MapReduce

CS121 Parallel Computing

Spring 2019

Overview

- MapReduce is a programming model for processing large amounts of data on a distributed cluster of commodity servers.
- Web and user data, text / image / video, physical / astronomical / biological data.
  - Claim is the answers lie in the data itself, so the more data the better.
  - Aka “data science”, “fourth paradigm” of science.
- Scale “out” to large cluster of cheap networked servers, instead of “up” to small number of high-end shared memory SMPs.
- Ties in well with cloud computing. MapReduce jobs run on dynamically allocated cloud servers.
- MapReduce originally developed by Google. The same functionality is implemented in the open-source Hadoop.
- Widely used in academia and industry due to low cost and ease of use.

Design philosophy

- MapReduce’s design is based on certain assumptions about the hardware and workload.
- Assume failures are common.
  - A 10,000 server cluster is likely to experience multiple failures a day.
- Move processing to data.
  - MapReduce tasks often do simple computations on lots of data. So transfer code to where data is already stored.
- Sequential instead of random data access.
  - Because of its simplicity, MapReduce tasks often make one or a few sequential passes over the data.
- Scalability
  - Many MapReduce tasks scale linearly with number of servers, due to limited communication.
- Many algorithms don’t satisfy these assumptions, and won’t work well on MapReduce.

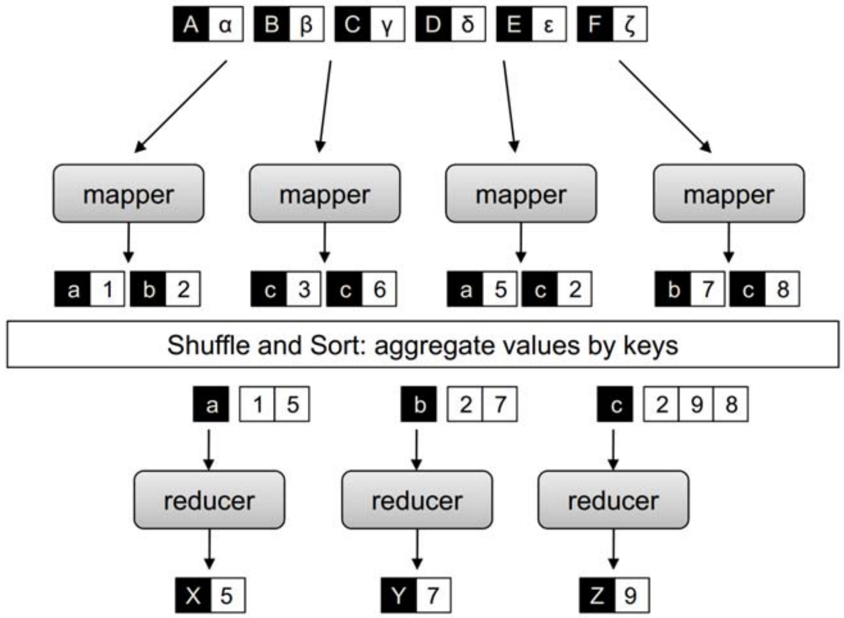
MapReduce model

- The basic data structure in MapReduce is key-value pairs.
  - Keys can be integers, floats, strings, etc.
  - Values can be arbitrary data structures, e.g. tuples, associative arrays, etc.
  - All computations must be expressed in these terms.
  - Ex To process a set of webpages, keys can be URLs and values the page content.
  - Ex To process a graph, keys can be node IDs and values be adjacency lists.
- A MapReduce computation consists of applying a map function followed by a reduce function.
  - map: (k, v) → [(k’, v’)], where [...] denotes a list
  - reduce: (k, [v]) → [(k’, v’)]

MapReduce model

- Inputs to a MapReduce job are initially stored as files in a distributed file system.
- At run time files are distributed to multiple servers each performing map or reduce tasks.
- Each map server executes the map function on all key-value pairs in the files it receives. It outputs a set of key-value pairs.
  - These are called intermediate key-value pairs.
- The MapReduce runtime then performs a “shuffle and sort” stage, which groups all map output pairs with the same key together.
- The grouped pairs are partitioned, and a partition is sent to each reduce server.
  - Pairs for each reduce server are sorted by key.
  - Pairs between different reduce servers may not be sorted.
- Each reduce server executes the reduce function on the key-value pairs it receives. It outputs a set of key-value pairs, which are written persistently to a file.

Model and example



```
class Mapper
    method Map(docid $a$, doc $d$)
        for all term $t \in$ doc $d$ do
            Emit(term $t$, count 1)
class Reducer
    method Reduce(term $t$, counts $[c_1, c_2, \ldots]$)
        $sum \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ do
            $sum \leftarrow sum + c$
        Emit(term $t$, count $sum$)
```

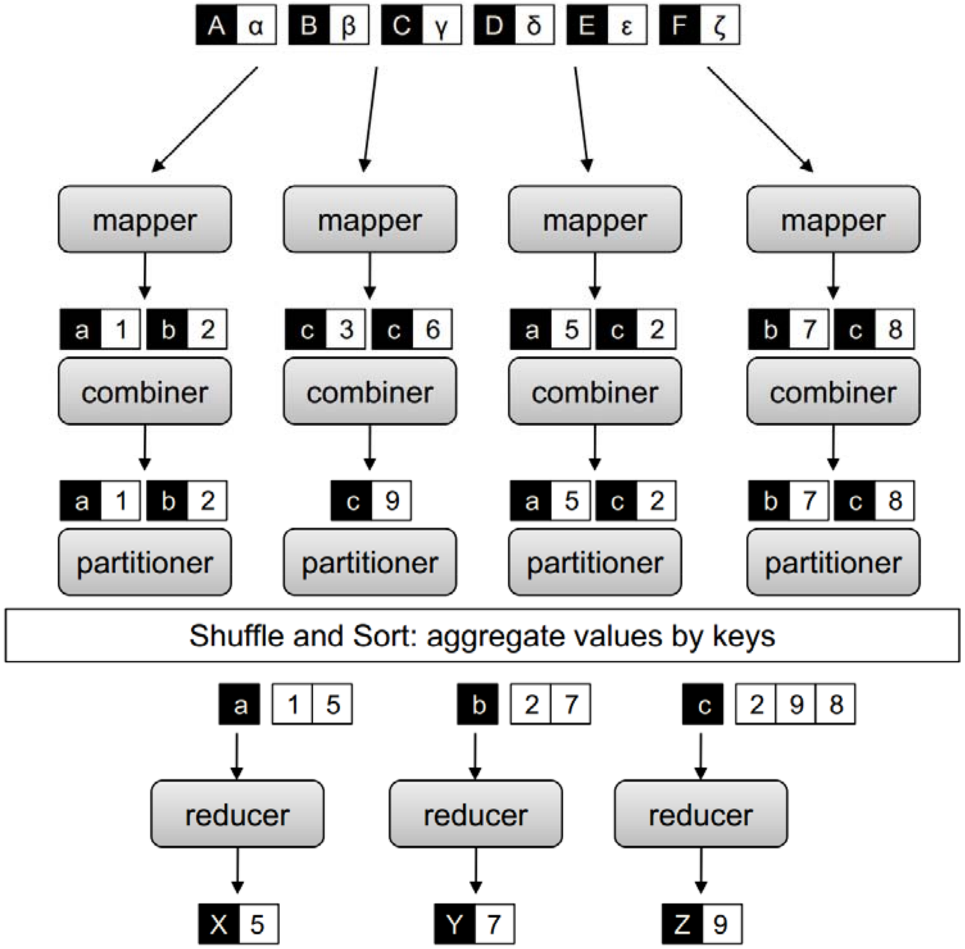
Source: Data-Intensive Text Processing with MapReduce, Lin and Dyer

- Ex MapReduce job for counting the number of occurrences of different words in several documents.
- For each (doc ID, document) input pair, a map task iterates over all words in the document and outputs (word, 1) pairs.
- In shuffle and sort stage, all pairs with the same word are grouped together and sent to the same reduce server.
- Reduce task receives a set of (word, 1) pairs. For each word it counts how many such pairs occurs and outputs a (word, count) pair.

More details

- A MapReduce job can consist of thousands of individual map and reduce tasks.
- The tasks are mapped onto a limited number of servers.
    - Multiple tasks may be queued at a server.
- The reduce tasks can only start after all the map tasks have finished.
    - Performance is sensitive to stragglers, i.e. tasks that take longer than others.
        - Stragglers are caused by poor load balancing, or faulty hardware.
    - But key-value pairs can be shuffled and sorted while some map tasks are still executing.
- MapReduce tries to move code to data. But if a server is already handling too many tasks, MapReduce may move data to a less loaded server.

Partitioners and combiners



- Partitioners and combiners are user defined functions to optimize a job's performance.
- Partitioner uses a function on keys to specify which reducer each intermediate key-value pair gets sent to.
  - Ex It can map (k,v) to h(k) mod r, where h is a hash function and r is the number of reducers.
- Combiners do local key aggregation after mapping and before shuffle and reduce.
  - Ex In word count, combiner can compute local counts on (word, 1) pairs emitted by mapper, and produce one (word, count) for each word.
  - Can be thought of as "mini-reducers".
  - However, there's no guarantee that the runtime will execute the combiner on all (or any) of pairs emitted by mapper.

Task: Filtering

- Process a set of data and keep the ones that satisfy some condition.
  - Used to isolate interesting data, cleaning data, sampling, etc.
- Filtering requires only mappers, not reducers.
- Each mapper simply processes its key-value pairs, and outputs all values satisfying the condition.

Task: Top 10

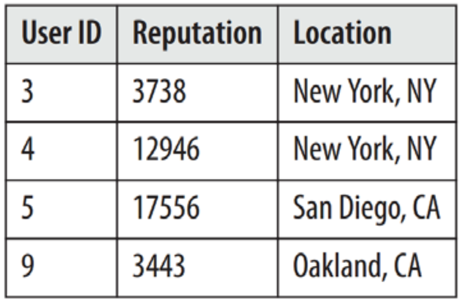
- Apply a ranking function to each input pair and keep the top 10 (or K) highest.
    - Find outliers or most interesting data items.
- Use multiple mapper tasks and one reducer task.
- Each mapper outputs top 10 values among its local inputs, as (NULL, value) pairs.
- All pairs get sent to reducer. It sorts data by value and outputs top 10.
    - Since each mapper produces a small number of outputs, reducer usually will not be overloaded.

Task: Distinct

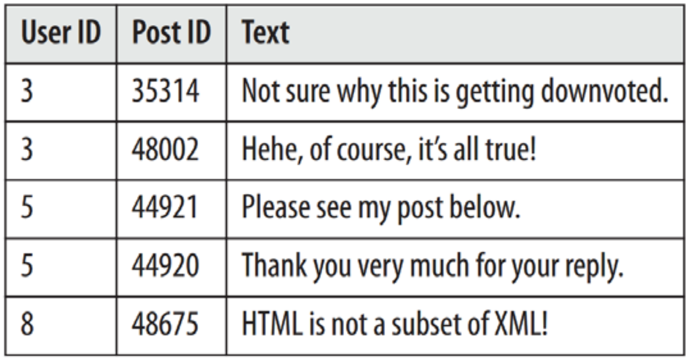
- Find the unique values among input, e.g. for deduplication.
- Each mapper does (key, value) → (value, NULL) and outputs the latter pair.
- All pairs with same value are sent to same reducer. Pairs are sorted by key (i.e. the value of the intermediate pair), so reducer just iterates through its pairs and outputs the distinct keys.
- Can use combiner in mapper to precombine matching values.
- We can use an arbitrary number of mappers and reducers.
  - More reducers improve performance until communication dominates.

Task: Joins

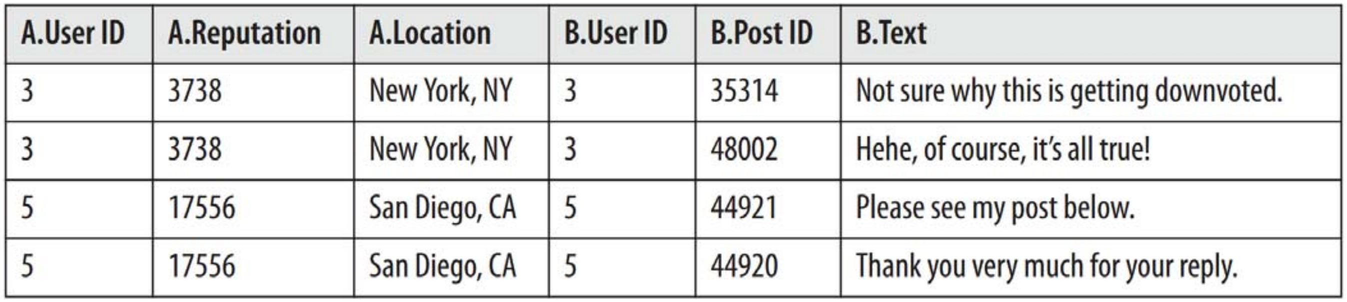
- Joins are used in databases to combine data from two tables.
  - Several types, including inner, left, right, outer, etc.
  - We'll consider inner join.
    - Takes two tables and a predicate as input.
    - Form the Cartesian product of all rows of the two tables, and keep the ones satisfying the predicate.
- Ex Join A and B, s.t. A.(User ID) = B.(User ID)
  - User ID is called the foreign key.



User ID	Reputation	Location
3	3738	New York, NY
4	12946	New York, NY
5	17556	San Diego, CA
9	3443	Oakland, CA



User ID	Post ID	Text
3	35314	Not sure why this is getting downvoted.
3	48002	Hehe, of course, it's all true!
5	44921	Please see my post below.
5	44920	Thank you very much for your reply.
8	48675	HTML is not a subset of XML!



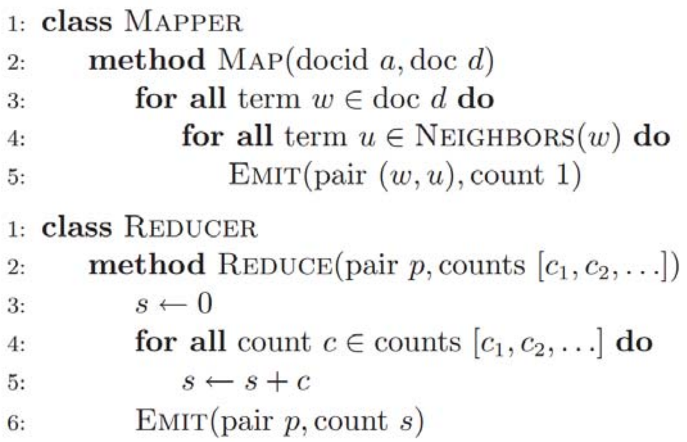
A.User ID	A.Reputation	A.Location	B.User ID	B.Post ID	B.Text
3	3738	New York, NY	3	35314	Not sure why this is getting downvoted.
3	3738	New York, NY	3	48002	Hehe, of course, it's all true!
5	17556	San Diego, CA	5	44921	Please see my post below.
5	17556	San Diego, CA	5	44920	Thank you very much for your reply.

Inner join

- Mapper goes through each record in the two data sets and extracts the foreign key and the rest of the record.
  - It outputs a pair (foreign key, rest of record + X), where X is a binary value indicating which data set the record came from.
- Shuffle and reduce send all records with the same foreign key to one reducer.
- A reducer takes each (key, [record + X]), and stores the record in one of two lists L_1 and L_2, depending on X.
- It then does a nested loop over L_1 and L_2, and outputs a record of L_1 concatenated with a record of L_2.

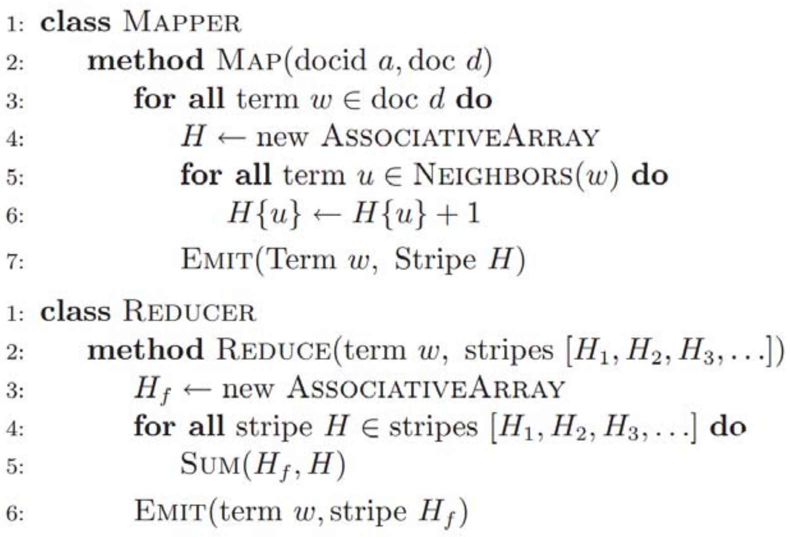
Task: Word co-occurrence

- Given a vocabulary of words, count the number of times a pair of words occur together in some context, e.g. a sentence, paragraph, document, etc.
  - □ Used in text mining, estimating joint events, correlation / mutual information.
- Neighbors(w) gives all words co-occurring with w in some context, e.g. a sliding window.
- Pairs approach outputs all co-occurring pairs as intermediate values, and sums them in reducer.
- Stripes approach stores all words co-occurring with each word w in an associative array (e.g. Java Map), along with number of co-occurrences.
  - □ Shuffle and sort the associative arrays.
  - □ Reducer sums the arrays, e.g. using Java Map's merge(), to get the overall co-occurrences with each word.



```
class Mapper
    method Map(docid $a$, doc $d$)
        for all term $w \in$ doc $d$ do
            for all term $u \in$ Neighbors($w$) do
                Emit(pair $(w, u)$, count 1)

class Reducer
    method Reduce(pair $p$, counts $[c_1, c_2, \ldots]$)
        $s \leftarrow 0$
        for all count $c \in$ counts $[c_1, c_2, \ldots]$ do
            $s \leftarrow s + c$
        Emit(pair $p$, count $s$)
```

Pairs approach

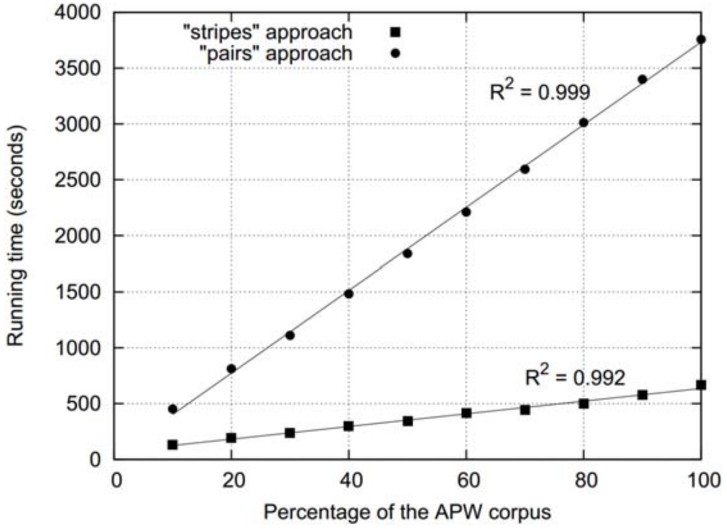


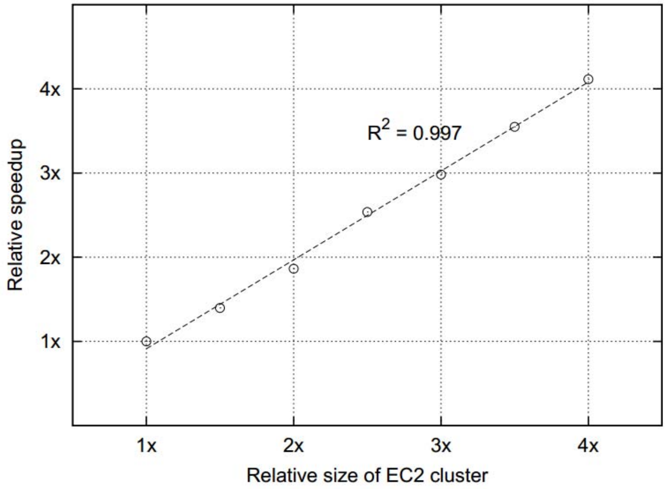
```
class Mapper
    method Map(docid $a$, doc $d$)
        for all term $w \in$ doc $d$ do
            $H \leftarrow$ new AssociativeArray
            for all term $u \in$ Neighbors($w$) do
                $H\{u\} \leftarrow H\{u\} + 1$
            Emit(Term $w$, Stripe $H$)

class Reducer
    method Reduce(term $w$, stripes $[H_1, H_2, H_3, \ldots]$)
        $H_f \leftarrow$ new AssociativeArray
        for all stripe $H \in$ stripes $[H_1, H_2, H_3, \ldots]$ do
            Sum($H_f, H$)
        Emit(term $w$, stripe $H_f$)
```

Stripes approach

Task: Word co-occurrence

- Can use combiners with both pairs and stripes approaches.
  - Combiners have limited effect with pairs, since the same pair of words is unlikely to recur at one mapper.
- Stripes approach holds associative arrays in memory (or requires paging), and may not work for very large files (tera or petabytes).
  - Pairs approach writes out pairs immediately, so not limited by memory.
- On real workload with 2.3M documents totaling 5.7GB, co-occurrence window size 2, and 19 slave nodes with two single-cores processors each:
  - Pairs mappers produced 2.6B intermediate pairs (1.1B after combining) totaling 31.2GB. Reducers output 142M key-value pairs.
  - Stripes mappers produced 653M intermediate pairs (29M after combiners) totaling 48GB. Reducers output 1.7M (key, value / array) pairs.

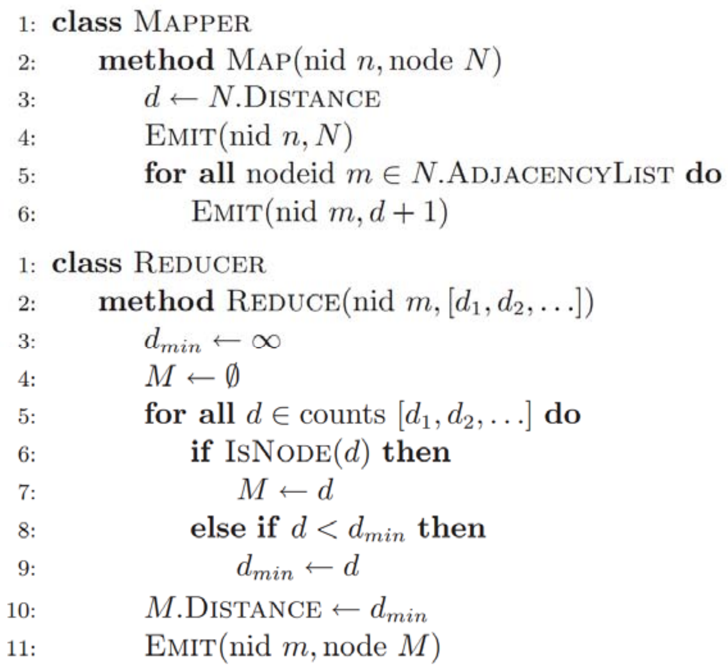




Stripes scaling

Task: Breadth first search

- BFS is an iterative algorithm where in iteration k we find all the nodes at distance k.
- Run multiple iterations of MapReduce.
  - Mapper increments distance to each node's neighbors.
  - Reducer receive (node, estimated distance) pairs grouped by node, and sets node's distance to min estimated distance.
- Not work efficient, because each iteration processes all nodes in graph.
  - Works reasonably well for small-world graphs.
- MapReduce "driver" program responsible for starting new iterations of the job.
  - Terminate when all nodes have been discovered (i.e. their distance is < ∞).
  - Can check this condition using "counters" in API, which count number of user defined events.



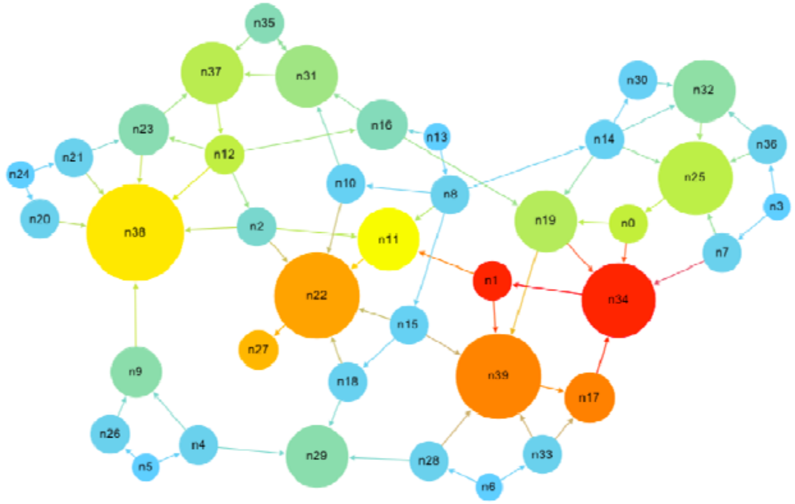
```
class Mapper
    method Map(nid $n$, node $N$)
        $d \leftarrow N$.Distance
        Emit(nid $n, N$)
        for all nodeid $m \in N$.AdjacencyList do
            Emit(nid $m, d+1$)
```

```
class Reducer
    method Reduce(nid $m, [d_1, d_2, \ldots]$)
        $d_{min} \leftarrow \infty$
        $M \leftarrow \emptyset$
        for all $d \in$ counts $[d_1, d_2, \ldots]$ do
            if IsNode($d$) then
                $M \leftarrow d$
            else if $d < d_{min}$ then
                $d_{min} \leftarrow d$
        $M$.Distance $\leftarrow d_{min}$
        Emit(nid $m$, node $M$)
```

Task: Single source shortest paths

- Only need to change BFS code so that node v emits d+w for a neighbor u, where d is u's current estimated distance and w is v's distance to u.
- Termination condition changes to checking when all node distances stop changing.
- This is essentially the Bellman-Ford SSSP algorithm.
- This algorithm very wasteful, since it relaxes all nodes in every iteration whereas Dijkstra's algorithm relaxes only the nearest non-settled node.
  - Ignoring sorting step, this algorithm does $O(VE)$ work instead of $O(E + V \log V\ )$ for Dijkstra, for graph with V nodes and E edges.
- SSSP is a problem not suited for MapReduce.

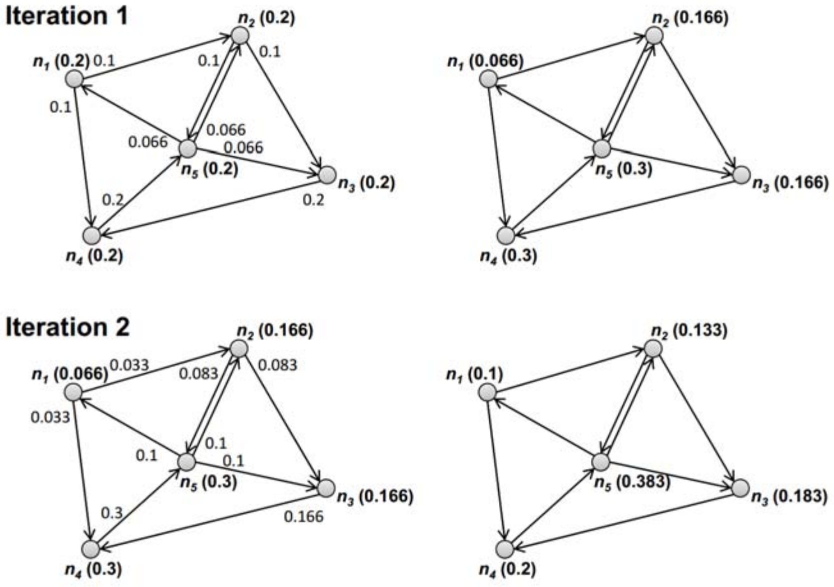
Task: PageRank



- An algorithm to help rank the importance of websites.
  - Idea is that important websites should have lots of incoming links (“recommendations” for site from other sites). Also, links / recommendations from important sites should carry more weight than links from unimportant sites.
  - Used in combination with many other heuristics by search engines.
- PageRank equation is

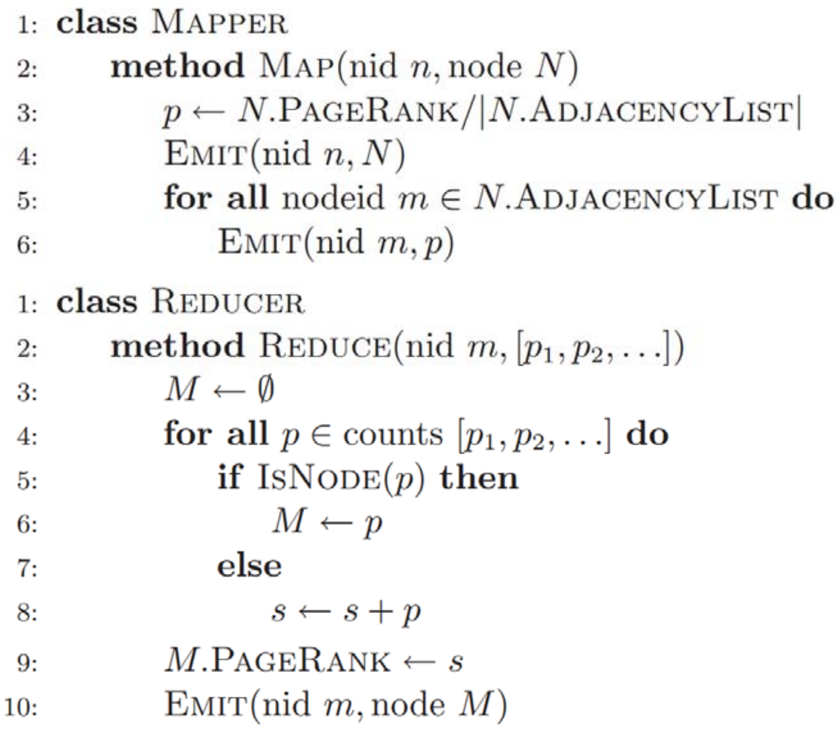
$$P(n) = \alpha\left(\frac{1}{|G|}\right) + (1-\alpha)\sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

  - P(n) is the PageRank of node n.
  - L(n) is the neighbors of n.
  - C(m) is the degree of node m.
  - |G| is the number of nodes in the webgraph G.
  - α is probability a user jumps to a random website instead of one pointed to from the current site.
- Equation says that the probability of a site being visited is the probability a user goes to it from a random site, plus the probability that user comes to it from a site linked to it.
  - Assumes if user is at a site m with C(m) outlinks, he follows one outlink randomly.
- PageRank can be computed by applying equation iteratively, after initializing all nodes with uniform probability.



Task: PageRank

- A simplified version of PageRank without random jumps, and assuming all nodes are reachable.
- Job is run iteratively until convergence, i.e. values of nodes don’t change anymore.
  - Can also run until the ranks of nodes don’t change, which is usually faster.
  - In practice, can stop after ~50-100 iterations.
- Combiners can be useful, but only if there are nodes that are pointed to by many other nodes processed by same mapper.
  - Try to partition graph into clusters with many internal edges.
  - Web graphs often form clusters when spidered.



```
class Mapper
    method Map(nid $n$, node $N$)
        $p \leftarrow N.\text{PageRank}/|N.\text{AdjacencyList}|$
        Emit(nid $n, N$)
        for all nodeid $m \in N.\text{AdjacencyList}$ do
            Emit(nid $m, p$)

class Reducer
    method Reduce(nid $m, [p_1, p_2, \ldots]$)
        $M \leftarrow \emptyset$
        for all $p \in$ counts $[p_1, p_2, \ldots]$ do
            if IsNode($p$) then
                $M \leftarrow p$
            else
                $s \leftarrow s + p$
        $M.\text{PageRank} \leftarrow s$
        Emit(nid $m$, node $M$)
```