7:             Emit(string s, pair < value k, count H{k} >)

1: class Reducer
2:     method Reduce(string s, pairs [<value_1, count_1>,<value_2, count_2>, ...])
3:         H ← new AssociativeArray
4:         for all M ∈ [< value_1, count_1 >, < value_2, count_2 >, ...] do
5:             H{M.value} ← H{M.value} + M.count
6:         maxOccurence ← 0
7:         mode ← 0
8:         for all keys k ∈ H do
9:             if H{k} > maxOccurence then
10:                mode ← k
11:                maxOccurence ← H{k}
12:        Emit(string s, value mode)
```

This MapReduce algorithm requires only one job. In general, I believe there are two approaches that could be taken with this problem. One could build up a list of all values associated with the same key and then simply call the primitive Mode() function in the reducer. This however presents a bottleneck in memory since the entire list of values associated with the same key must be assumed to be in memory in order to use the primitive Mode() function. To create a more scalable approach to this problem, I believe it is better to store frequency counts of each unique value. A combiner can also be used

to aggregate frequency counts in order to reduce the amount of intermediate data generated. Down in the reducer, all values and their frequency counts corresponding to the same string s are aggregated and a search for the value with the highest frequency is performed.

One potential scalability bottleneck that exist is the reducers must be able to store all *unique* values for a particular string in memory, along with their frequency count. This brings up an issue where the algorithm design should change depending on what kind of data the mode is being calculated on. If most of the strings in the data have duplicate values associated with them (i.e. the number 5 is associated with the same key multiple times), then the algorithm outlined above is the superior approach. However, if a majority of the strings in the data have mostly unique values associated with them (meaning a value only occurs once for each string), then there could be a memory issue. This occurs because using the approach above, there is a (value, count) pair associated with every value found for each string. Thus 2 numbers are stored to account for every unique value for each string. If one were to simply maintain a list of all values associated with a particular string, then only 1 number (the value itself) is stored to account for each value per string. In the data, if a majority of the strings only have one occurrence of a value, the approach above would require roughly twice as much memory in comparison to just maintaining a list of all the values associated for a particular string. In general though, I believe the more scalable approach is to store frequency counts, as outlined in the algorithm above.

2.B

Algorithm 2.B.1 : Calculating the median

```
1: class Mapper
2:     method Map(string s, value v)
3:         Emit(pair<string s, value v>, count 1)

1: class Combiner
2:     method Reduce(pair<s, v>, counts [count_1, count_2, ...])
3:         sum = 0
4:         for all count c ∈ counts [count_1, count_2, ...] do
5:             sum ← sum + c
6:         Emit(pair<s, v>, count sum)

1: class Partitioner
2:     method Partition(pair key<string s, value v>, count c, int NumReducers)
3:         Return key.s.hashcode() % NumReducers

1: class Reducer
2:     method Init
3:         H ← ∅
4:         total ← 0
5:         index ← 0
6:     method Reduce(pair<s, v>, counts [count_1, count_2, ...])
7:         freq ← 0
8:         for all count c ∈ counts [count_1, count_2, ...] do
9:             freq ← freq + c
10:        total ← total + freq
11:        H.add(pair < v, freq >)
12:    method Close
13:        index ← 0
14:        median ← 0
15:        medianIndex ← total/2
16:        if Odd(total) then
17:            medianIndex ← medianIndex + 1
18:        for all pair < v, c > ∈ H do
19:            if index + c < medianIndex then
20:                index ← index + c
21:            else
22:                median ← v
23:                break
24:        Emit(string s, value median)
```

This MapReduce algorithm only requires one job. In the mapper, value-to-key conversion is done so that all values for a particular string are sorted by the execution framework. Since a complex key is used, we must create a partitioner that partitions on the left element of the key (the actual string). A Combiner is used to aggregate frequency counts of the same value for the same string when possible. This will reduce some of the intermediate data generated by the Mapper. For the Reducer, values will arrive in sorted order, which allows us to build up a *sorted* list of all values and their corresponding frequency for a particular string s. In the pseudocode above, this is done by storing a complex pair in the set H.

Actual calculation for the median is done in the Close() method of the Reducer. Note that the check to determine if *total* is odd is only performed in order to prevent a 1-off error by the algorithm and is not meant to confuse the reader. This approach is similar to the MapReduce algorithm above for finding the mode, in that frequency counts of each unique value for a particular string are stored instead of the entire list of values which could consist of multiple copies of the same value. The argument about why algorithm 2.A.1 is superior and more scalable than the primitive function it imitates can be applied to this algorithm and the primitive MEDIAN() function as well.

There is a slight scalability bottleneck in the reducer because it must store a set H of all unique values and their frequency in memory, however in practice this would be considered a reasonable assumption to make and not be an issue.

2.C

Algorithm 2.C.1 : Calculating the maximum

Calculating the maximum.

```
1: class Mapper
2:     method Map(string s, value v)
3:         Emit(string s, value v)

1: class Combiner
2:     method Reduce(string s, values [v_1, v_2, ...])
3:         max ← 0
4:         for all v ∈ [v_1, v_2, ...] do
5:             if v > max then
6:                 max ← v
7:         Emit(string s, int max)

1: class Reducer
2:     method Reduce(string s, values [v_1, v_2, ...])
3:         max ← 0
4:         for all v ∈ [v_1, v_2, ...] do
5:             if v > max then
6:                 max ← v
7:         Emit(string s, int max)
```

This MapReduce algorithm only requires one job. Use of the primitive Max() function could have been used in both the Reducer and Combiner, however that assumes the list of numbers associated with the same key can fit in memory. The reducer does not necessarily store all values in memory at once (and the same applies to the Combiner since it implements the Reduce class), thus a more scalable solution would be to use an iterator to move through the list of values, keeping track of the maximum value seen so far. Since the Max() operation is both associative and commutative, code for the Reducer and Combiner is exactly the same. The only scalable bottleneck that exist with this implementation would be the amount of intermediate data generated. Similar to the "pairs" approach, much intermediate data is generated, however use of the combiner on the Map side and Reduce side of the job will greatly reduce this problem.

3

3.A

Algorithm 3.A.1 : Finding the 2-hop neighborhood of every node - naïvely

```
1: class Mapper
2:     method Map(nid n, Node N)
3:         for all nid m ∈ N.AdjacencyList do
4:             Emit(nid m, Node N)

1: class Reducer
2:     method Reduce(nid m, Nodes [N_1, N_2, ...])
3:         neighbors ← ∅
4:         for all Node N ∈ [N_1, N_2, ...] do
5:             if N.nid ∉ neighbors then
6:                 neighbors.add(N.nid)
7:             for all nid h ∈ N.AdjacencyList do
8:                 if h ∉ neighbors then
9:                     neighbors.add(h)
10:        Emit(nid m, neighbors)
```

Algorithm 3.A.2 : Finding the 2-hop neighborhood of every node - less naïvely

```
1: class Mapper
2:     method Map(nid n, Node N)
3:         for all nid m ∈ N.AdjacencyList do
4:             Emit(nid m, Node N)

1: class Combiner
2:     method Reduce(nid m, Nodes [N_1, N_2, ...])
3:         H ← new Node
4:         for all Node N ∈ [N_1, N_2, ...] do
5:             if N.nid ∉ H.adjacencylist then
6:                 H.adjacencylist.add(N.nid)
7:             for all nid h ∈ N.AdjacencyList do
8:                 if h ∉ H.adjacencylist then
9:                     H.adjacencylist.add(h)
10:        Emit(nid m, H)

1: class Reducer
2:     method Reduce(nid m, Nodes [N_1, N_2, ...])
3:         neighbors ← ∅
4:         for all Node N ∈ [N_1, N_2, ...] do
5:             if N.nid ∉ neighbors then
6:                 neighbors.add(N.nid)
7:             for all nid h ∈ N.AdjacencyList do
8:                 if h ∉ neighbors then
9:                     neighbors.add(h)
10:        Emit(nid m, neighbors)
```

The naïve MapReduce algorithm is shown in algorithm 3.A.1 and requires only one job. For the sake of clarity, it is assumed that the Node data structure contains both the adjacency list and node id (denoted as "nid" in the pseudocode for shortness) for a particular node in the graph.

One scalability issue that arises with this implementation is the amount of intermediate data generated. 1 copy of a nodes' data structure is being sent over the network to each of its neighbors. To understand the less naïve approach in algorithm 3.A.2, one must first realize that all we essentially are trying to do is aggregate adjacency lists out to a 2-hop distance. This

is important because the concept of a "node" is lost in the combiner when adjacency lists are aggregated. However use of the Combiner reduces the amount of intermediate data being sent over the network, in the sense that the adjacency lists in the Node data structures that are all being sent to to the same key in the reducer are potentially de-duped by the Combiner. The performance gained from the Combiner in algorithm 3.A.2 will be quite significant in very dense graphs where adjacency lists will be large.

One last scalability issue can be found in the Reducer stage, when the list of node id's corresponding to a nodes' 2-hop neighbors is buffered in memory while being built. This would only become a problem on very large dense graphs (of the order of a billion nodes). If each integer is 4 bytes and we were working with a dense million-node graph, this would correspond to a neighbor list requiring at most ≈ 3.8 MB of space. Once you begin processing billion-node graphs, this raises the memory requirement to ≈ 3.8 GB. For commodity machines, this would be around the maximum graph size that could be handled. Machines in a data center are often provided with more RAM however, thus would be more scalable. It seems to be a reasonable assumption that the entire *neighbor* list could be stored in memory since to my knowledge, there are very few (if any) groups out there processing billion-node graphs.

3.B

Algorithm 3.B.1 : Job 1 - Compute local clustering coefficient for every node

```
1: class Mapper
2:     method Map(nid n, Node N)
3:         for all nid m ∈ N.AdjacencyList do
4:             Emit(nid m, Node N)

1: class Reducer
2:     method Reduce(nid m, Nodes [N_1, N_2, ...])
3:         neighbors ← ∅
4:         total ← 0
5:         for all Node N ∈ [N_1, N_2, ...] do
6:             neighbors.add(N.nid)
7:             total ← total + 1
8:         edges ← 0
9:         for all nid h ∈ neighbors do
10:            for all Node N ∈ [N_1, N_2, ...] do
11:                if h ∈ N.AdjacencyList then
12:                    edges ← edges + 1
13:        edges ← edges / 2
14:        coefficient ← edges / total
15:        Emit(nid m, coefficient)
```

Algorithm 3.B.2 : Job 2 - Compute average clustering coefficient

```
1: class Mapper
2:     method Map(nid n, value coefficient)
3:         Emit(1, pair <coefficient, 1>)

1: class Combiner
2:     method Reduce(key, values[<coefficient_1, count_1>,<coefficient_2, count_2>...])
3:         sum ← 0
4:         count ← 0
5:         for all pair v ∈ values do
6:             sum ← sum + v.coefficient
7:             count ← count + v.count
8:         Emit(key, pair <sum, count>)

1: class Reducer
2:     method Reduce(key, values[<coefficient_1, count_1>,<coefficient_2, count_2>...])
3:         sum ← 0
4:         count ← 0
5:         for all pair v ∈ values do
6:             sum ← sum + v.coefficient
7:             count ← count + v.count
8:         Emit(key, sum/count)
```

This MapReduce algorithm requires two jobs. The first job is responsible for calculating the local clustering coefficient for every node in the graph and the second job computes the average clustering coefficient over the entire graph. Unlike algorithm 3.A.2, a Combiner cannot be used in job 1 the same way because it is necessary to maintain the adjacency list of each node in order to count the number of edges between two friends (nodes). Note that in the Reducer, double counting is performed when counting the number of edges that have been "realized" between two friends due to the fact that the adjacency lists have reciprocal directional edges. Thus the number of edges is halved before the local clustering coefficient is calculated. Since only the node id of all 1-hop neighbors is being stored in memory, it is considered a reasonable assumption and would be scalable.

For the second job, we send each local clustering coefficient to one reducer and sum over all coefficients. Calculating the Average() is normally not commutative and associative, however by keeping track of values and counts

separately through a complex value pair, we are able to turn it into a commutative and associative operation. Thus the Combiner and Reducer share mostly the same code with the exception of the output in the reducer finally computing the average clustering coefficient as a fraction. The scalability bottleneck that exist here is a runtime issue, not memory. The runtime of the reducer will be limited by the number of nodes in the graph.

3.C

Algorithm 3.A.1 is a naïve approach, because as previously mentioned, given a certain node with f friends, there are f copies of the Node data structure sent over the network. This means for a graph of size n where b is the average number of friends someone has, there will be $\approx b \times n$ Node structures generated as intermediate data. Within each Node structure, there is assumed to be a node id and an adjacency list of neighbor node ids. Thus each Node structure will require approximately $(b+1) \times 4 + c$ bytes of space (assuming every integer is represented by 4 bytes) where c is the number of bytes required to store the data as an "object". Therefore, it follows that there will be $\approx b \times n \times ((b+1) \times 4 + c) = 4b^2n + 4bn + cbn$ bytes of intermediate data generated. Using Facebook as an example, it we assumed each person had on average $b = 350$ friends and it only required $c = 8$ bytes to store java objects, then to compute the clustering coefficient for a graph of just $n = 50000$ people (which is quite small in the world of Facebook), that would equate to ≈ 23 GB of generated intermediate data being shuffled/sorted over the network.

3.D

To compute the *approximate* average clustering coefficient, we can do random sampling. This will allow us to process local clustering coefficients much faster and essentially reduce the complexity by an order of magnitude. To do this, we would need to randomly sample the neighbors of a node when counting the total number of "friends" (neighbors) a node has. This would require a modification of lines 5-7 of the Reducer in job 1 in algorithm 3.B.1. Instead of iterating through all neighbors, uniformly randomly select neighbors. Job 2 of this algorithm, summing over all local clustering coefficients to find the average, would not change.