Data Managment with MapReduce

Talel.Abdessalem@telecom-paristech.fr

Based on slides from Jimmy Lin, Jiaheng Lu, the book of Anand Rajaraman and Jeff Ullman titled "Mining of Massive Datasets“, and the book of Serge Abiteboul et al. titled “Web Data Management”.

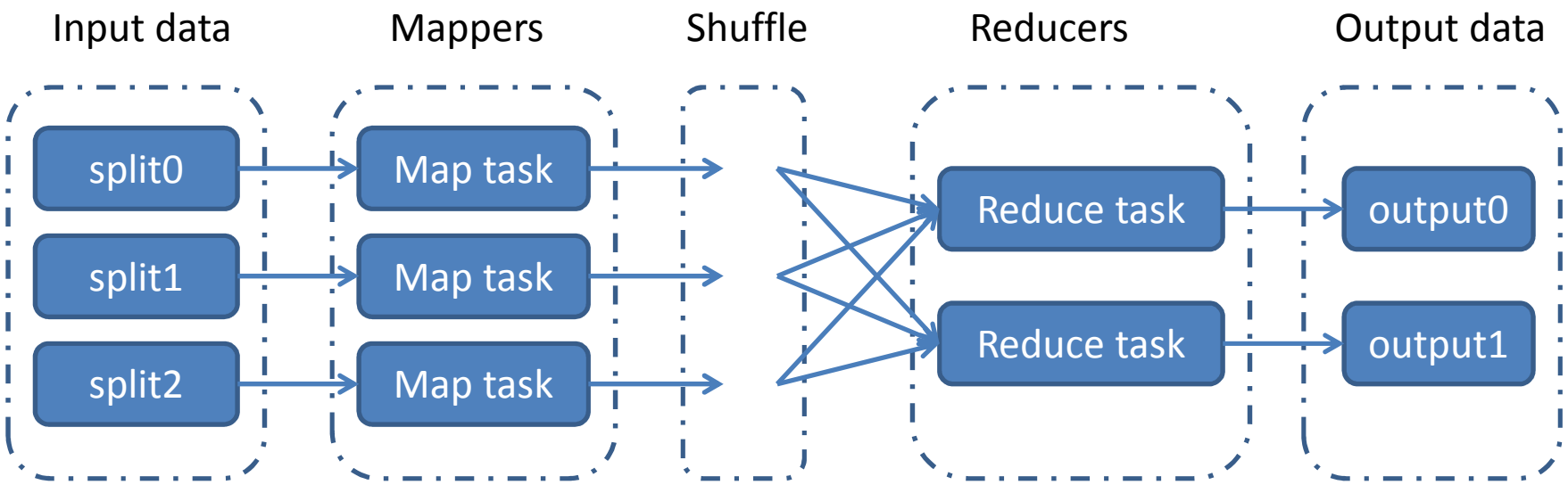
Outline

- Review of MapReduce
- MapReduce for processing relational data
- Graph algorithms in MapReduce

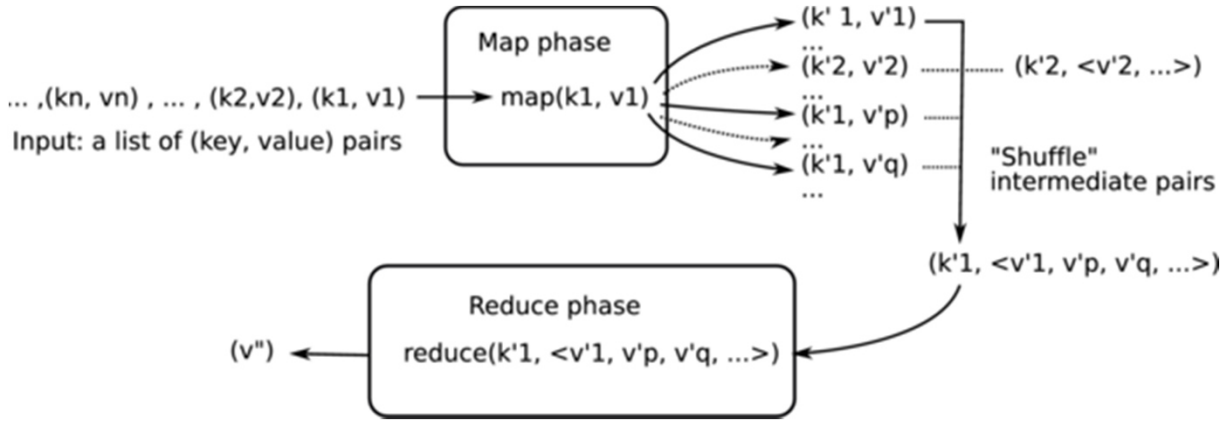
Review of MapReduce

MapReduce

- A parallel programming framework
- Divide and merge



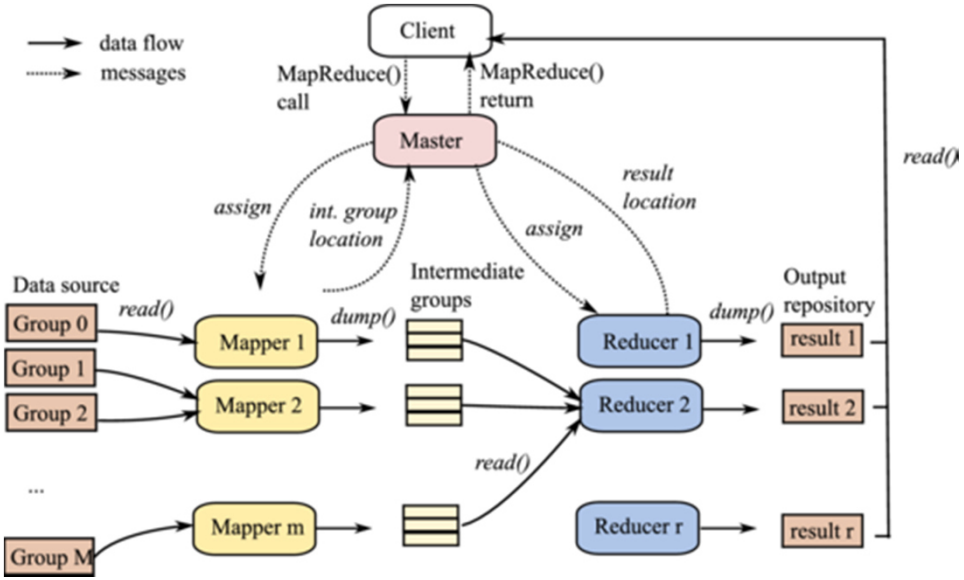
Programming model



- MAP:
  - Input a list of pairs (k,v), where k belongs to a key space K_1 and v to a value space V_1.
  - Processes *independently* each pair and produces (for each pair), another *list* of pairs(k',v') $\in K_2 \times V_2$.
  - Note that the key space and value space of the intermediate pairs, K_2 and V_2, may be different from those of the input pairs, K_1 and V_1.
  - May produce several pairs $(k'_1,v'_1),\cdots,(k'_1,v'_p),\cdots$, for the same key value component.
  - All the values for the same key as grouped in structures of type (K_2, *list*(V_2)), for instance $(k'_1,\langle v'_1,\cdots,v'_p,\cdots\rangle)$.
- REDUCE:
  - Operates on the grouped instances of intermediate pairs, each instance is processed *independently* from the others.
  - The user-defined *reduce()* function outputs a result, usually a single value.

Source: http://webdam.inria.fr/Jorge/html/wdmch17.html#x23-34600016

MapReduce jobs



- The input data set is assumed to be partitioned over a set of *M* nodes in the cluster.
- The *map()* function is distributed to these nodes and applies to the local subset of the data (data locality principle).
- These bags (local subsets of the data) constitute the units of the distributed computation of the MAP: each MAP task works on one and only one bag.
- The input of a MAPREDUCE job can be a variety of data sources, ranging from a relational database to a file system, with all possible semi-structured representations in between.
- In the case of a relational system, each node hosts a DBMS server and a bag consists of one of the blocks in a partition of a relational table. In the case of a file system, a bag is a set of files stored on the node.

Source: http://webdam.inria.fr/Jorge/html/wdmch17.html#x23-34600016

Sample

Consider a program *CountWords()* that counts the number of word occurrences in a collection of documents: for each word *w*, we want to count how many times *w* occurs in the entire collection.

MAP:

- takes as input a pair (*i*,*doc*), where *i* is a document id and *doc* its content, and applies *mapCW* to each pair in the list.
- Poduces a list of intermediate pairs (*t*,*c*), where *t* is a term occurring in the input document and *c* the number of occurrences of *t* in the document.

```
mapCW(String key, String value):
 // key: document name
 // value: document contents
 // Loop on the terms in value
 for each term t in value:
  let result be the number of occurrences of t in value
  // Send the result
  return (t,result);
```

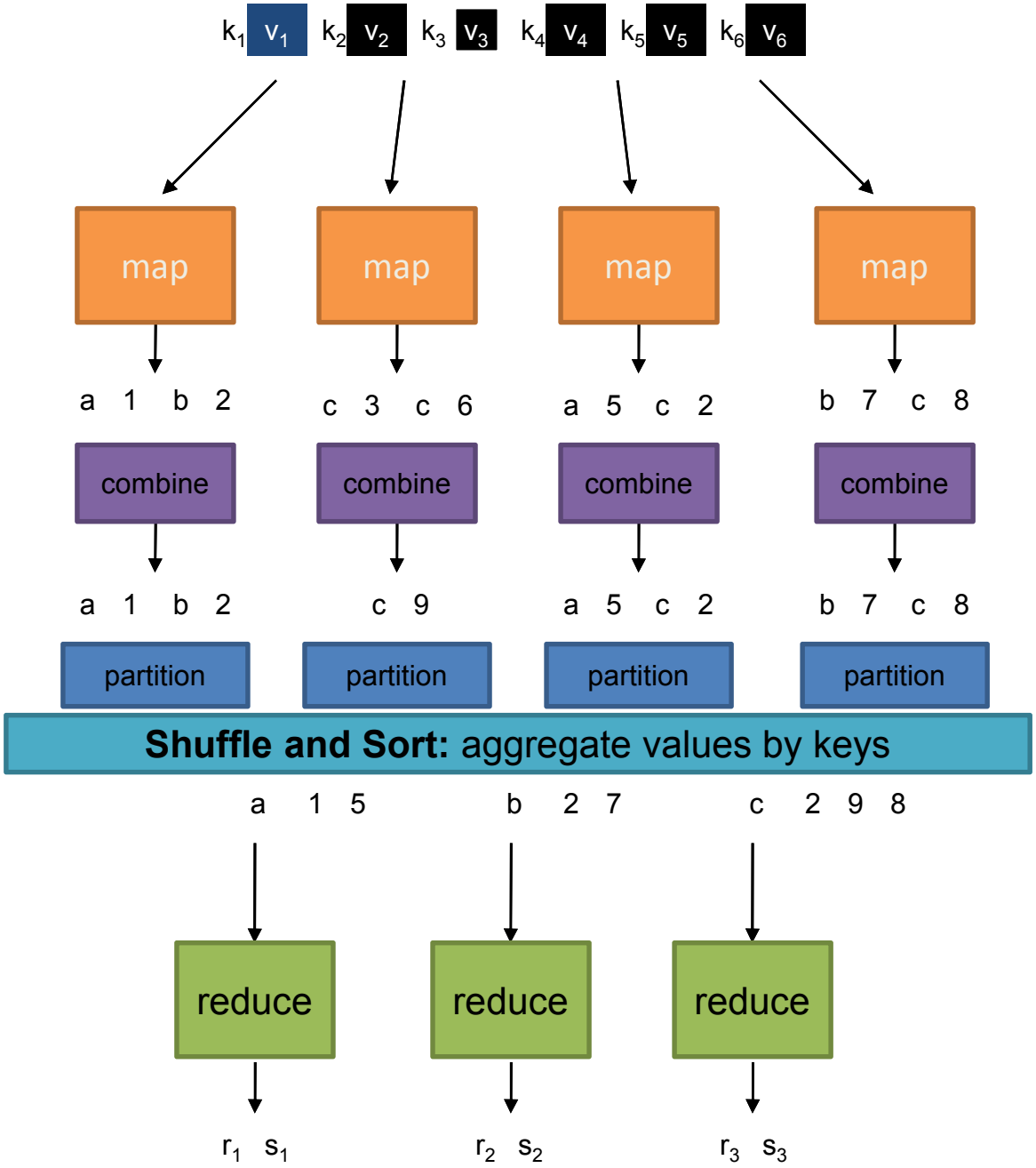
REDUCE:

- The REDUCE phase uses a user-defined function *reduceCW* that takes as input a pair (*t*,*list*(*c*)), *t* being a term and *list*(*c*) a list of all the partial counts produced during the MAP phase. The function simply sums the counts.

```
reduceCW(String key, Iterator values):
 // key: a term
 // values: a list of counts
 int result = 0;
 // Loop on the values list; accumulate in result
 for each v in values:
  result += v;
 // Send the result
 return result;
```

Source: http://webdam.inria.fr/Jorge/html/wdmch17.html#x23-34600016

More details: http://hadoop.apache.org/common/docs/current/mapred_tutorial.html#Overview


k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
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
r1 s1
r2 s2
r3 s3

MapReduce: Recap

- Programmers must specify:
  map (k, v) → list(<k', v'>)
  reduce (k', list(v')) → <v''>
  – All values with the same key are reduced together

- Optionally, also:
  partition (k', number of partitions) → partition for k'
  – Often a simple hash of the key, e.g., hash(k') mod n
  – Divides up key space for parallel reduce operations
  combine (k', v') → <k', v'>*
  – Mini-reducers that run in memory after the map phase
  – Used as an optimization to reduce network traffic

- The execution framework handles everything else…

“Everything Else”

- The execution framework handles everything else…
  - Scheduling: assigns workers to map and reduce tasks
  - “Data distribution”: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts

- Limited control over data and execution flow
  - All algorithms must be expressed in m, r, c, p

- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

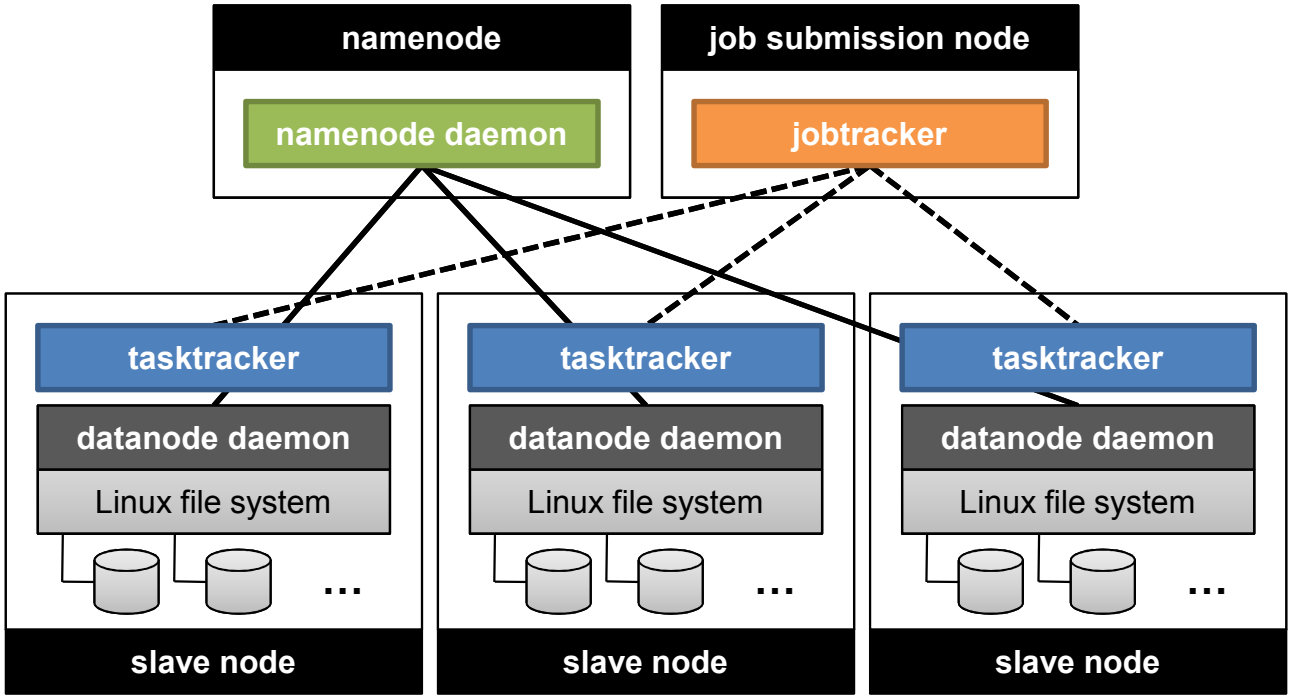
Scalable Hadoop Algorithms

- Avoid object creation
  - Inherently costly operation
  - Garbage collection
- Avoid buffering
  - Limited heap size
  - Works for small datasets, but won't scale!

Basic Cluster Components

- Master node:
  - Namenode (NN)
  - Jobtracker (JT)
- Slave machines:
  - Tasktracker (TT)
  - Datanode (DN)

Putting everything together…



Anatomy of a Job

- MapReduce program in Hadoop = Hadoop job
  - Jobs are divided into map and reduce tasks
  - An instance of running a task is called a task attempt
  - Multiple jobs can be composed into a workflow

- Job submission process
  - Client (i.e., driver program) creates a job, configures it, and submits it to job tracker
  - JobClient computes input splits (on client end)
  - Job data (jar, configuration XML) are sent to JobTracker
  - JobTracker puts job data in shared location, enqueues tasks
  - TaskTrackers poll for tasks
  - Off to the races…

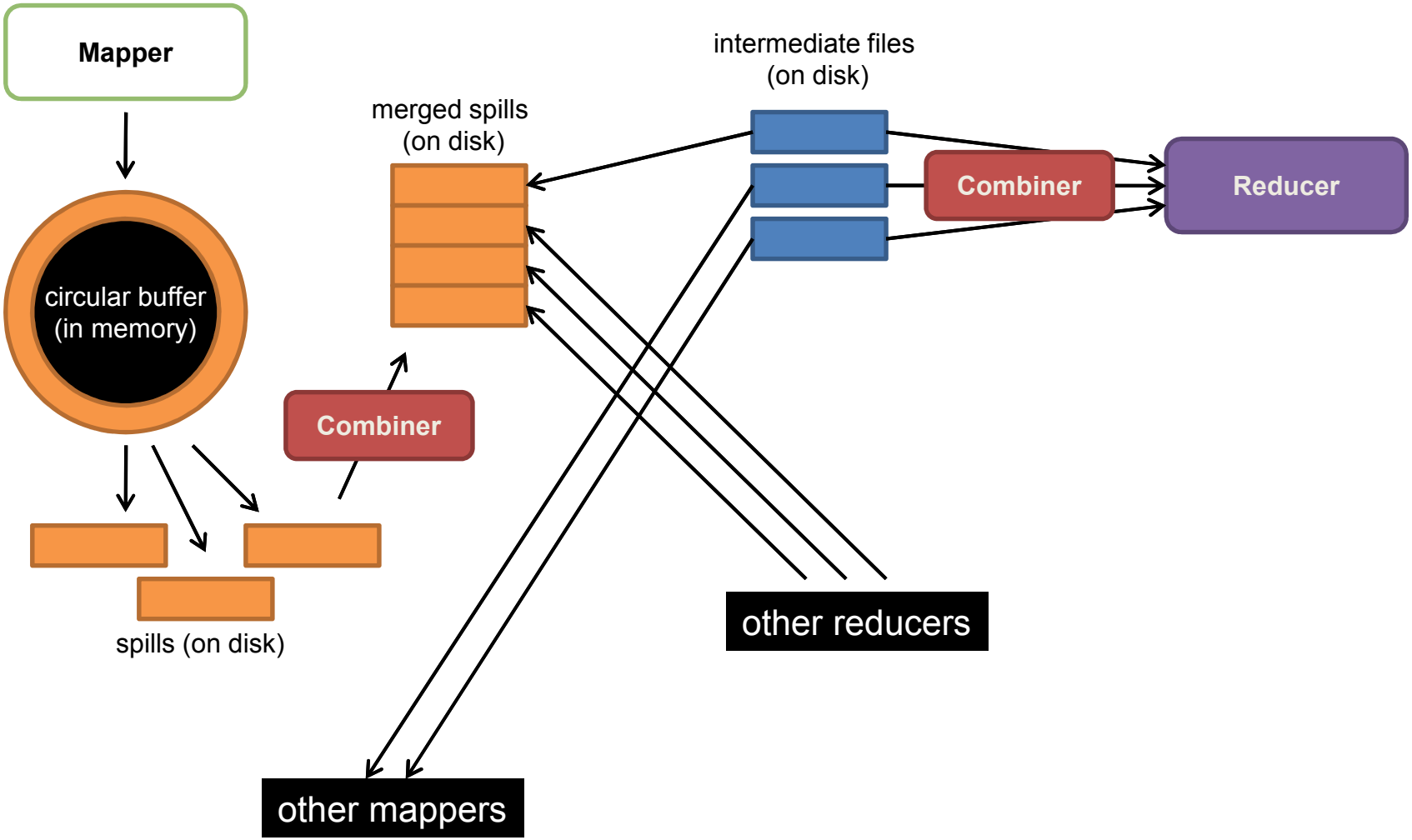
Input and Output

- InputFormat:
  - TextInputFormat
  - KeyValueTextInputFormat
  - SequenceFileInputFormat
  - …
- OutputFormat:
  - TextOutputFormat
  - SequenceFileOutputFormat
  - …

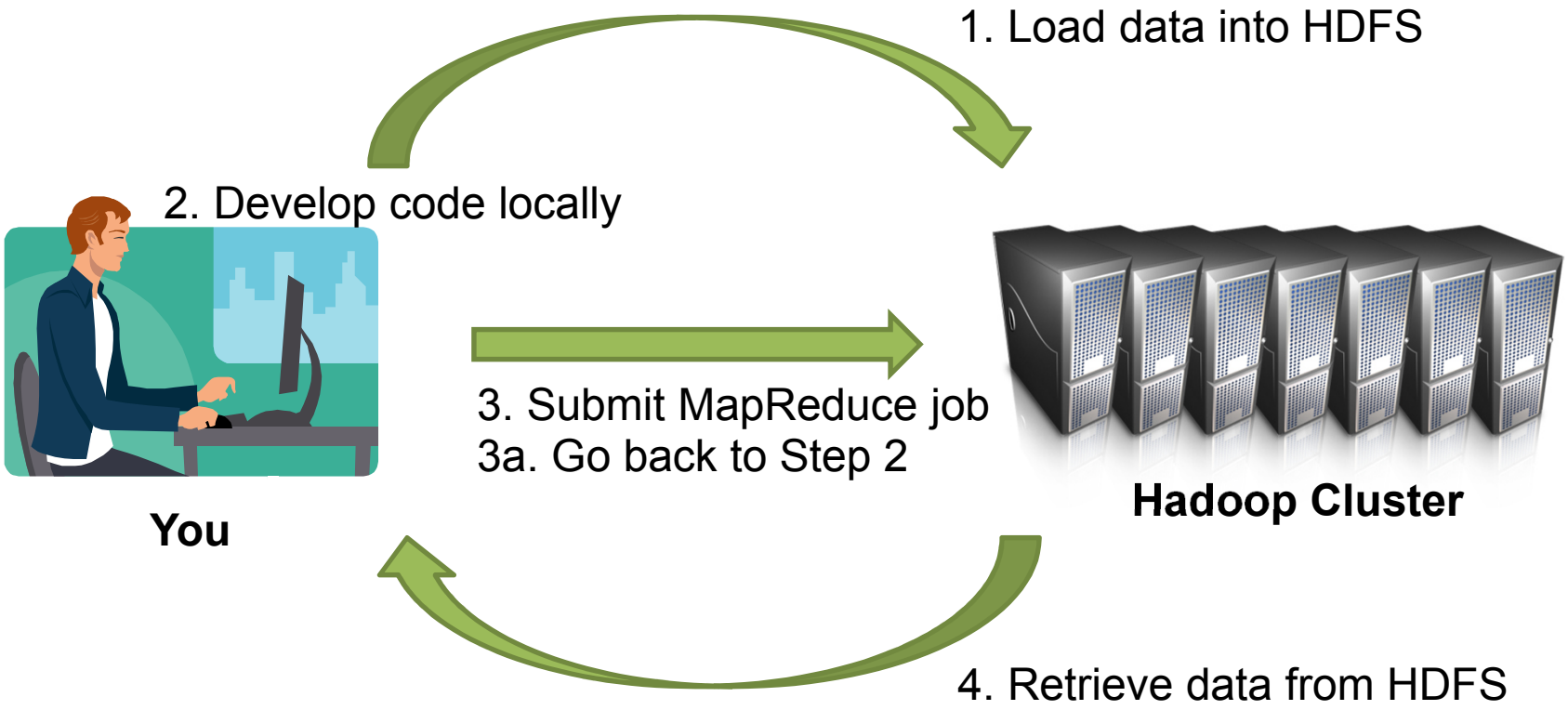
Shuffle and Sort in Hadoop

- Probably the most complex aspect of MapReduce!
- Map side
  - Map outputs are buffered in memory in a circular buffer
  - When buffer reaches threshold, contents are “spilled” to disk
  - Spills merged in a single, partitioned file (sorted within each partition): combiner runs here
- Reduce side
  - First, map outputs are copied over to reducer machine
  - “Sort” is a multi-pass merge of map outputs (happens in memory and on disk): combiner runs here
  - Final merge pass goes directly into reducer

Shuffle and Sort



Hadoop Workflow



Chaining MapReduce jobs

- Chaining in a sequence
- Chaining with complex dependency
- Chaining preprocessing and postprocessing steps

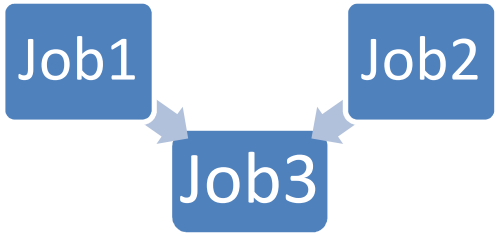
Chaining in a sequence

- Simple and straightforward
- [MAP / REDUCE]+; MAP+ / REDUCE / MAP*
- Output of last is the input to the next
- Similar to pipes

```
Configuration conf = getConf();
JobConf job = new JobConf(conf);
job.setJobName("ChainJob");
job.setInputFormat(TextInputFormat.class);
job.setOutputFormat(TextOutputFormat.class);

FileInputFormat.setInputPaths(job, in);
FileOutputFormat.setOutputPath(job, out);

JobConf map1Conf = new JobConf(false);

ChainMapper.addMapper(job, Map1.class, LongWritable.class, Text.class,
Text.class, Text.class, true, map1Conf);
```

Chaining with complex dependency

- Jobs are not chained in a linear fashion



- Use *addDependingJob()* method to add dependency information:

```
x.addDependingJob(y)
```

Chaining preprocessing and postprocessing steps

- Example: remove stop words in IR
- Approaches:
  - Separate: inefficient
  - Chaining those steps into a single job
    - Use ChainMapper.addMapper() and ChainReducer.setReducer

Map+ / Reduce / Map*

Exercises

- **Log processing with MAPREDUCE**: *A big company stores all incoming emails in log files. How can you count the frequency of each email address found in these logs with* MAPREDUCE*?*

- **SP relational queries:** *Propose a* MAPREDUCE *job (using pseudo-code for map() and reduce()) for the following queries:*

  SELECT *num, Cru*
  FROM *Vins_produits*
  WHERE *region='Bourgogne'*
  AND *millésime=2010*

  SELECT *Region,* **count***(num)*
  FROM *Vins_produits*
  WHERE *millésime=2010*
  GROUP BY *Region*

  When is the reduce function really useful?

- **Sorting with MAPREDUCE:** *How can you obtain a parallel sort with* MAPREDUCE*? For instance, what would be the* MAPREDUCE *parallel execution of the following SQL query:*

  SELECT *Cru*
  FROM *Vins_produits*
  ORDER BY millésime

- **Joins with MAPREDUCE:** *How can you express joins?*

- **Graph traversal with MAPREDUCE:** *How to express a reachability query ?*

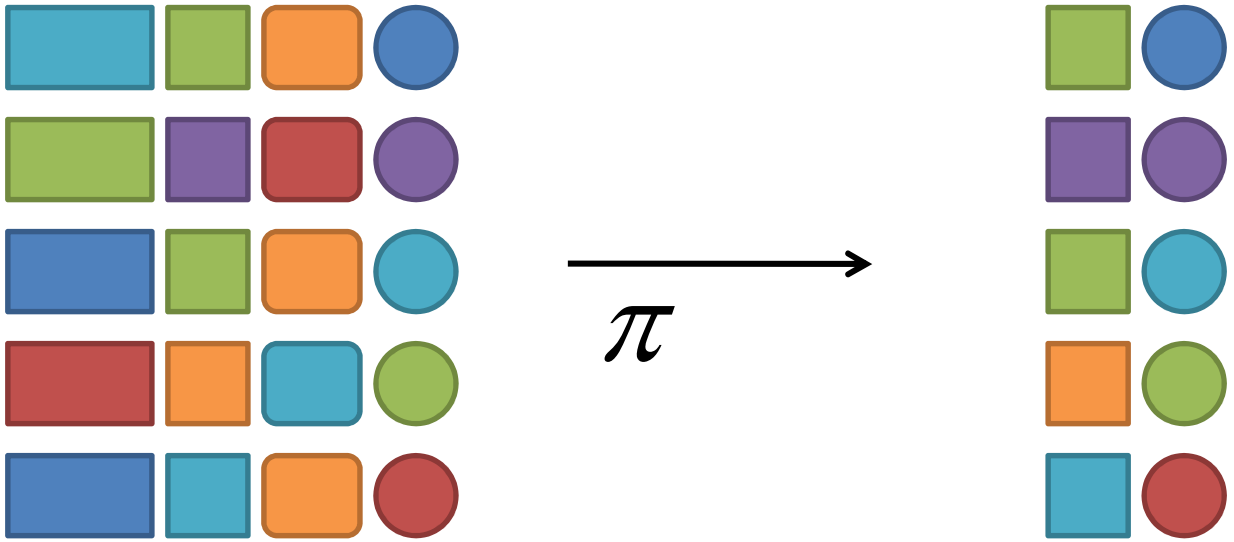
Source: http://webdam.inria.fr/Jorge/html/wdmch17.html#x23-34600016

MapReduce algorithms for processing relational data

Relational Algebra

- Primitives
  - Projection (π)
  - Selection (σ)
  - Cartesian product ($\times$)
  - Set union ($\cup$)
  - Set difference ($-$)
  - Rename (ρ)
- Other operations
  - Join ($\bowtie$)
  - Group by… aggregation
  - …

Projection

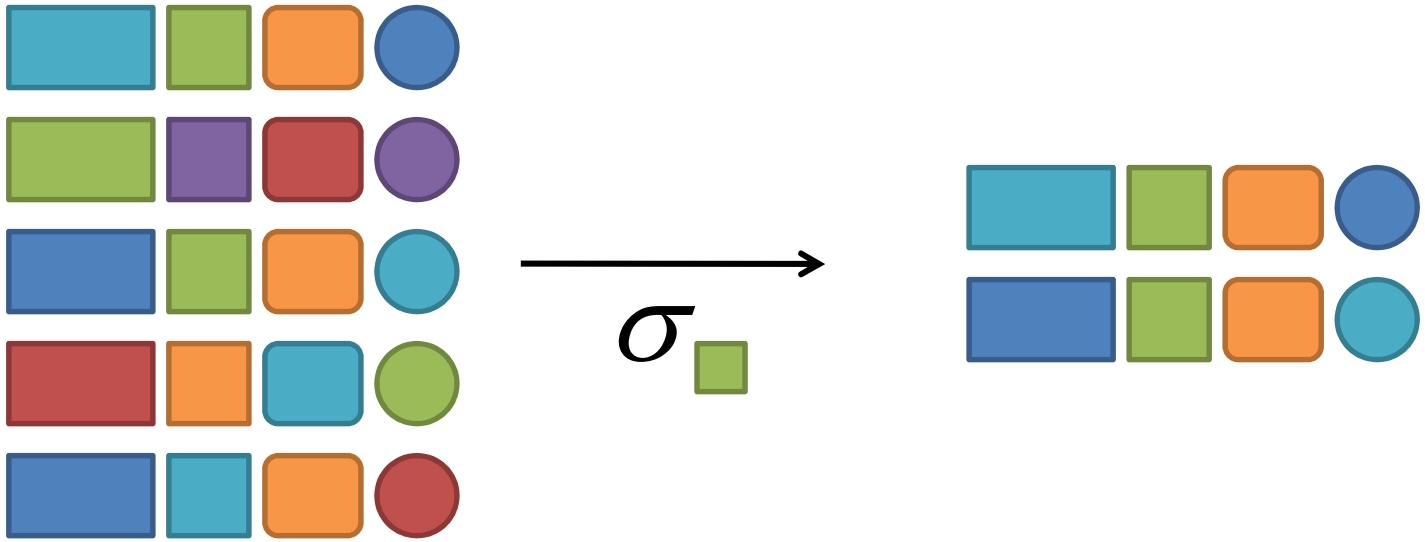


Projection in MapReduce

- Easy!
  - Map over tuples, emit new tuples with appropriate attributes
  - Reduce: take tuples that appear many times and emit only one version (duplicate elimination)
    - Tuple t in R: Map(t, t) -> (t',t')
    - Reduce (t', [t', …,t']) -> [t',t']

- Basically limited by HDFS streaming speeds
  - Speed of encoding/decoding tuples becomes important
  - Relational databases take advantage of compression
  - Semistructured data? No problem!

Selection



Selection in MapReduce

- Easy!
  - Map over tuples, emit only tuples that meet criteria
  - No reducers, unless for regrouping or resorting tuples (reducers are the identity function)
  - Alternatively: perform in reducer, after some other processing
- But very expensive!!! Has to scan the database
  - Better approaches?

Union, Set Intersection and Set Difference

- Similar ideas: each map outputs the tuple pair (t,t). For union, we output it once, for intersection only when in the reduce we have (t, [t,t])
- For Set difference?

Set Difference

- Map Function: For a tuple t in R, produce key-value pair (t, R), and for a tuple t in S, produce key-value pair (t, S).
- Reduce Function: For each key t, do the following.

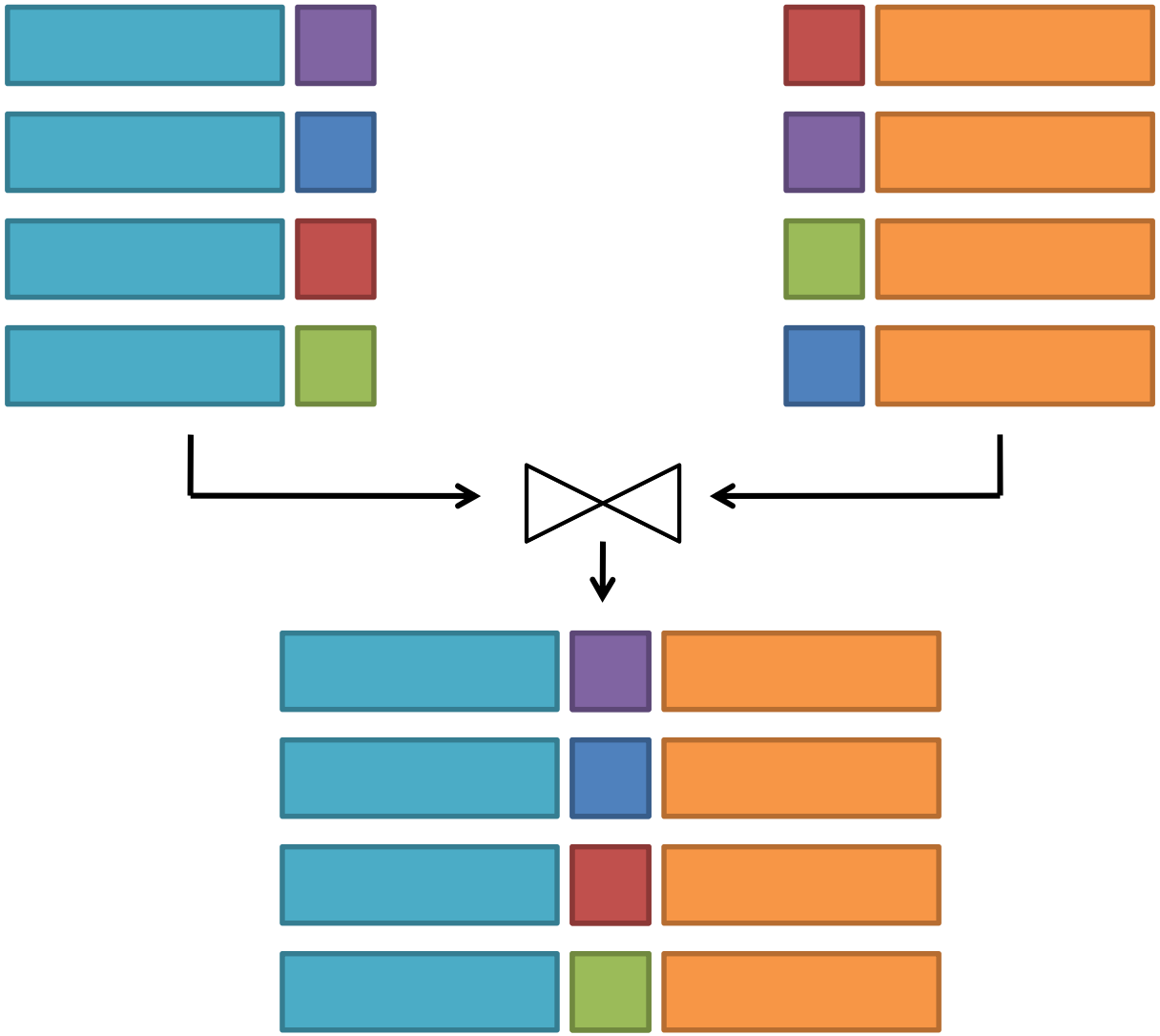
    1. If the associated value list is [R], then produce (t, t).

    2. If the associated value list is anything else, which could only be [R, S], [S, R], or [S], produce (t, NULL).

Group by… Aggregation

- Example: What is the average time spent per URL?
- In SQL:
  - SELECT url, AVG(time) FROM visits GROUP BY url
- In MapReduce:
  - Map over tuples, emit time, keyed by url
  - Framework automatically groups values by keys
  - Compute average in reducer
  - Optimize with combiners

Relational Joins



Join Algorithms in MapReduce

- Reduce-side join
- Map-side join
- In-memory join
  - Striped variant
  - Memcached variant

Reduce-side Join

- Basic idea: group by join key
  - Map over both sets of tuples
  - Emit tuple as value with join key as the intermediate key
  - Execution framework brings together tuples sharing the same key
  - Perform actual join in reducer
  - Similar to a “sort-merge join” in database terminology
- Two variants
  - 1-to-1 joins
  - 1-to-many and many-to-many joins

Reduce-side join

- Map
  - output <key, value>
  - key>>join key, value>>tagged with data source
- Reduce
  - do a full cross-product of values
  - output the combination results

Example

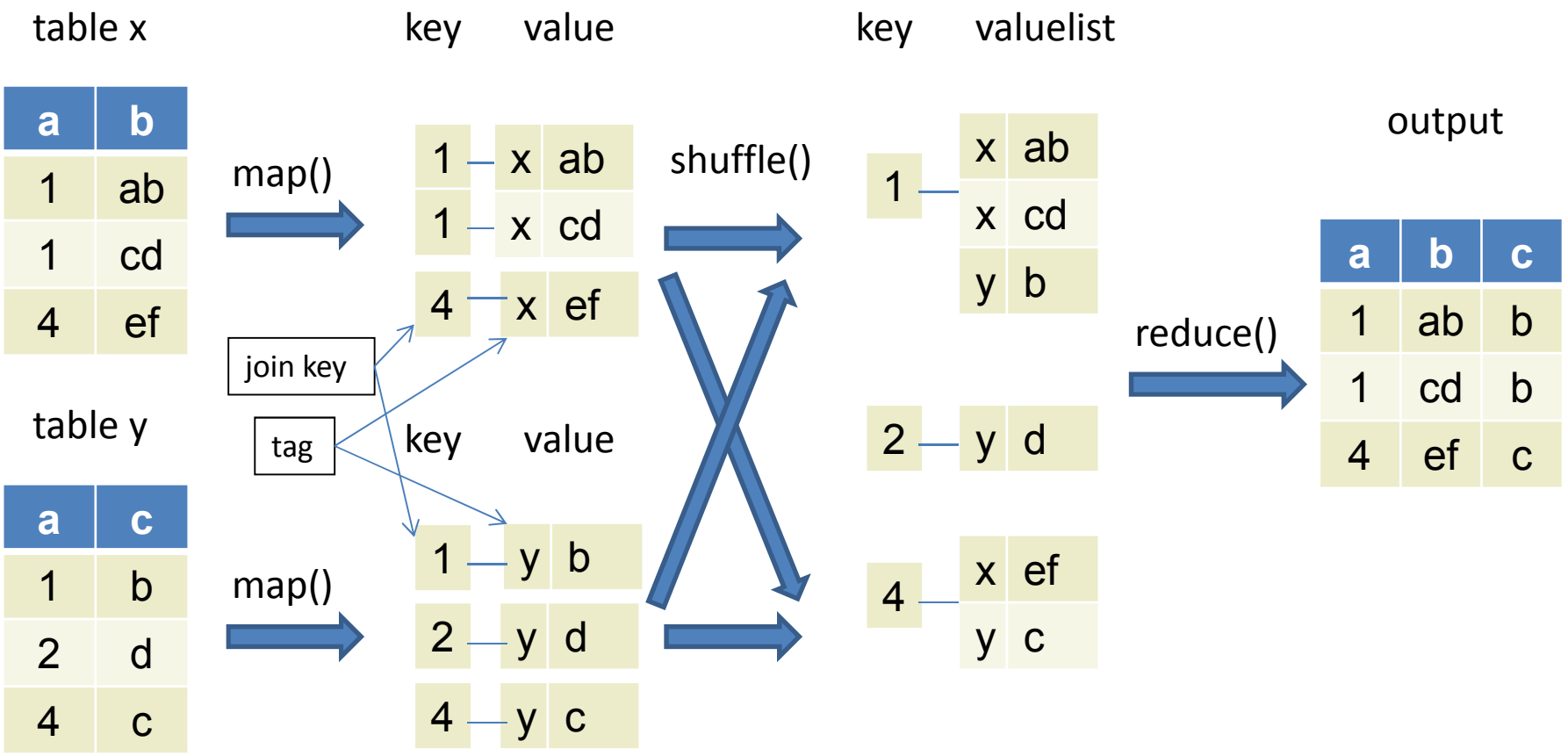


table x
a b
1 ab
1 cd
4 ef
table y
a c
1 b
2 d
4 c
map()
map()
join key
tag
key value
1 x ab
1 x cd
4 x ef
1 y b
2 y d
4 y c
shuffle()
key valuelist
1 x ab
x cd
y b
2 y d
4 x ef
y c
reduce()
output
a b c
1 ab b
1 cd b
4 ef c

In-Memory Join (broadcasted join)

- Basic idea: load one dataset into memory, stream over other dataset
  - Works if R << S and R fits into memory
  - It's a "replicated hash join"

- MapReduce implementation
  - Distribute R to all nodes
  - Map over S, each mapper loads R in memory, hashed by join key
  - For every tuple in S, look up join key in R
  - No reducers, unless for regrouping or resorting tuples

In-Memory Join: Variants

- Striped variant:
  - R too big to fit into memory?
  - Divide R into R_1, R_2, R_3, … s.t. each R_n fits into memory
  - Perform in-memory join: $\forall n$, $R_n \bowtie S$
  - Take the union of all join results

- Memcached join:
  - Load R into memcached (ditributed cache)
  - Replace in-memory hash lookup with memcached lookup

Memcached Join

- Memcached join:
  - Load R into memcached
  - Replace in-memory hash lookup with memcached lookup
- Capacity and scalability?
  - Memcached capacity >> RAM of individual node
  - Memcached scales out with cluster
- Latency?
  - Memcached is fast (basically, speed of network)
  - Batch requests to amortize latency costs

Source: See tech report by Lin et al. (2009)

Map-side Join: Parallel Scans

- If datasets are sorted by join key, join can be accomplished by a scan over both datasets
- How can we accomplish this in parallel?
  - Partition and sort both datasets in the same manner
- In MapReduce:
  - Map over one dataset, read from other corresponding partition
  - No reducers necessary (unless to repartition or resort)
- Consistently partitioned datasets: realistic to expect?

Which join to use?

- In-memory join > map-side join > reduce-side join
  - Why?
- Limitations of each?
  - In-memory join: memory
  - Map-side join: sort order and partitioning
  - Reduce-side join: general purpose

Processing Relational Data: Summary

- MapReduce algorithms for processing relational data:
  - Group by, sorting, partitioning are handled automatically by shuffle/sort in MapReduce
  - Selection, projection, and other computations (e.g., aggregation), are performed either in mapper or reducer
  - Multiple strategies for relational joins
- Complex operations require multiple MapReduce jobs
  - Example: top ten URLs in terms of average time spent
  - Opportunities for automatic optimization

Graph algorithms in MapReduce

Graphs

- G = (V,E), where
  - V represents the set of vertices (nodes)
  - E represents the set of edges (links)
  - Both vertices and edges may contain additional information

Graphs and MapReduce

- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph
- Key questions:
  - How do you represent graph data in MapReduce?
  - How do you traverse a graph in MapReduce?

Representing Graphs

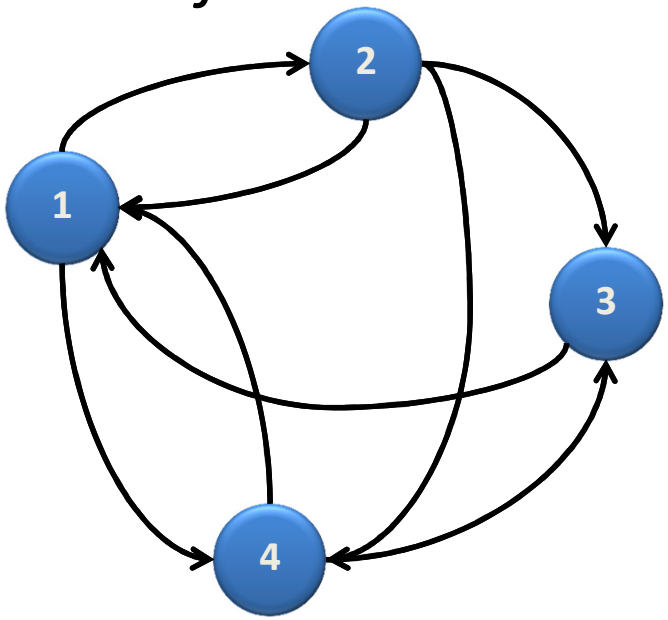
- G = (V, E)
- Two common representations
  - Adjacency matrix
  - Adjacency list

Adjacency Matrices

Represent a graph as an *n* x *n* square matrix *M*

- *n* = |V|
- M_{ij} = 1 means a link from node *i* to *j*

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0



Adjacency Matrices: Critique

- Advantages:
  - Amenable to mathematical manipulation
  - Iteration over rows and columns corresponds to computations on outlinks and inlinks
- Disadvantages:
  - Lots of zeros for sparse matrices
  - Lots of wasted space

Adjacency Lists

Take adjacency matrices… and throw away all the zeros

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0

1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3

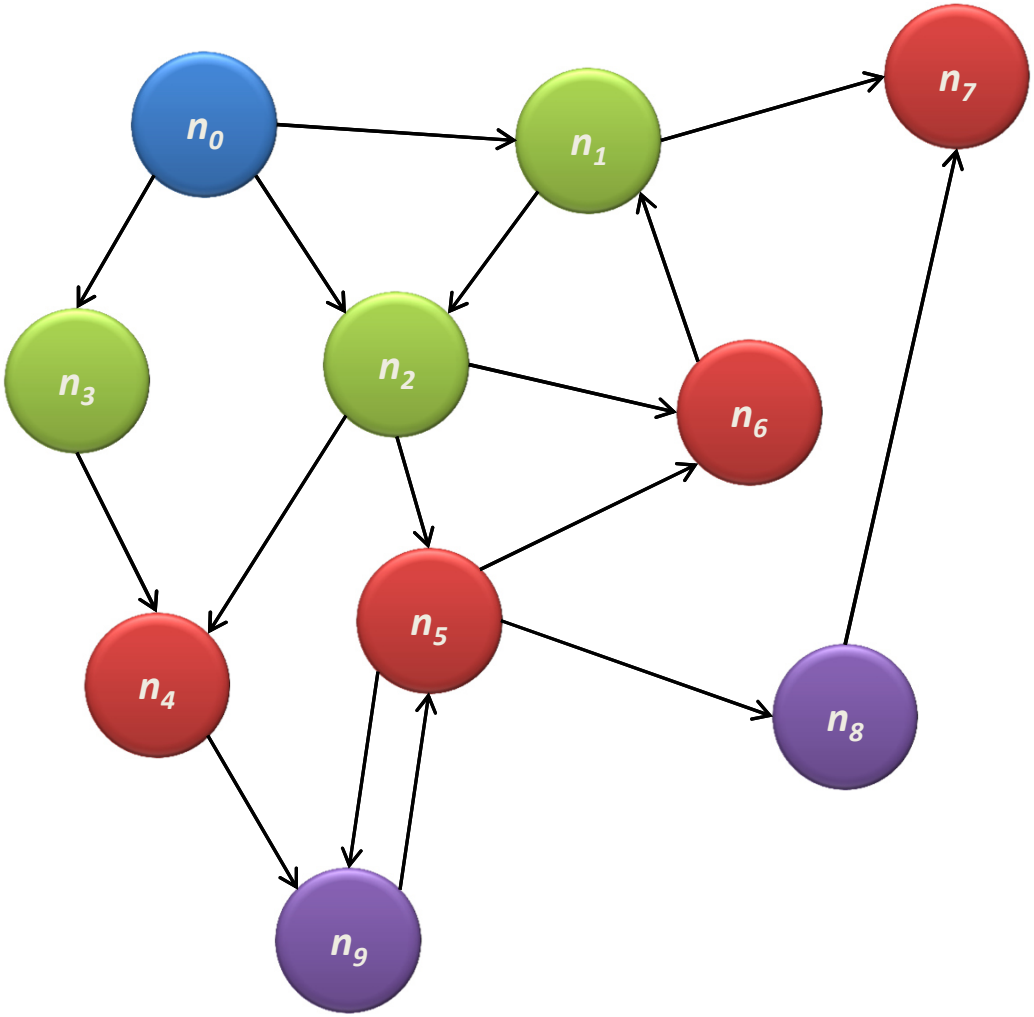
Adjacency Lists: Critique

- Advantages:
  - Much more compact representation
  - Easy to compute over outlinks
- Disadvantages:
  - Much more difficult to compute over inlinks

Finding the Shortest Path

- Consider simple case of equal edge weights
- Solution to the problem can be defined inductively
- Here's the intuition:
  - Define: *b* is reachable from *a* if *b* is on adjacency list of *a*
  - DISTANCETO(*s*) = 0
  - For all nodes *p* reachable from *s*,
    DISTANCETO(*p*) = 1
  - For all nodes *n* reachable from some other set of nodes *M*,
    DISTANCETO(*n*) = 1 + min(DISTANCETO(*m*), $m \in M$)

Visualizing Parallel BFS



From Intuition to Algorithm

- Data representation:
  - Key: node *n*
  - Value: *d* (distance from start), adjacency list (list of nodes reachable from *n*)
  - Initialization: for all nodes except for start node, $d = \infty$
- Mapper:
  - $\forall m \in$ adjacency list: emit (m, $d + 1$)
- Sort/Shuffle
  - Groups distances by reachable nodes
- Reducer:
  - Selects minimum distance path for each reachable node
  - Additional bookkeeping needed to keep track of actual path

Multiple Iterations Needed

- Each MapReduce iteration advances the "known frontier" by one hop
  - Subsequent iterations include more and more reachable nodes as frontier expands
  - Multiple iterations are needed to explore entire graph
- Preserving graph structure:
  - Problem: Where did the adjacency list go?
  - Solution: mapper emits (*n*, adjacency list) as well

BFS Pseudo-Code

```
1: class Mapper
2:     method Map(nid n, node N)
3:         d ← N.Distance
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, d + 1)                  ▷ Emit distances to reachable nodes

1: class Reducer
2:     method Reduce(nid m, [d1, d2, ...])
3:         d_min ← ∞
4:         M ← ∅
5:         for all d ∈ counts [d1, d2, ...] do
6:             if IsNode(d) then
7:                 M ← d                           ▷ Recover graph structure
8:             else if d < d_min then              ▷ Look for shorter distance
9:                 d_min ← d
10:        M.Distance ← d_min                      ▷ Update shortest distance
11:        Emit(nid m, node M)
```

Stopping Criterion

- How many iterations are needed in parallel BFS (equal edge weight case)?
- Convince yourself: when a node is first "discovered", we've found the shortest path
- Now answer the question...
  - Six degrees of separation?

Graphs and MapReduce

- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph
- Generic recipe:
  - Represent graphs as adjacency lists
  - Perform local computations in mapper
  - Pass along partial results via outlinks, keyed by destination node
  - Perform aggregation in reducer on inlinks to a node
  - Iterate until convergence: controlled by external “driver”
  - Don’t forget to pass the graph structure between iterations

Random Walks Over the Web

- Random surfer model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- PageRank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
- PageRank captures notions of page importance
  - One of thousands of features used in web search
  - Note: query-independent

PageRank: Defined

Given page *x* with inlinks $t_1 \ldots t_n$, where

- *C(t)* is the out-degree of *t*
- α is probability of random jump
- *N* is the total number of nodes in the graph

$$PR(x) = \alpha\left(\frac{1}{N}\right) + (1-\alpha)\sum_{i=1}^{n}\frac{PR(t_i)}{C(t_i)}$$

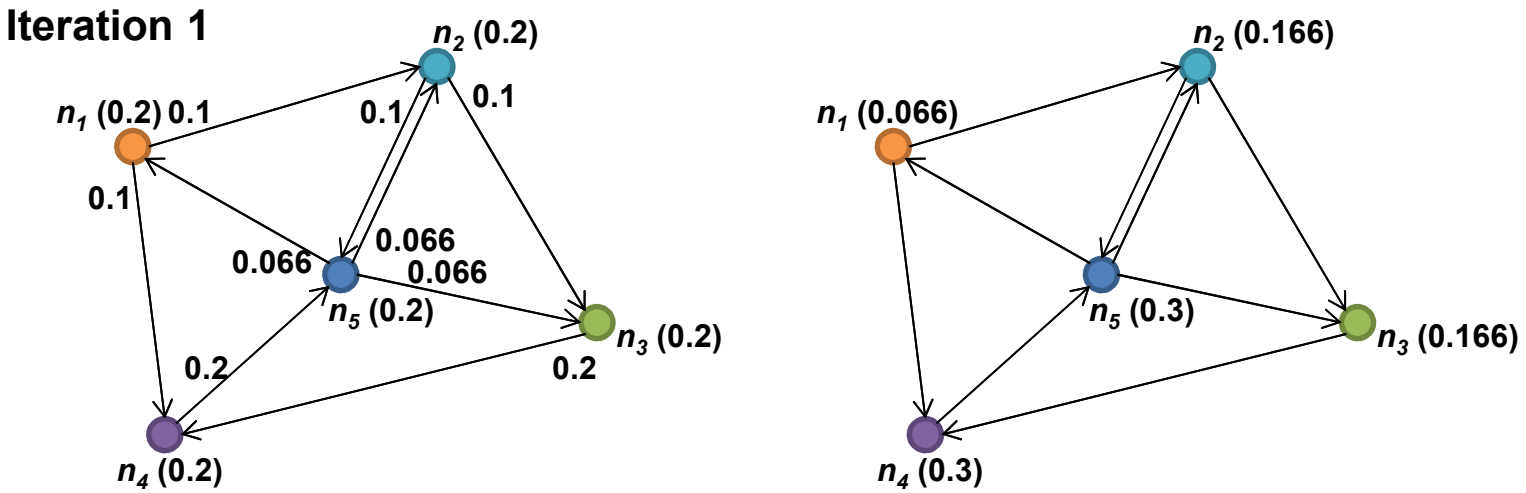
Computing PageRank

- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed PR_i values
  - Each page distributes PR_i "credit" to all pages it links to
  - Each target page adds up "credit" from multiple in-bound links to compute PR_{i+1}
  - Iterate until values converge

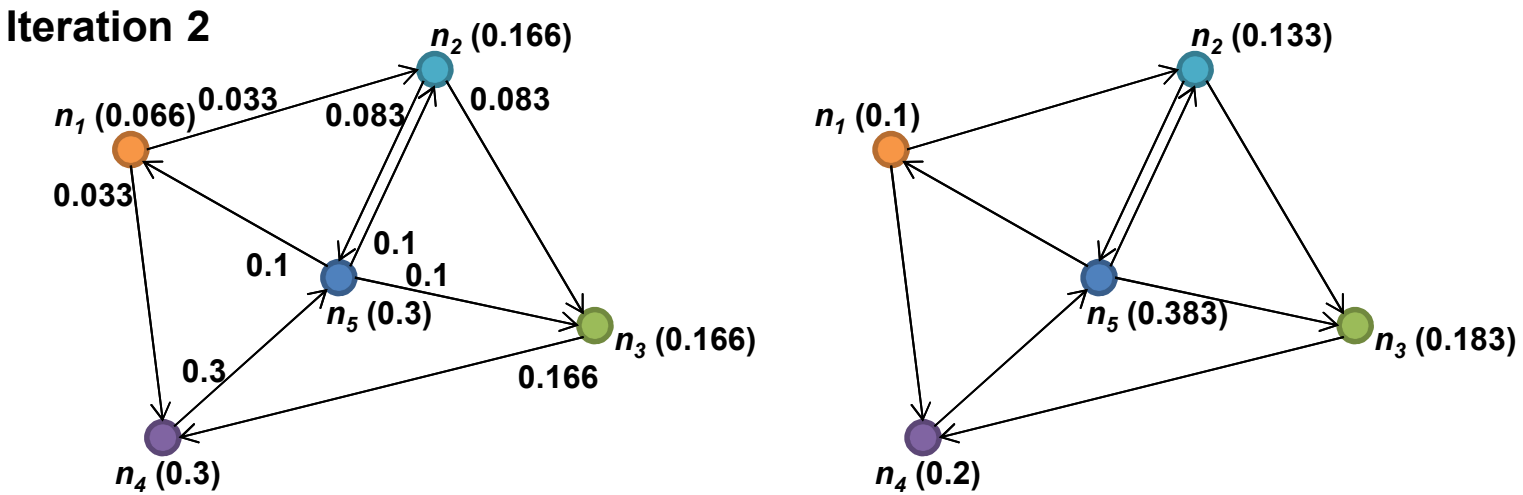
Simplified PageRank

- First, tackle the simple case:
  - No random jump factor
  - No dangling links
- Then, factor in these complexities…
  - Why do we need the random jump?
  - Where do dangling links come from?

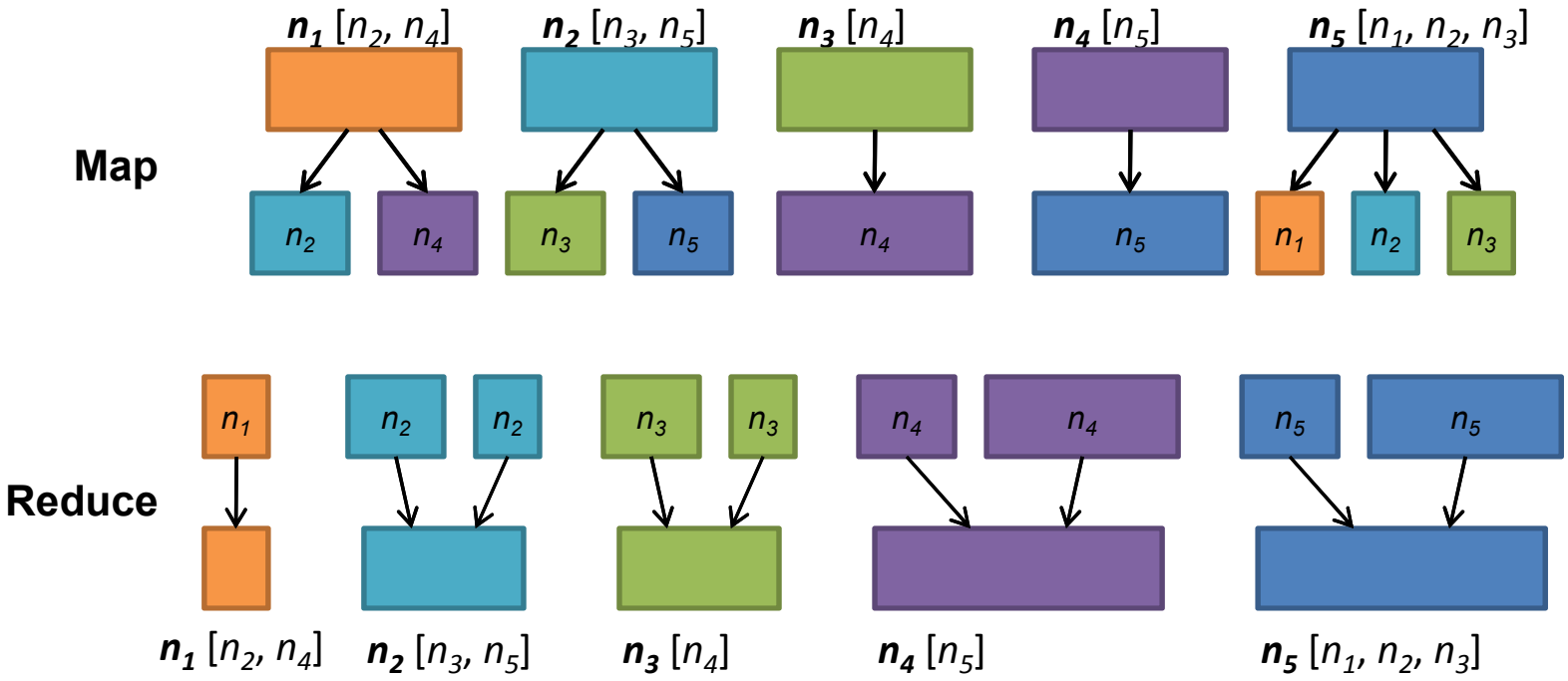
Sample PageRank Iteration (1)



Sample PageRank Iteration (2)



PageRank in MapReduce



PageRank Pseudo-Code

```
1: class Mapper
2:     method Map(nid n, node N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, p)                      ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ counts [p1, p2, ...] do
5:             if IsNode(p) then
6:                 M ← p                           ▷ Recover graph structure
7:             else
8:                 s ← s + p                       ▷ Sums incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

Complete PageRank

- Two additional complexities
  - What is the proper treatment of dangling nodes?
  - How do we factor in the random jump factor?
- Solution:
  - Second pass to redistribute "missing PageRank mass" and account for random jumps

$$p' = \alpha\left(\frac{1}{|G|}\right) + (1-\alpha)\left(\frac{m}{|G|} + p\right)$$

  - p is PageRank value from before, p' is updated PageRank value
  - $|G|$ is the number of nodes in the graph
  - m is the missing PageRank mass

PageRank Convergence

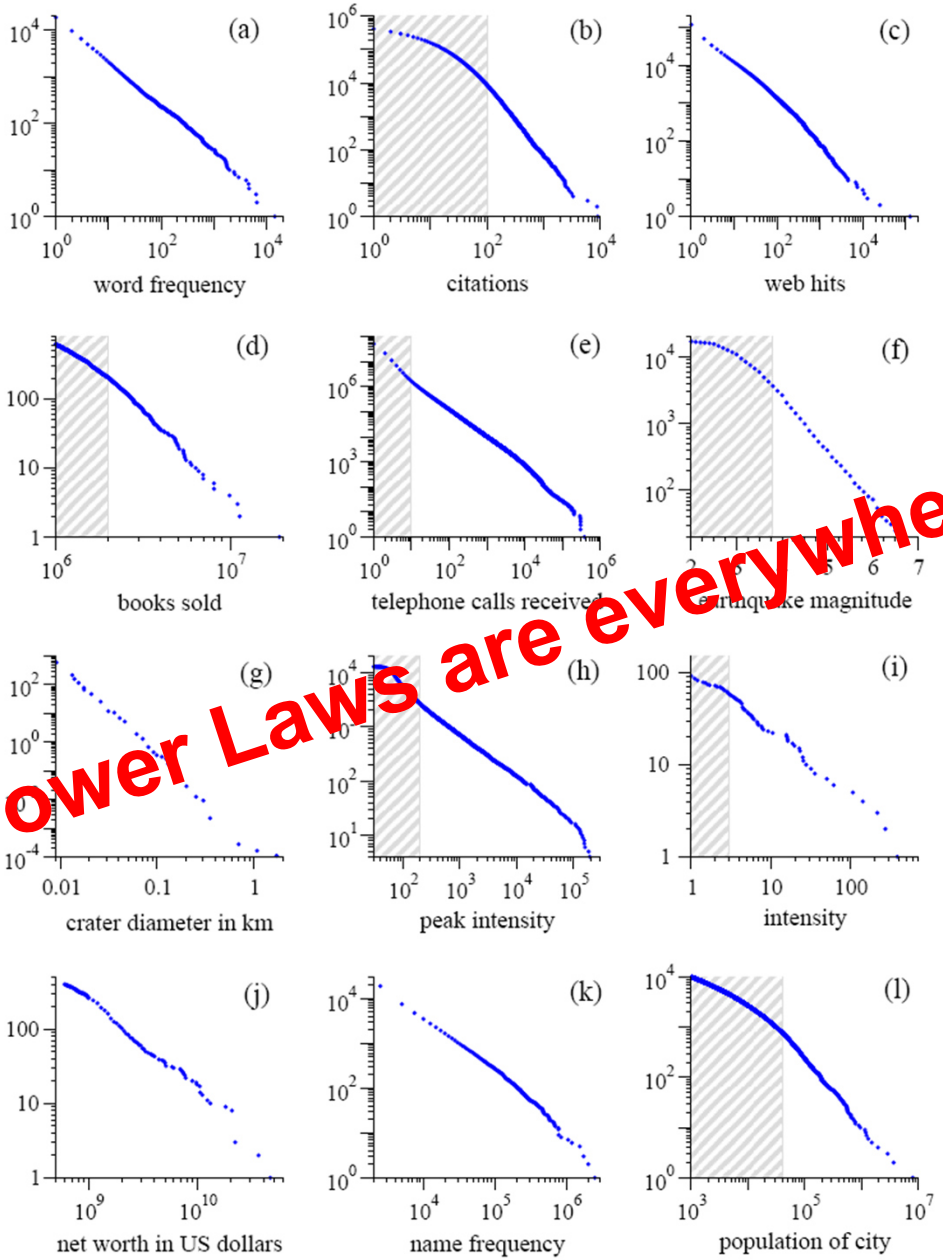
- Alternative convergence criteria
  - Iterate until PageRank values don't change
  - Iterate until PageRank rankings don't change
  - Fixed number of iterations
- Convergence for web graphs?

Beyond PageRank

- Link structure is important for web search
  - PageRank is one of many link-based features: HITS, SALSA, etc.
  - One of many thousands of features used in ranking…
- Adversarial nature of web search
  - Link spamming
  - Spider traps
  - Keyword stuffing
  - …

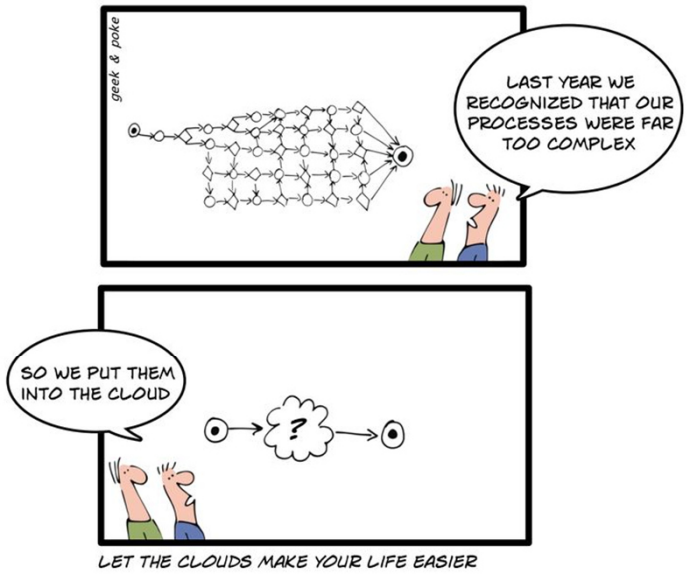
Efficient Graph Algorithms

- Sparse vs. dense graphs
- Graph topologies



Power Laws are everywhere!

Questions ?


geek & poke
LAST YEAR WE RECOGNIZED THAT OUR PROCESSES WERE FAR TOO COMPLEX
SO WE PUT THEM INTO THE CLOUD
?
LET THE CLOUDS MAKE YOUR LIFE EASIER