Programming Map/Reduce

Hadoop Docs:
https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

Tools:

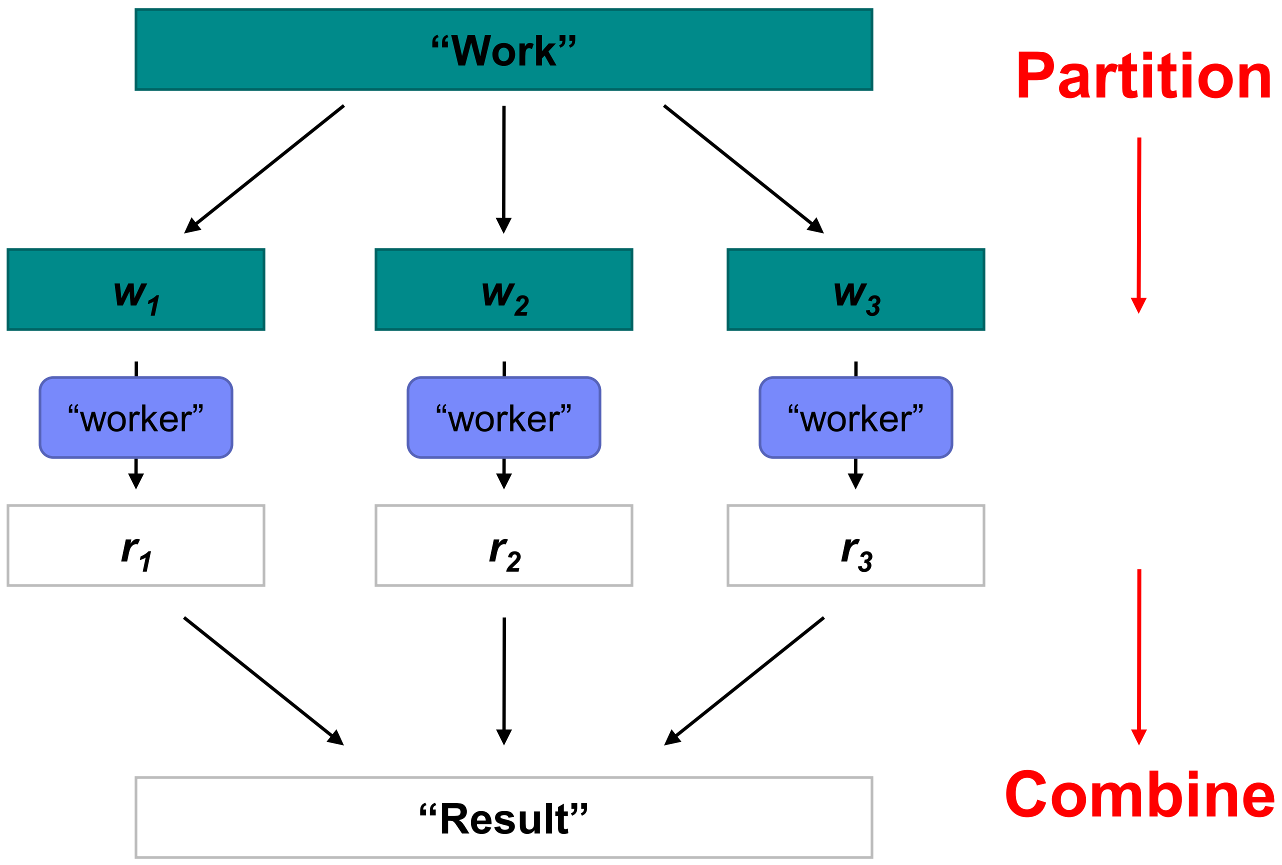
- Eclipse (w/plugin https://github.com/winghc/hadoop2x-eclipse-plugin)
- IntelliJ Idea

Hadoop on Docker (https://www.cloudera.com/documentation/enterprise/5-6-x/topics/quickstart_docker_container.html):

```
run --name hadoop -p 8888:8888 -p 8000:80 --privileged=true --ho
stname=quickstart.cloudera -t -i -d -v D:/:/shared/d -p 8088:8088  cloudera
/quickstart:latest /usr/bin/docker-quickstart
```

Demos:
Python: http://www.michael-noll.com/tutorials/writing-an-hadoop-mapreduce-program-in-python/
Java: WordCount (in class)

Divide and Conquer



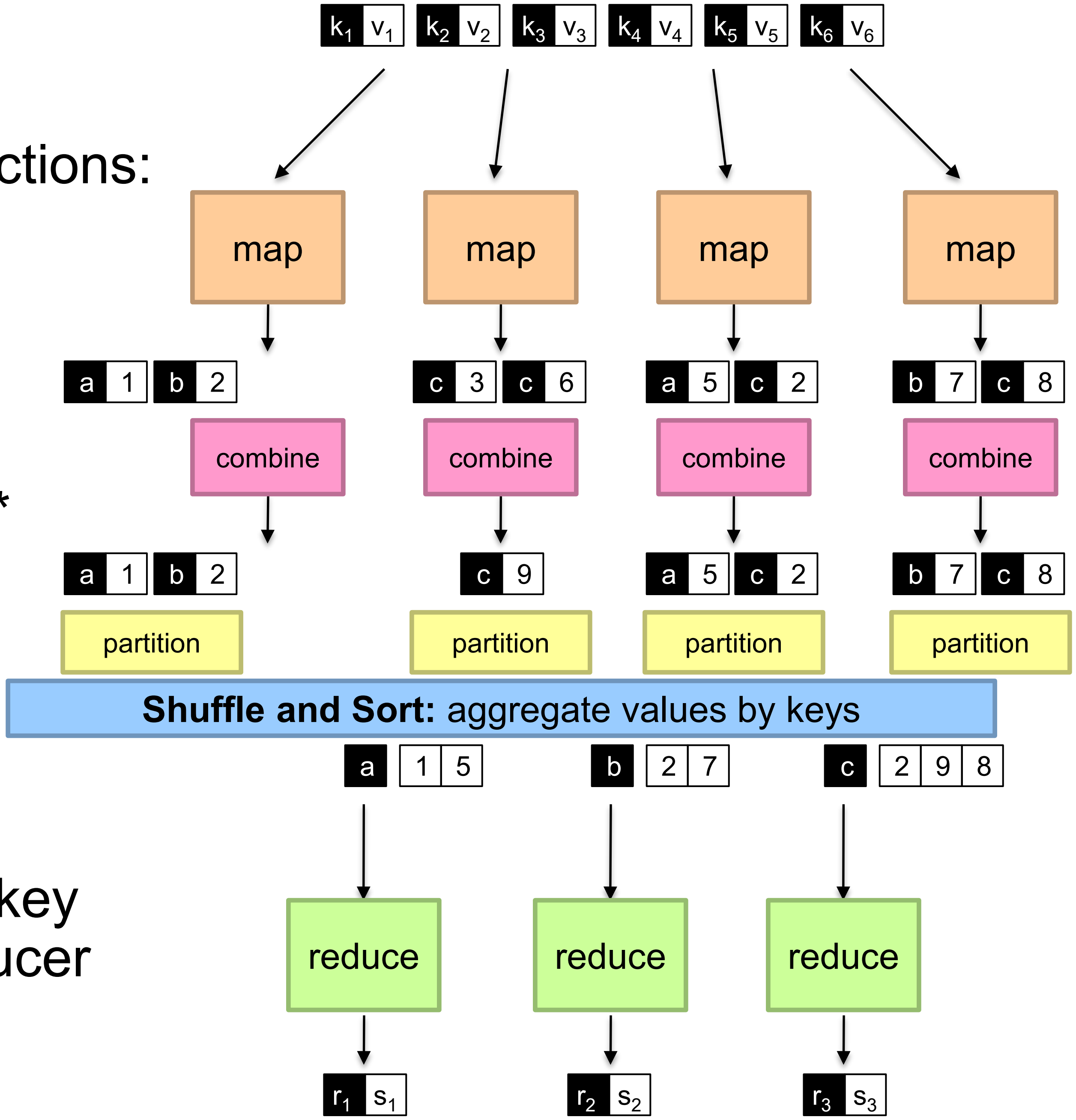
MapReduce:

- Programmers specify functions:

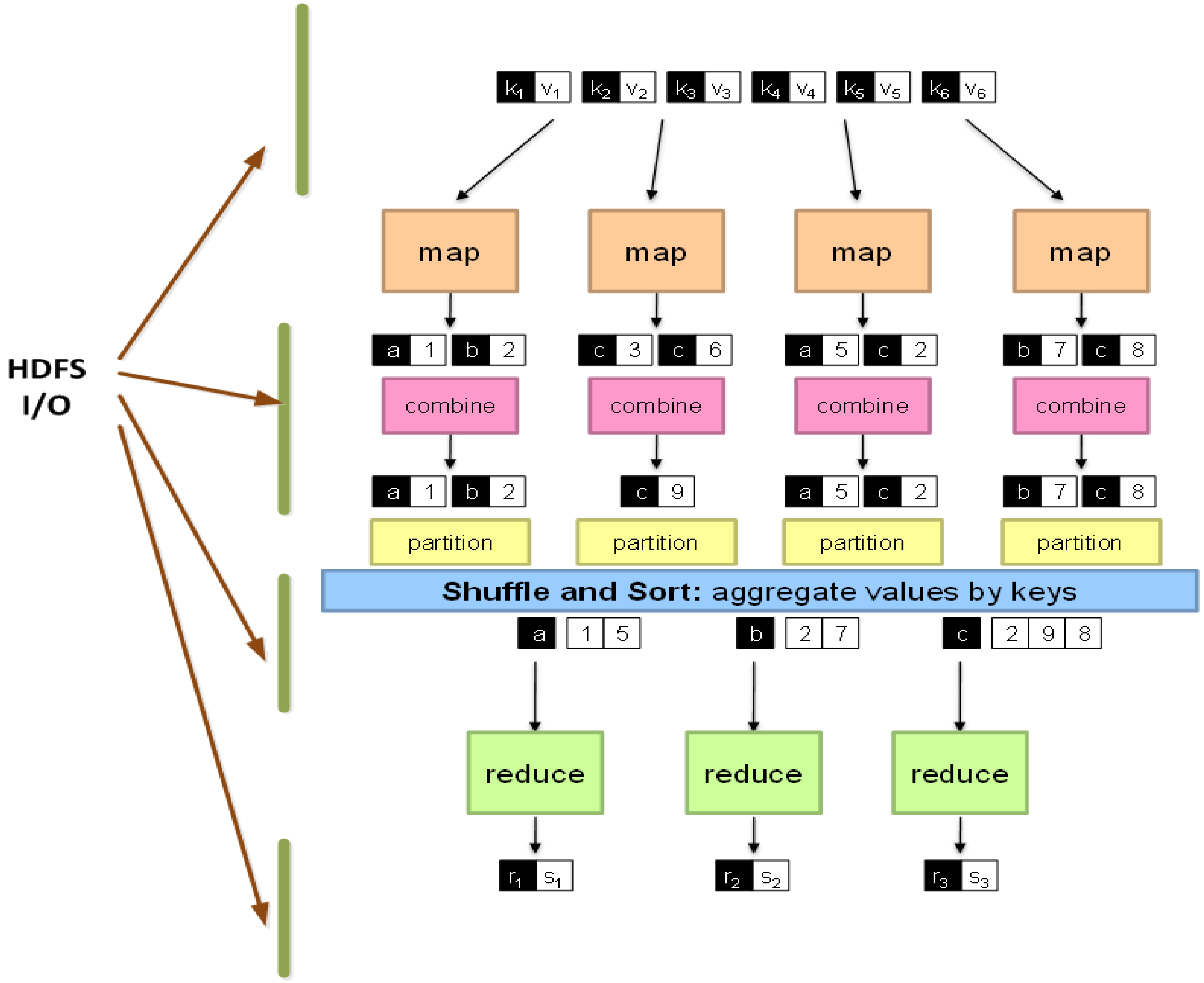
map (k, v) → <k', v'>*

reduce (k', v') → <k', v'>*

All values with the same key
are sent to the same reducer



MapReduce:



HDFS
I/O
k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

MapReduce

Divided in two phases

– Map phase
– Reduce phase

- **Both phases use key-value pairs as input and output**
- **The implementer provides map and reduce functions**
- **MapReduce framework orchestrates splitting, and distributing of Map and Reduce phases**

Most of the pieces can be easily overridden

Source: http://www.coreservlets.com/hadoop-tutorial/

MapReduce

Job – execution of map and reduce functions to accomplish a task
Equal to Java's main

Task – single Mapper or Reducer:
Performs work on a fragment of data

1. Configure the Job: Specify Input, Output, Mapper, Reducer and Combiner
2. Implement Mapper : Input is text – e.g. a line
Tokenize the text and emit first character with a count of 1 - <token, 1>
3. Implement Reducer Sum up counts for each letter
Write out the result to HDFS
4. Run the job

1: Configure Job

- **Job class**

– Encapsulates information about a job

– Controls execution of the job

- **A job is packaged within a jar file**

– Hadoop Framework distributes the jar on your behalf

– Needs to know which jar file to distribute

– The easiest way to specify the jar that your job resides in is by calling job.setJarByClass

– Hadoop will locate the jar file that contains the provided class

Note: Hadoop IO Classes

- **Hadoop uses it's own serialization mechanism for writing data in and out of network, database or files**

– Optimized for network serialization
– A set of basic types is provided
– Easy to implement your own

- **org.apache.hadoop.io package**

– LongWritable for Long
– IntWritable for Integer
– Text for String
– Etc...

1: Configure Job - Input

```
TextInputFormat.addInputPath(job, new Path(args[0]));
job.setInputFormatClass(TextInputFormat.class);
```

- **Can be a file, directory or a file pattern**

– Directory is converted to a list of files as an input

- **Input is specified by implementation of InputFormat - in this case TextInputFormat**

– Responsible for creating splits and a record reader

– Controls input types of key-value pairs, in this case LongWritable and Text

- – File is broken into lines, mapper will receive 1 line at a time

1: Configure Job – Output

```
TextOutputFormat.setOutputPath(job, new Path(args[1]));
job.setOutputFormatClass(TextOutputFormat.class);
```

- **OutputFormat defines specification for outputting data from Map/Reduce job**
- job utilizes an implementation of OutputFormat: TextOutputFormat

– Define output path where reducer should place its output

- If path already exists then the job will fail

– Each reducer task writes to its own file

- By default a job is configured to run with a single reducer

– Writes key-value pair as plain text

1: Configure Job – Output

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

- **Specify the output key and value types for both mapper and reducer functions**
  - – Many times the same type
  - – If types differ then use
  - • setMapOutputKeyClass()
  - • setMapOutputValueClass()

2: Implement Mapper

Class has 4 Java Generics parameters**

– (1) input key (2) input value (3) output key (4) output value

– Input and output utilizes hadoop's IO framework

- org.apache.hadoop.io

- **Your job is to implement map() method**

– Input key and value

– Output key and value

– Logic is up to you

- **map() method injects Context object, use to:**

– Write output

– Create your own counters

- **Java Generics provide a mechanism to abstract Java types. To learn more visit: http://docs.oracle.com/javase/tutorial/extra/generics/index.html

3: Implement Reducer

- **Analogous to Mapper – generic class with four types**
  – (1) input key (2) input value (3) output key (4) output value
  – The output types of map functions must match the input types of reduce function
- **In this case Text and IntWritable**
  – Map/Reduce framework groups key-value pairs produced by mapper by key
- **For each key there is a set of one or more values**
- **Input into a reducer is sorted by key**
- **Known as Shuffle and Sort**
  – Reduce function accepts key->setOfValues and outputs key/value pairs
- **Also utilizes Context object (similar to Mapper)**

3: Reducer as a Combiner

- **Combine data per Mapper task to reduce amount of data transferred to reduce phase**

- **Reducer can very often serve as a combiner**
  – Only works if reducer's output key-value pair types are the same as mapper's output types

- **Combiners are not guaranteed to run**
  – Optimization only
  – Not for critical logic

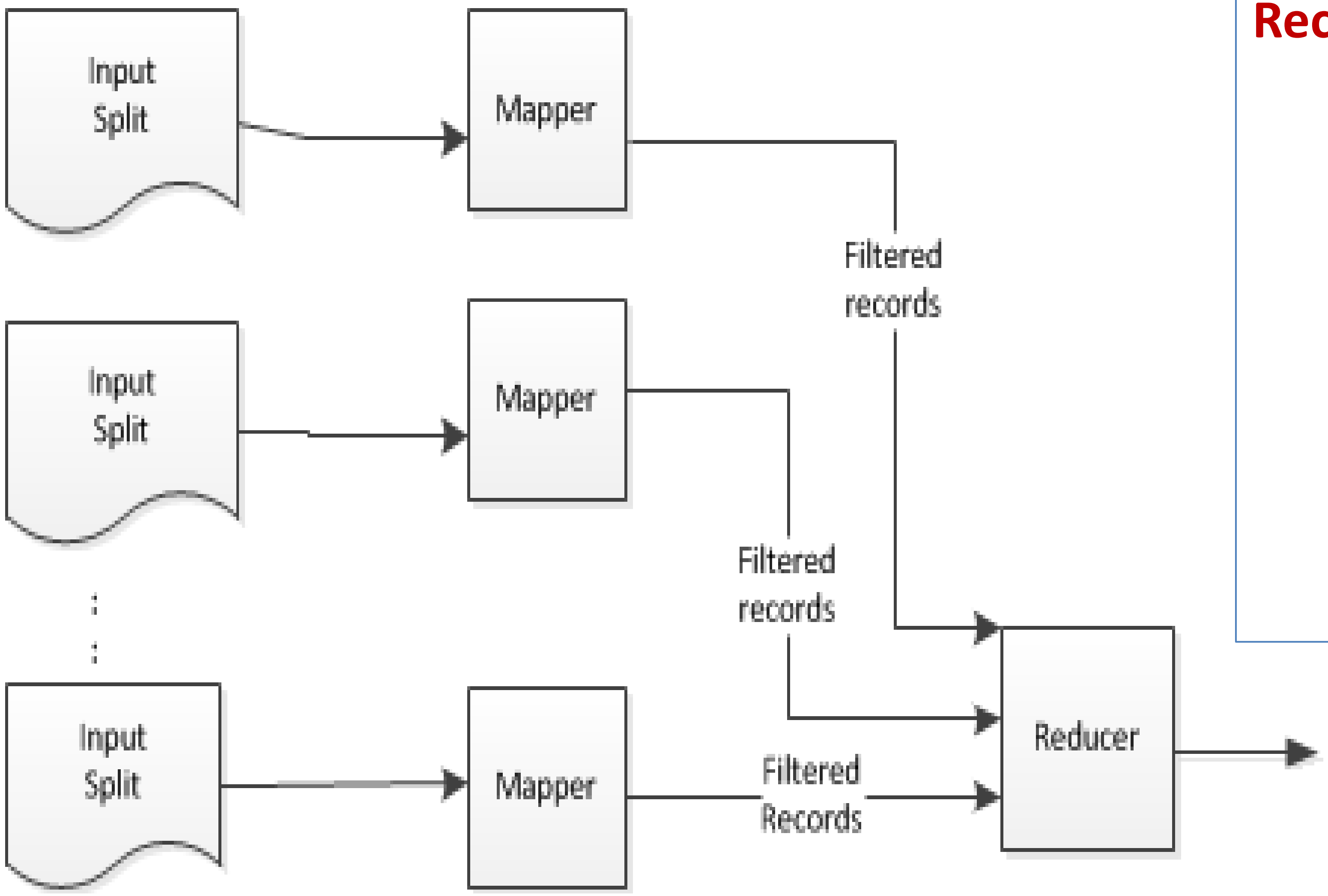
4: Run Count Job

- **HANDS-ON**

Pattern: Filtering

- Most **basic** pattern.
- Use case: Filter out records that are of no interest.
- Why?
  - Recall MapReduce intermediate sorting/shuffling is I/O heavy
  - Want to reduce dataset as much as possible in the map phase.

Filtering



Recall Mapper signature:

```
Map(k,v) {
        // filter
        if (f(k,v))
        {
                :
                emit()
        }
}
```

QUESTIONS?

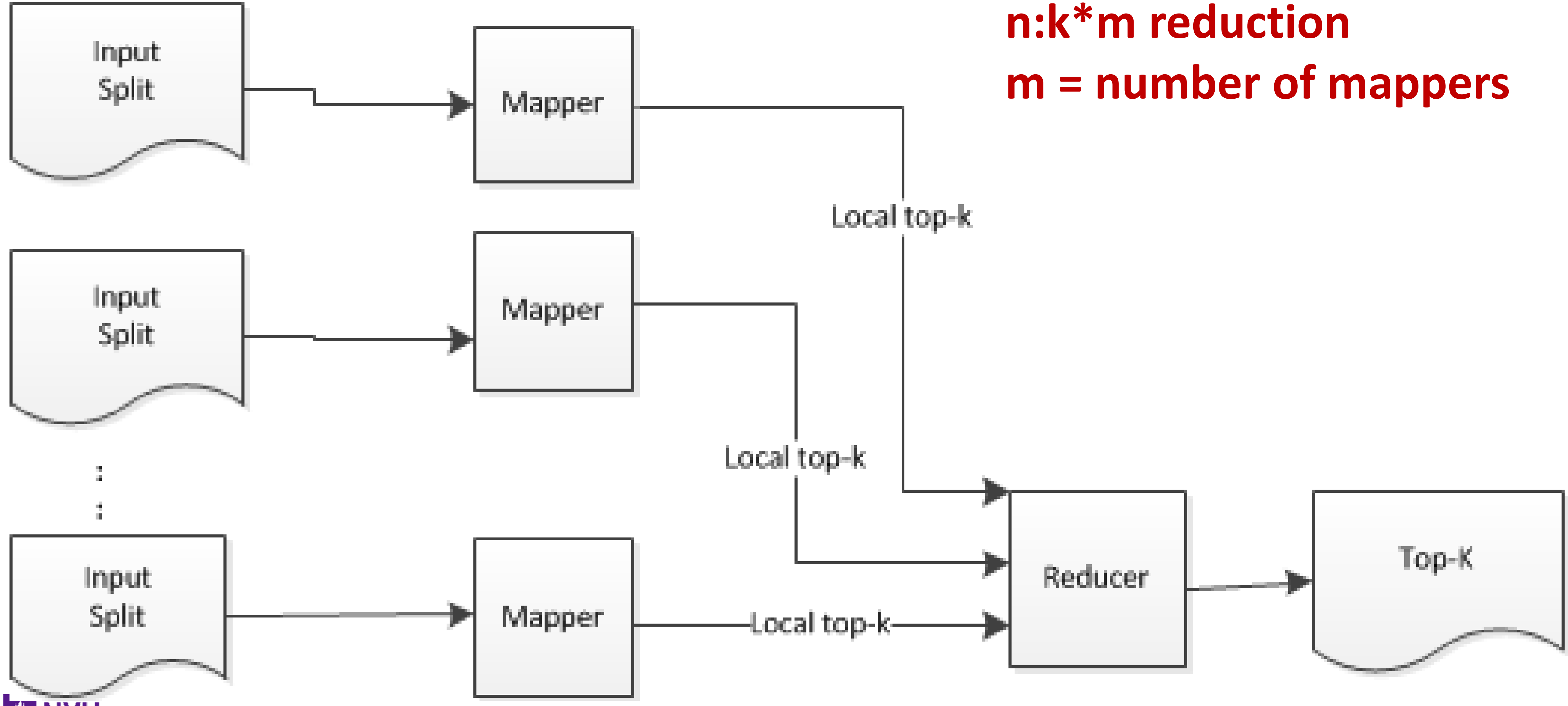
Top 10

Select the top *k* items from a dataset, no matter how large the data.

- Need to evaluate ALL the data and rank according to the top-k criteria. How do we do this in MapReduce?

NYU

Top 10

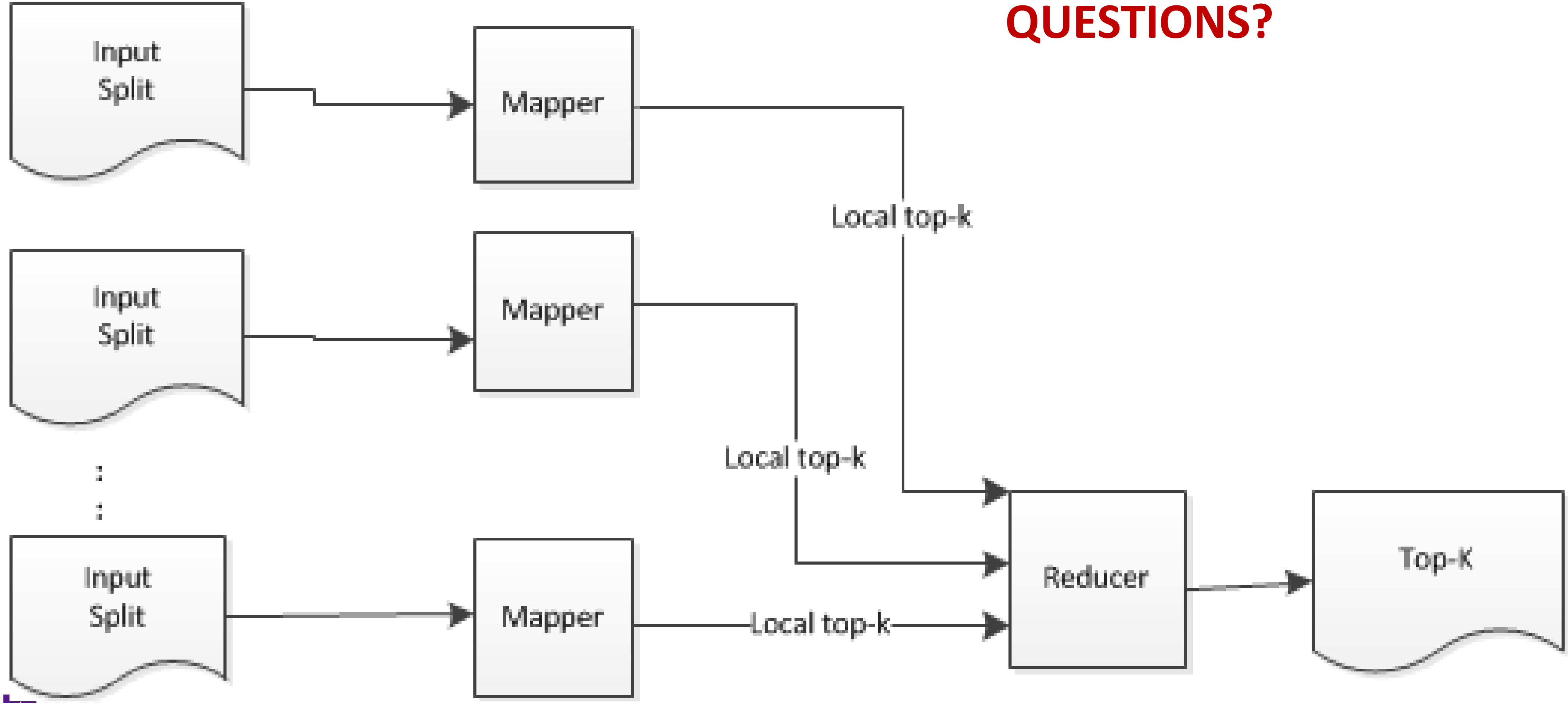


n:k*m reduction

m = number of mappers

NYU

Top 10



QUESTIONS?

Pattern: Binning

Move records into bins/categories, irrespective of the order of records

Recall the review slide on MapReduce:

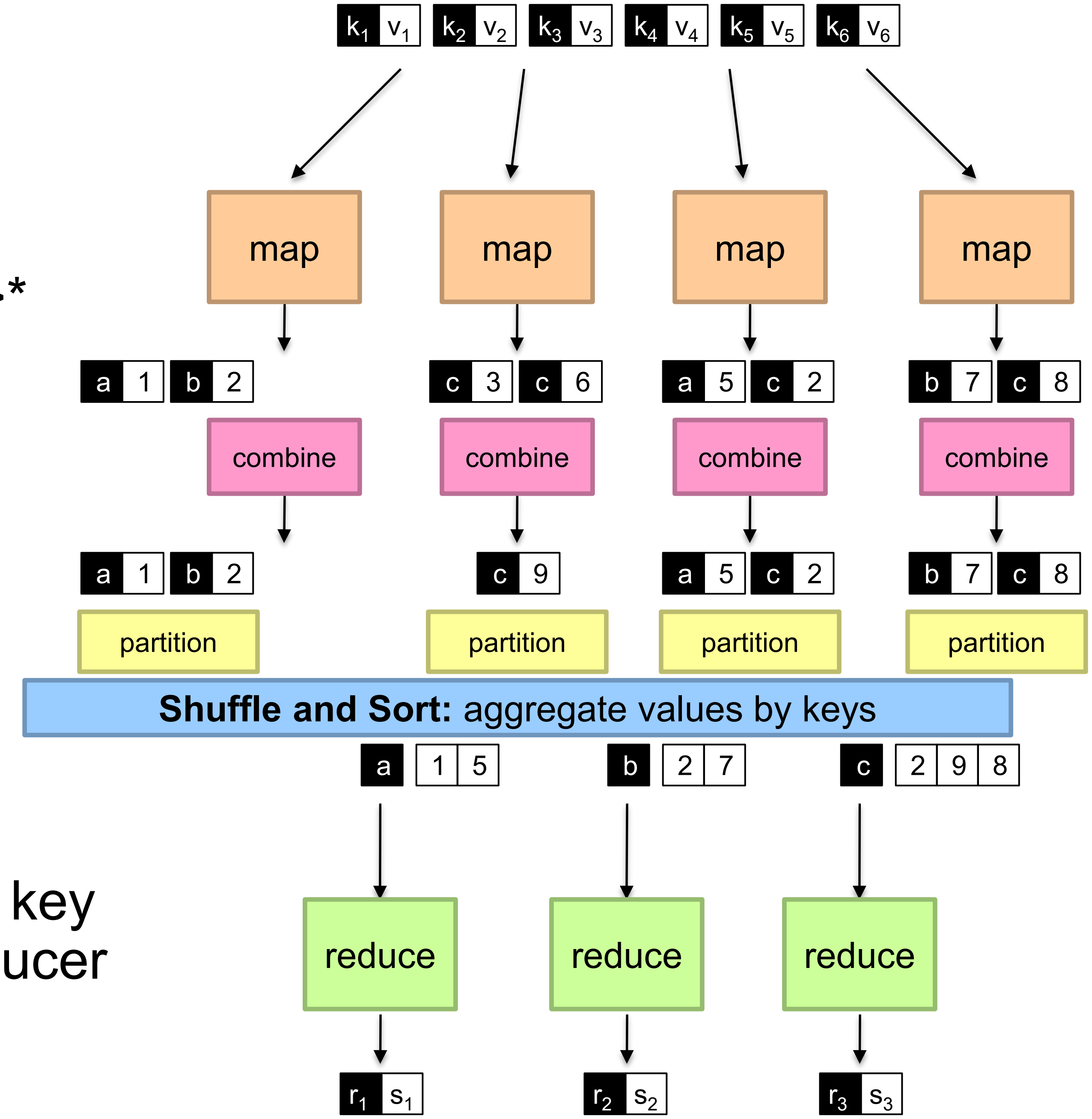
- MapReduce has a partitioner class, which does the exact same job as binning.
- So why use binning?
- **I/O**

NYU

Binning

map (k, v) → <k', v'>*

reduce (k', v') → <k', v'>*



All values with the same key are sent to the same reducer

Pattern: Binning

Move records into bins/categories, irrespective of the order of records

So why use binning?

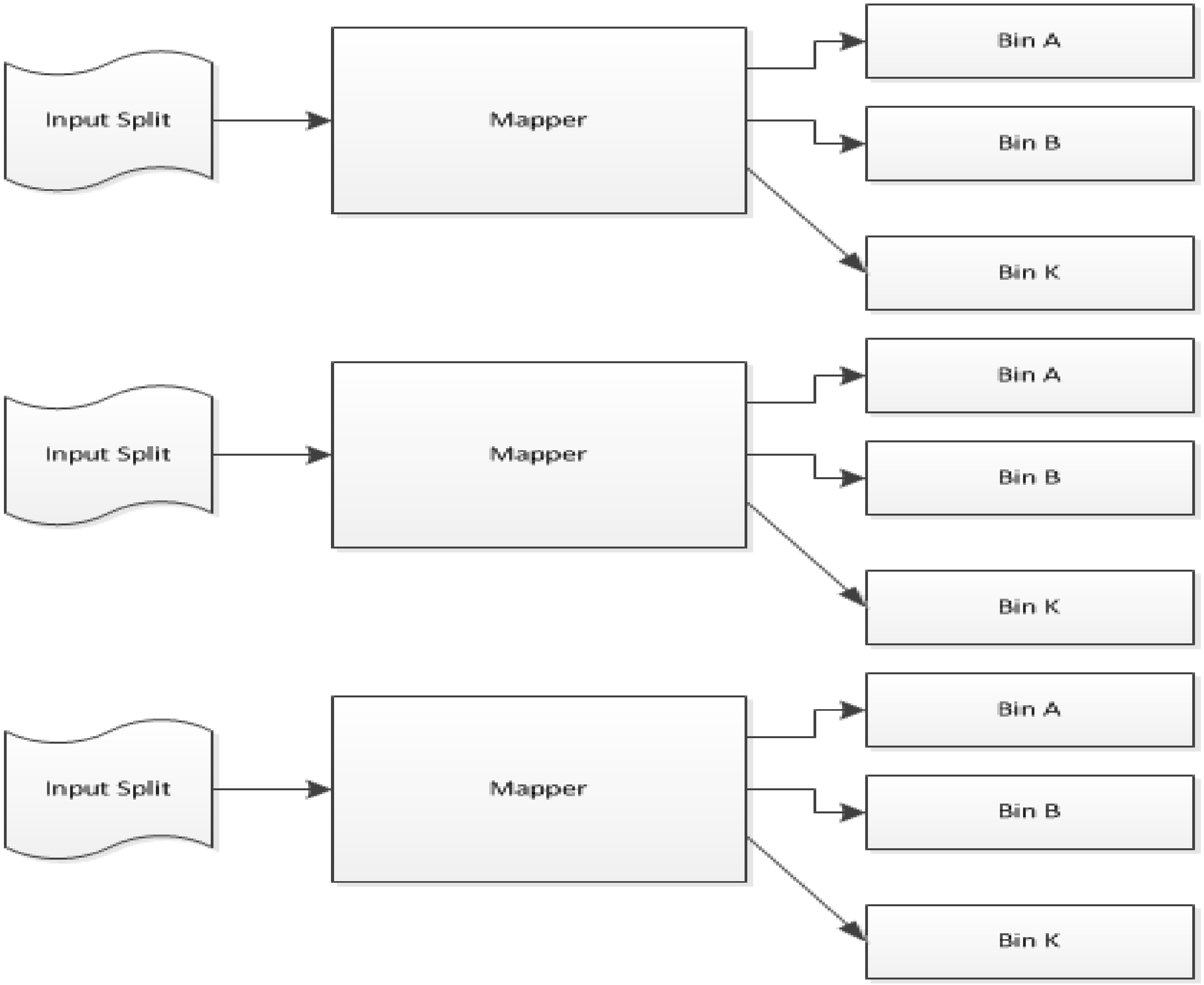
I/O

Binning

- intermediate files out of the Mapper

- No need for a reducer

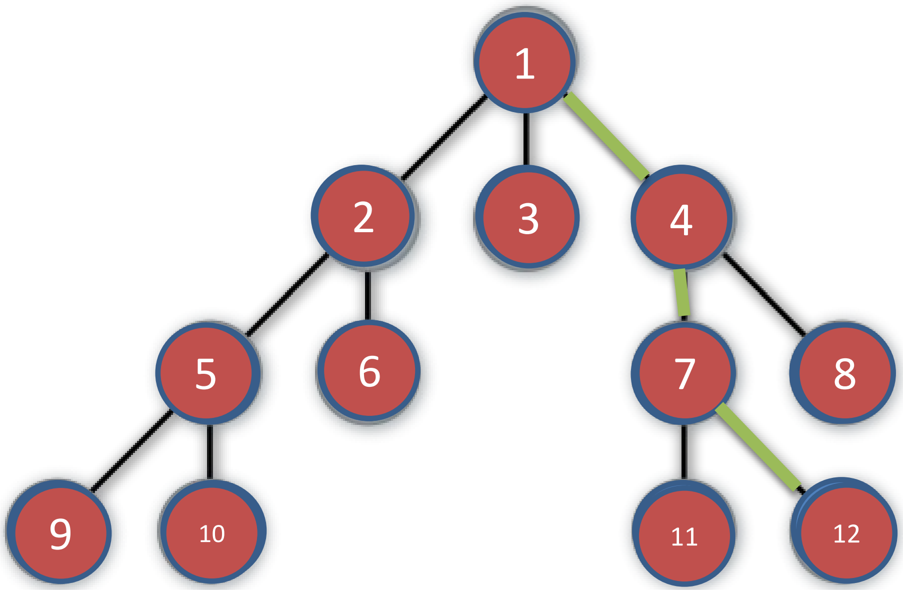
- All output files can
- Just be concatenated



QUESTIONS?

NYU

Pattern: BFS – Breadth First Search



BFS = general technique for traversing a graph.
BFS on a graph with n vertices and m edges: O(n + m)

Algorithm:

– Input: Simple Connected directed graph with ‘n’ vertices and the node to be searched.

– Output: if node is found “Yes” is printed and the corresponding path is displayed
else “No” is printed.

Why do you care?
Shortest Path
• BFS Guarantees to find the shortest path to the destined node if it exists in the graph.

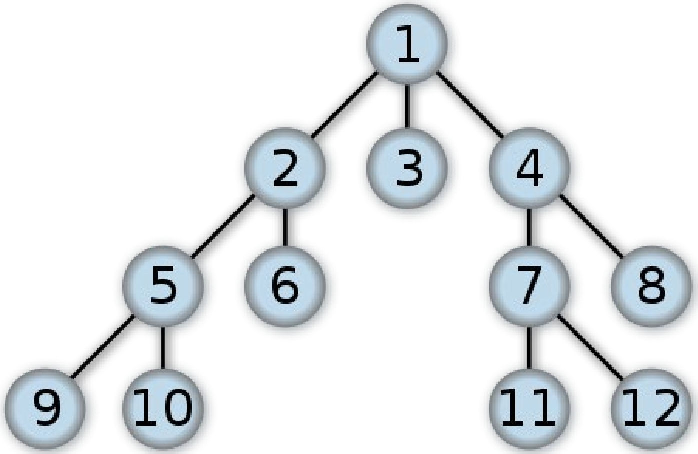
BFS – Breadth First Search

Graph is represented as adjacency list.

- **Key**: Node ID
- **Value**: EDGES|DISTANCE_FROM_SOURCE|COLOR|

  Where EDGES is a comma delimited list of the ids of the nodes that are connected to this node. in the beginning, we do not know the distance and will use Integer. MAX_VALUE for marking "unknown". Color tells us whether or not we've seen the node before, so this starts off as white.

Key	Value
1	2,3,4|0|GRAY|
2	5,6|Integer.MAX_VALUE|WHITE|
3	NULL|Integer.MAX_VALUE|WHITE|
4	7,8|Integer.MAX_VALUE|WHITE|
5	8|Integer.MAX_VALUE|WHITE|
:	



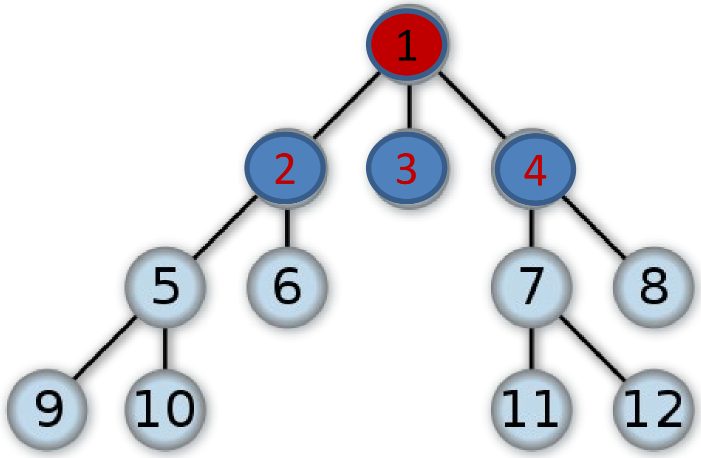
BFS – Breadth First Search

Map:

- For each gray node, the mappers emit a new gray node, with distance = distance + 1. they also then emit the input gray node, but colored red. (we're done with it.) Mappers also emit all non-gray nodes, with no change. so, the output of the first map iteration would be:
-

Key	Value
1	2,3,4|0|RED|
2	NULL|1|BLUE|
3	NULL|1|BLUE|
4	NULL|1|BLUE|
2	1,5,6|Integer.MAX_VALUE|WHITE|
3	NULL|Integer.MAX_VALUE|WHITE|
4	7,8|Integer.MAX_VALUE|WHITE
5	8|Integer.MAX_VALUE|WHITE|

:



BFS – Breadth First Search

Reduce:

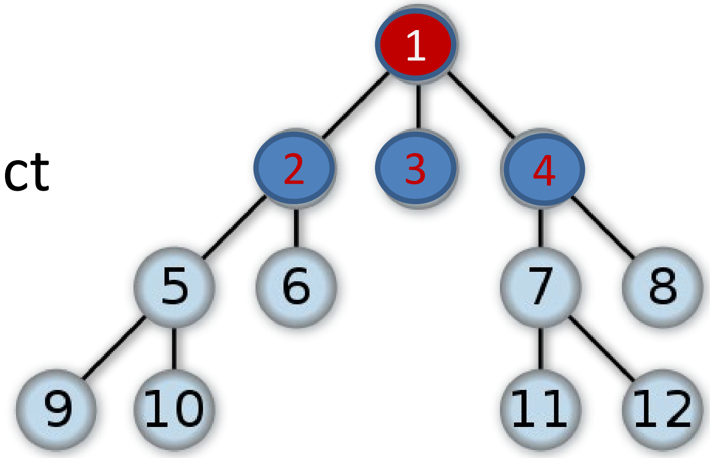
Reducers, receives all data for a given key:
in this case it means that they receive the data for all "copies" of each node.

e.g, the reducer that receives the data for key = 2 gets:

2 NULL|1|BLUE|
2 1,5,6|Integer.MAX_VALUE|WHITE|

The reducers job is to take all this data and construct a new node. Output of Reducer:

2 1,5,6|1|BLUE

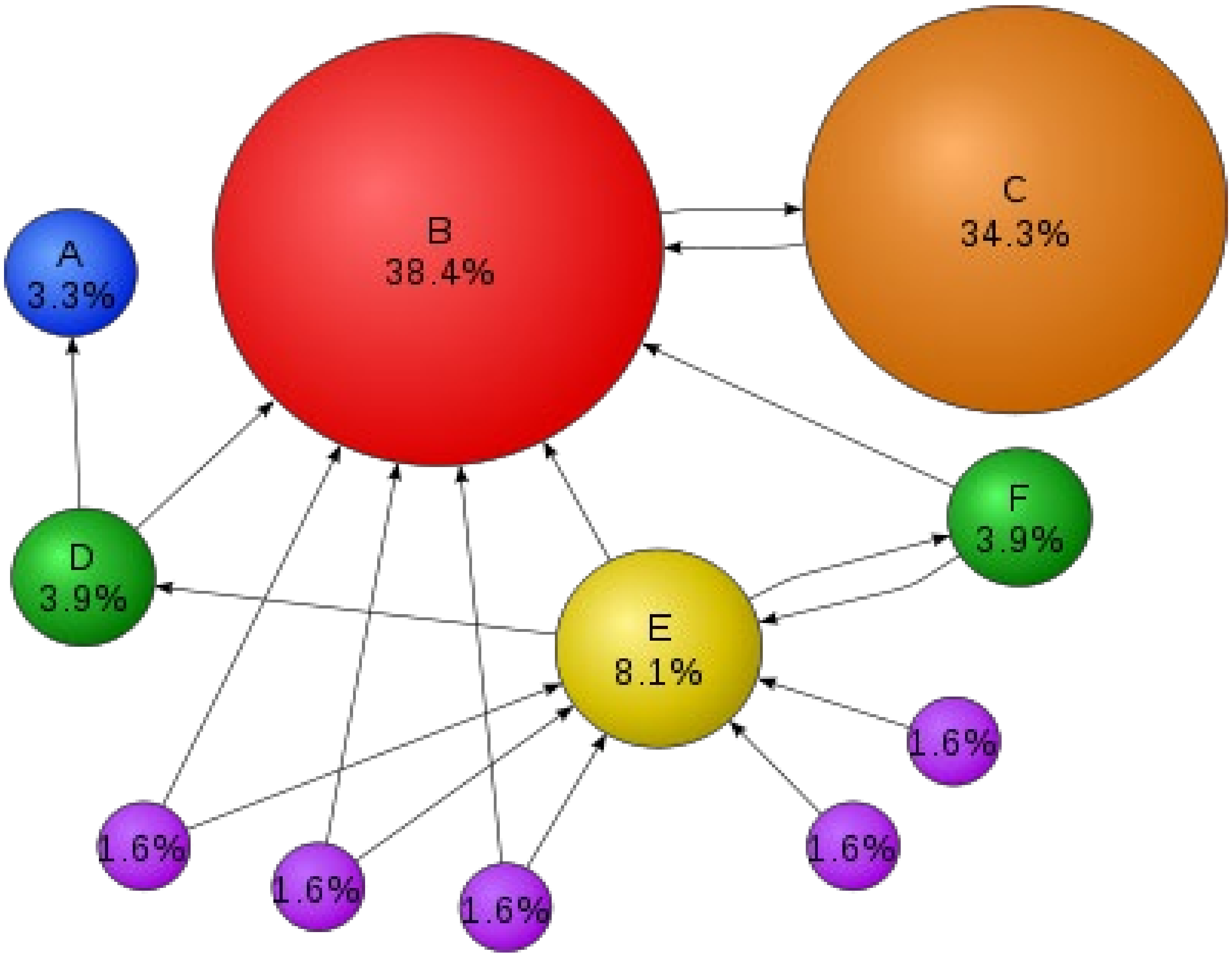


Then?

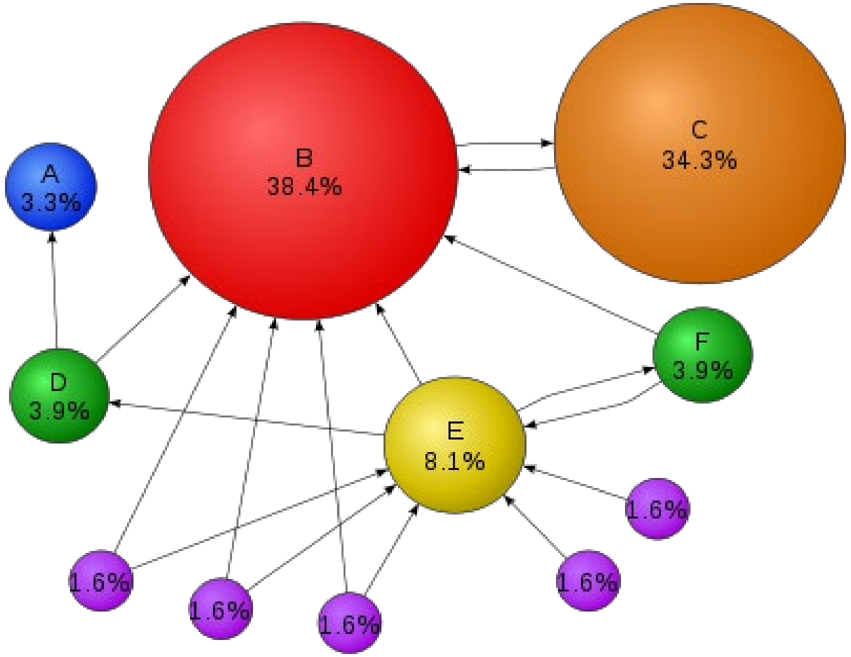
Repeat MAP/REDUCE sequence until node is found, or n times, where n=# nodes

PageRank

Graph Algorithm



PageRank



- Jimmy Lin and Michael Schatz. [Design Patterns for Efficient Graph Algorithms in MapReduce.] *Proceedings of the 2010 Workshop on Mining and Learning with Graphs Workshop (MLG-2010)*, July 2010, Washington, D.C.

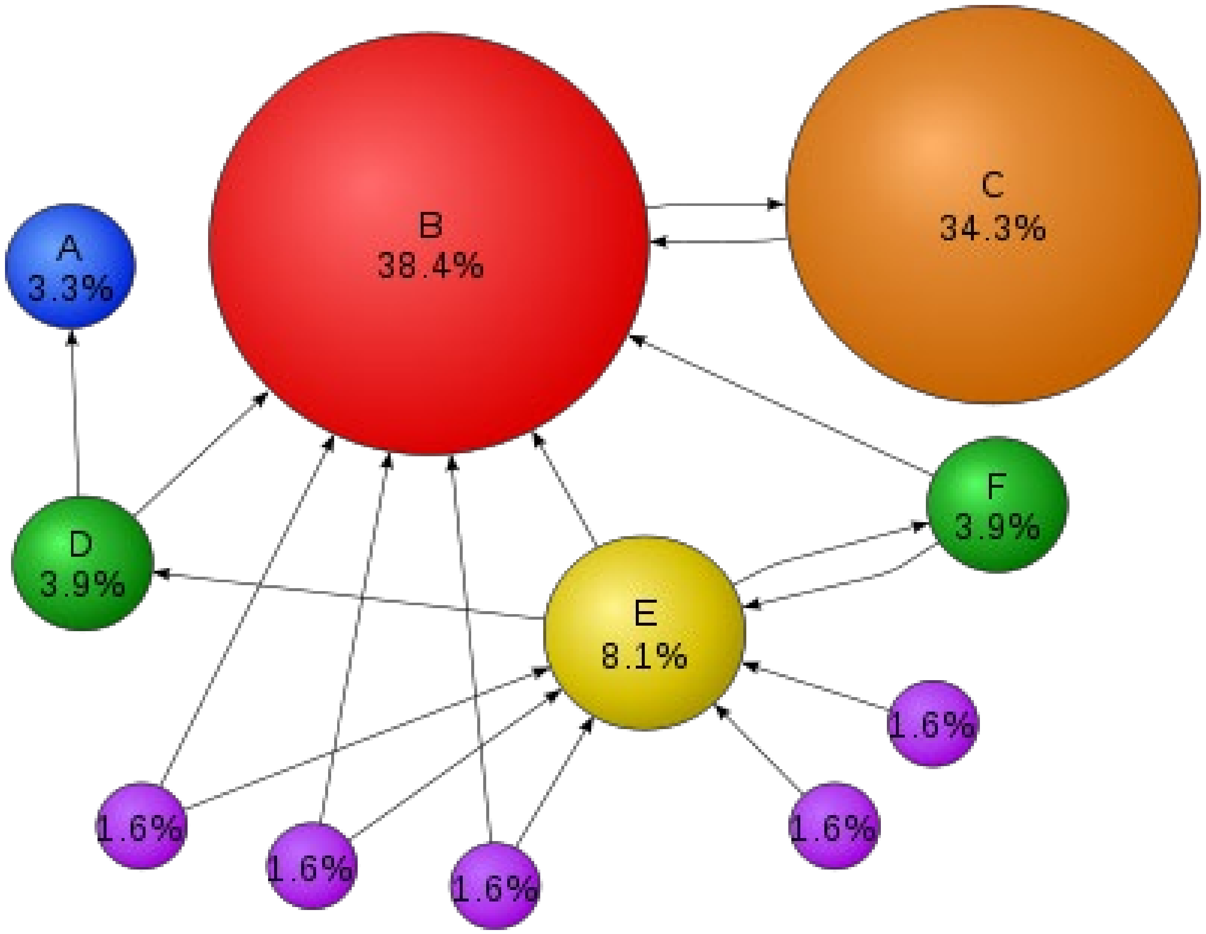
PageRank

The basic algorithm:

At each iteration; evenly distribute a node's probability mass to its neighbors ; until convergence

The hard part:

How do you do this in MR?



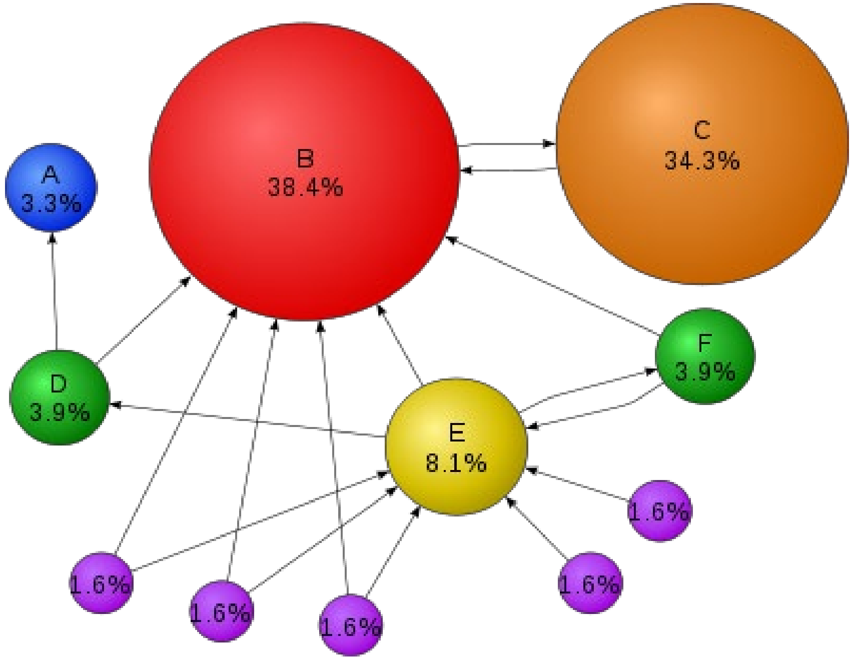
PageRank

- **The idea**
- 1. Represent the graph as an adjacency list
- 2. Partition the graph using Hash functions (binning)
- 3. At each mapper: **emit** the node and it’s entire adjacency list (need to preserve the graph structure)
- 4. At each reducer, reconstruct the node and it’s neighbors, compute incoming probability mass
- 5. Repeat until converge

PageRank

```
class Mapper
    method Map(id n, vertex N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(id n, vertex N)
        for all nodeid m ∈ N.AdjacencyList do
            Emit(id m, value p)
```

```
class Reducer
    method Reduce(id m, [p1, p2, ...])
        M ← ∅
        for all p ∈ [p1, p2, ...] do
            if IsVertex(p) then
                M ← p
            else
                s ← s + p
        M.PageRank ← s
        Emit(id m, vertex M)
```



- In the map phase we evenly divide up each vertex's PageRank mass and pass each piece along outgoing edges to neighbors.
- In the reduce phase PageRank contributions are summed up at each destination vertex.
- Each MapReduce job corresponds to one iteration of the algorithm.