UNIVERSITY OF WATERLOO

Big Data Infrastructure

CS 489/698 Big Data Infrastructure (Winter 2016)

Week 5: Analyzing Graphs (2/2)

February 4, 2016

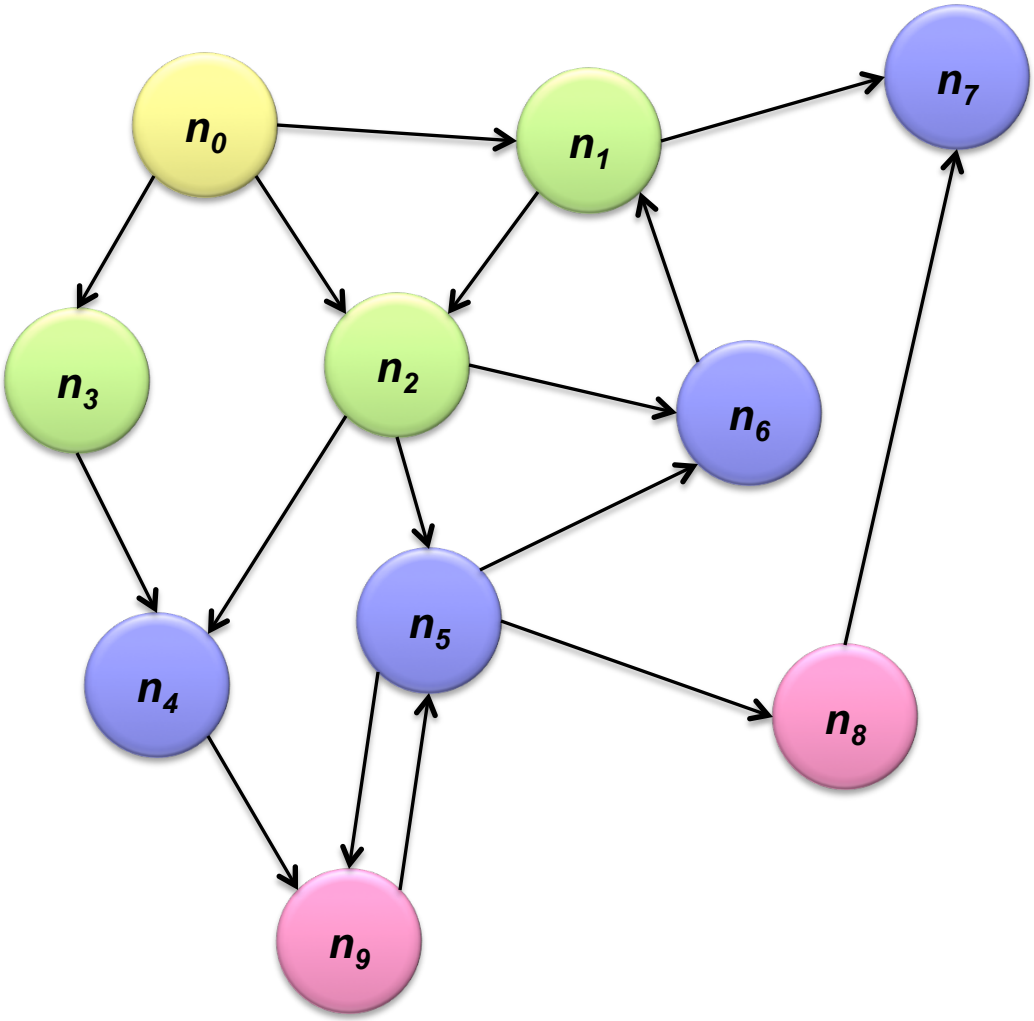
Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo

These slides are available at http://lintool.github.io/bigdata-2016w/

Single Source Shortest Path

Source: Wikipedia (Wave)

Visualizing Parallel BFS



From Intuition to Algorithm

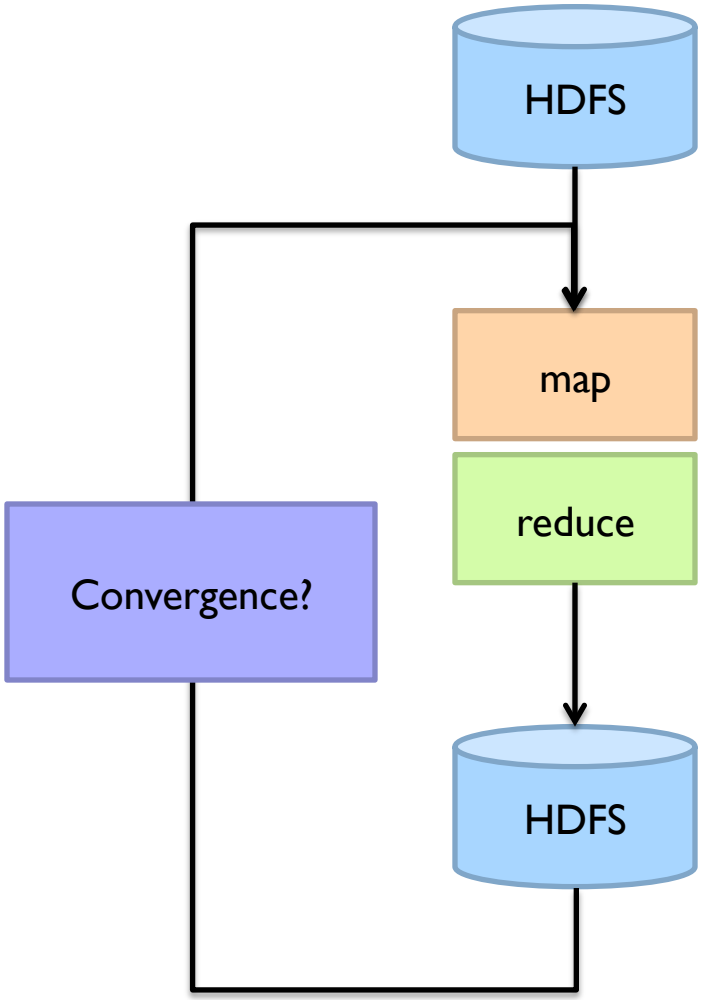
- Data representation:
  - Key: node *n*
  - Value: *d* (distance from start), adjacency list (nodes reachable from *n*)
  - Initialization: for all nodes except for start node, $d = \infty$
- Mapper:
  - $\forall m \in$ adjacency list: emit $(m, d + 1)$
  - Remember to also emit distance to yourself
- 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 "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

Ugh! This is ugly!

Implementation Practicalities



Application: Social Search

Source: Wikipedia (Crowd)

Social Search

- When searching, how to rank friends named "John"?
  - Assume undirected graphs
  - Rank matches by distance to user
- Naïve implementations:
  - Precompute all-pairs distances
  - Compute distances at query time
- Can we do better?

All-Pairs?

- Floyd-Warshall Algorithm: difficult to MapReduce-ify…
- Multiple-source shortest paths in MapReduce: run multiple parallel BFS *simultaneously*
  - Assume source nodes $\{s_0, s_1, \ldots s_n\}$
  - Instead of emitting a single distance, emit an array of distances, with respect to each source
  - Reducer selects minimum for each element in array
- Does this scale?

Landmark Approach (aka sketches)

- Select *n* seeds $\{s_0, s_1, \ldots s_n\}$
- Compute distances from seeds to every node:

Nodes

A = [2, 1, 1]
B = [1, 1, 2]
C = [4, 3, 1]
D = [1, 2, 4]

Distances to seeds

  - What can we conclude about distances?
  - Insight: landmarks bound the maximum path length
- Lots of details:
  - How to more tightly bound distances
  - How to select landmarks (random isn't the best…)
- Use multi-source parallel BFS implementation in MapReduce!

Graphs and MapReduce

- A large class of graph algorithms 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

PageRank

(The original "secret sauce" for evaluating the importance of web pages)

(What's the "Page" in PageRank?)

Random Walks Over the Web

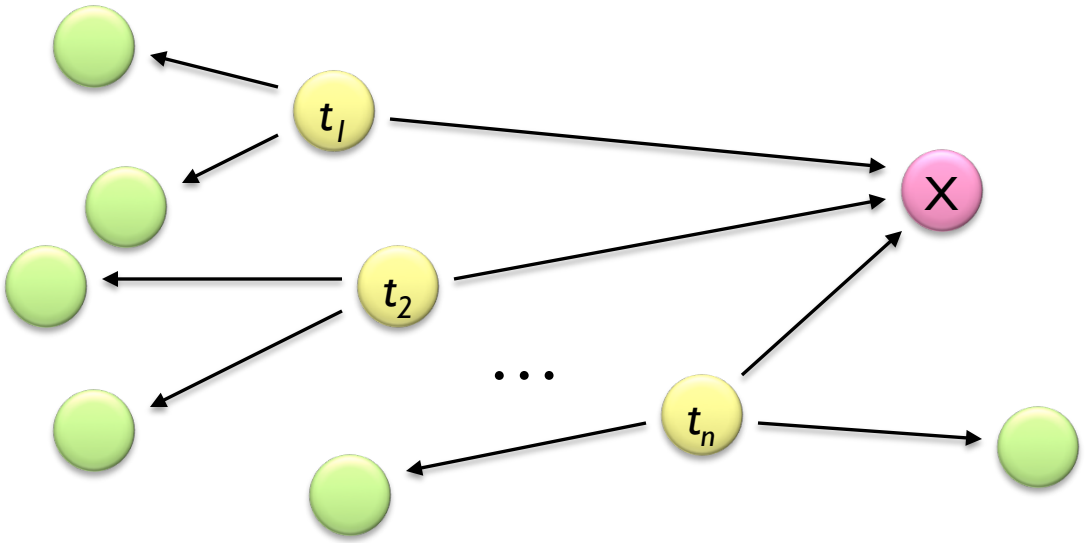
- Random surfer model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- PageRank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
- PageRank captures notions of page importance
  -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

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

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



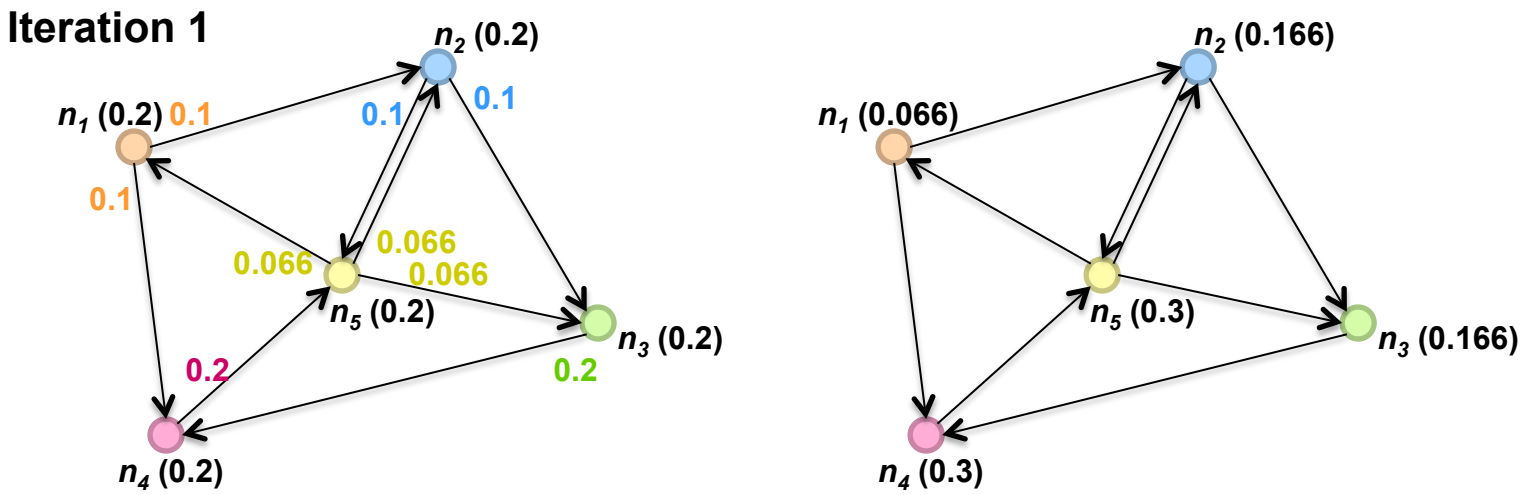
Computing PageRank

- 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

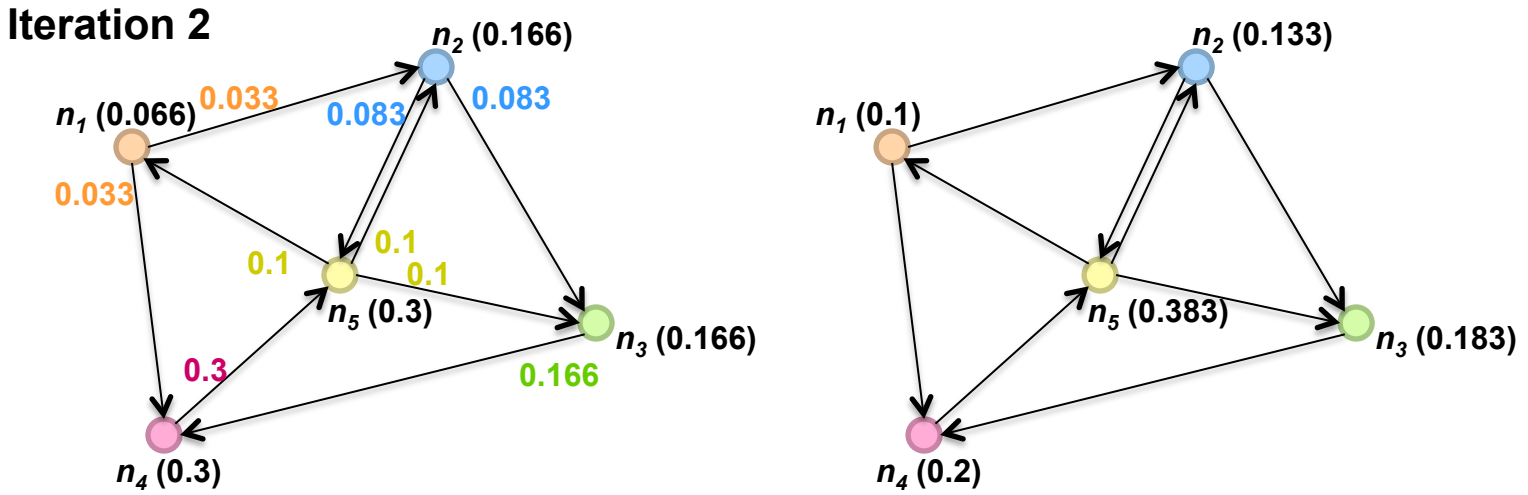
Simplified PageRank

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

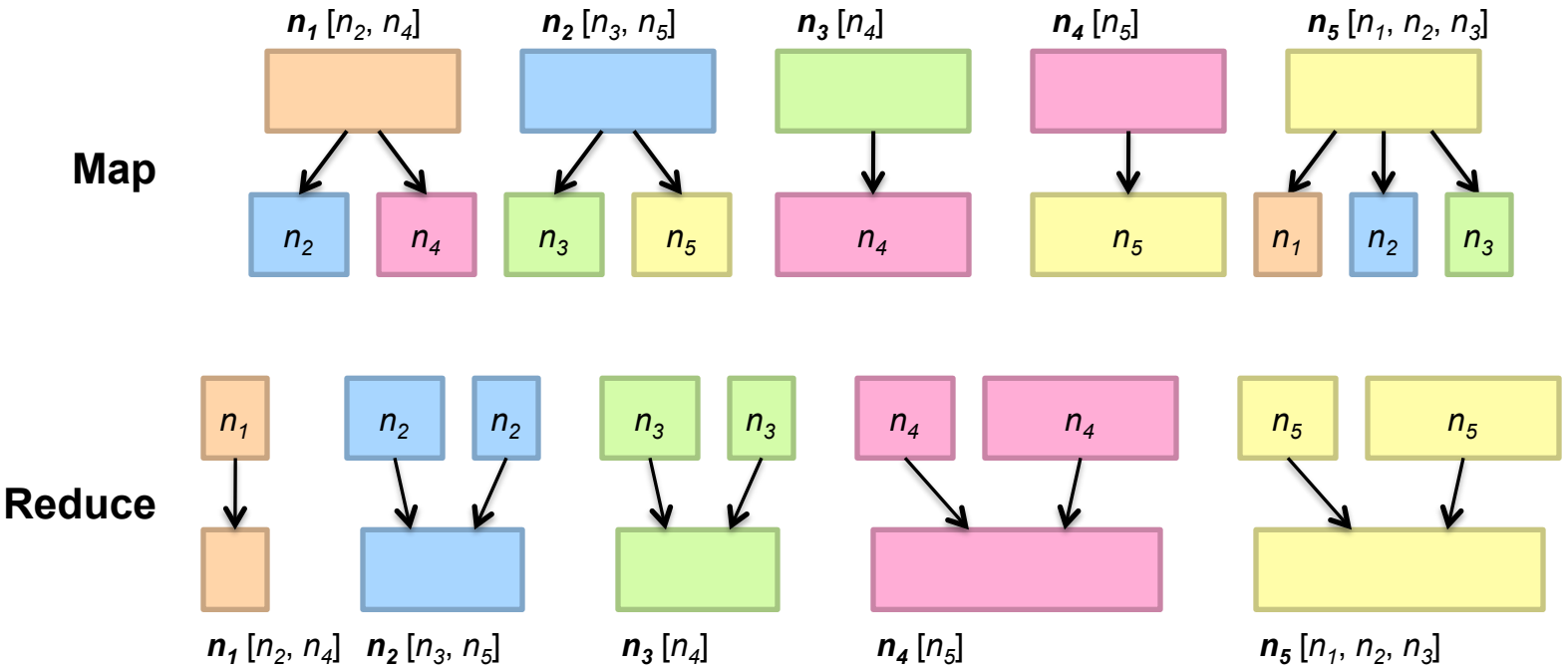
Sample PageRank Iteration (1)



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)
```

PageRank vs. BFS

	PageRank	BFS
Map	PR/N	d+1
Reduce	sum	min

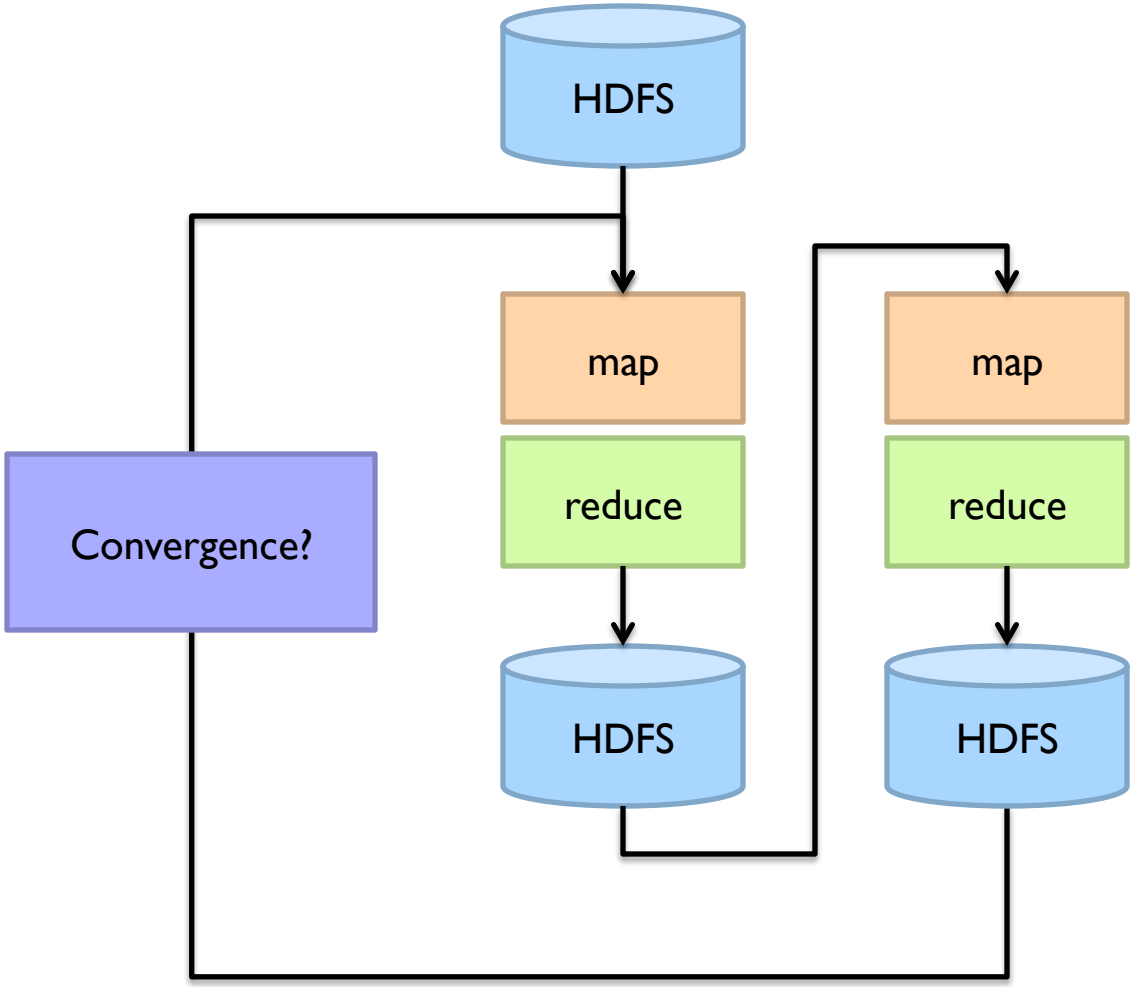
Complete PageRank

- 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

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

  - *p* is PageRank value from before, *p*' is updated PageRank value
  - *N* is the number of nodes in the graph
  - *m* is the missing PageRank mass
- Additional optimization: make it a single pass!

Implementation Practicalities



Ugh. Spark to the rescue?

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?
  - Not a straightforward question
- Watch out for link spam and the perils of SEO:
  - Link farms
  - Spider traps
  - …

Beyond PageRank

- Variations of PageRank
  - Weighted edges
  - Personalized PageRank
- Variants on graph random walks
  - Hubs and authorities (HITS)
  - SALSA

Applications

- Static prior for web ranking
- Identification of “special nodes” in a network
- Link recommendation
- Additional feature in any machine learning problem

More Implementation Practicalities

- How do you actually extract the webgraph?
- Lots of details…

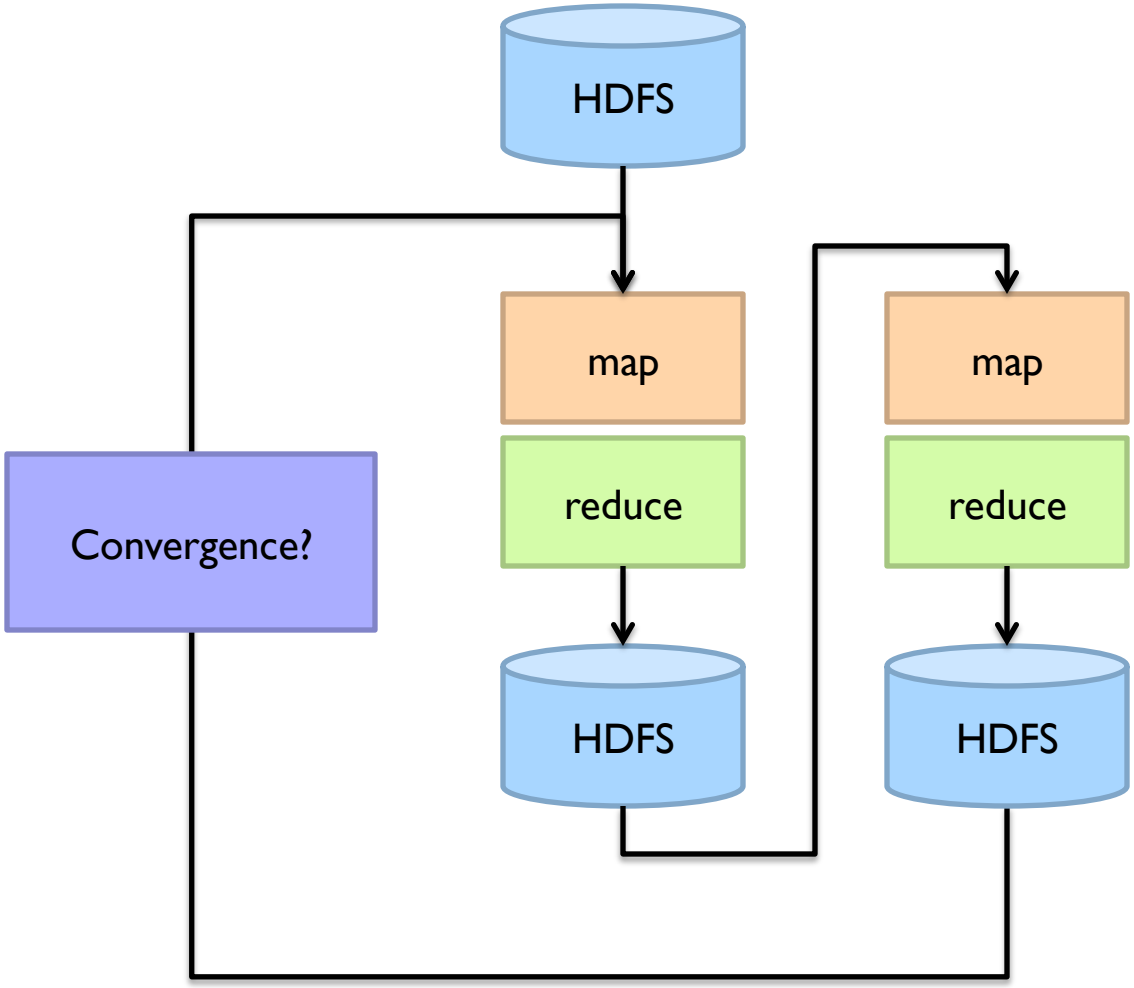
Source: http://www.flickr.com/photos/fusedforces/4324320625/

MapReduce Sucks

- Java verbosity
- Hadoop task startup time
- Stragglers
- Needless graph shuffling
- Checkpointing at each iteration

Spark to the rescue?

Implementation Practicalities

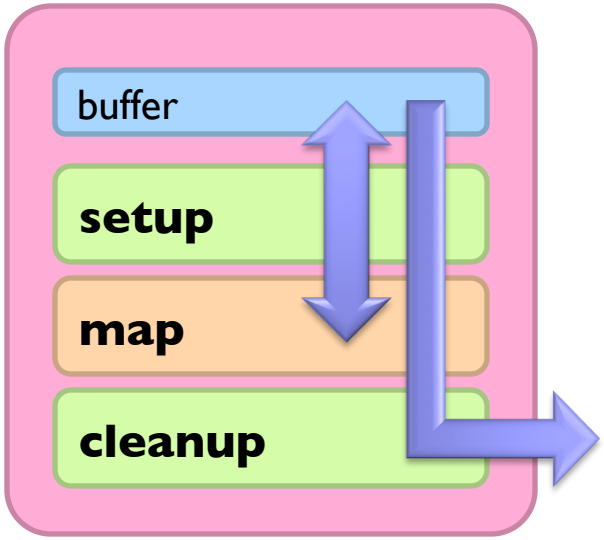


Iterative Algorithms

Source: Wikipedia (Water wheel)

In-Mapper Combining

- Use combiners
  - Perform local aggregation on map output
  - Downside: intermediate data is still materialized
- Better: in-mapper combining
  - Preserve state across multiple map calls, aggregate messages in buffer, emit buffer contents at end
  - Downside: requires memory management



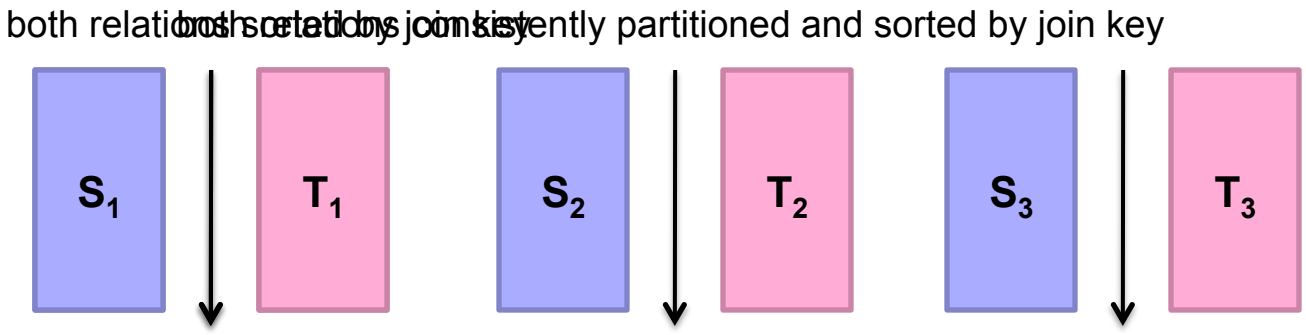
Emit all key-value pairs at once

Better Partitioning

- Default: hash partitioning
  - Randomly assign nodes to partitions
- Observation: many graphs exhibit local structure
  - E.g., communities in social networks
  - Better partitioning creates more opportunities for local aggregation
- Unfortunately, partitioning is **hard**!
  - Sometimes, chick-and-egg…
  - But cheap heuristics sometimes available
  - For webgraphs: range partition on domain-sorted URLs

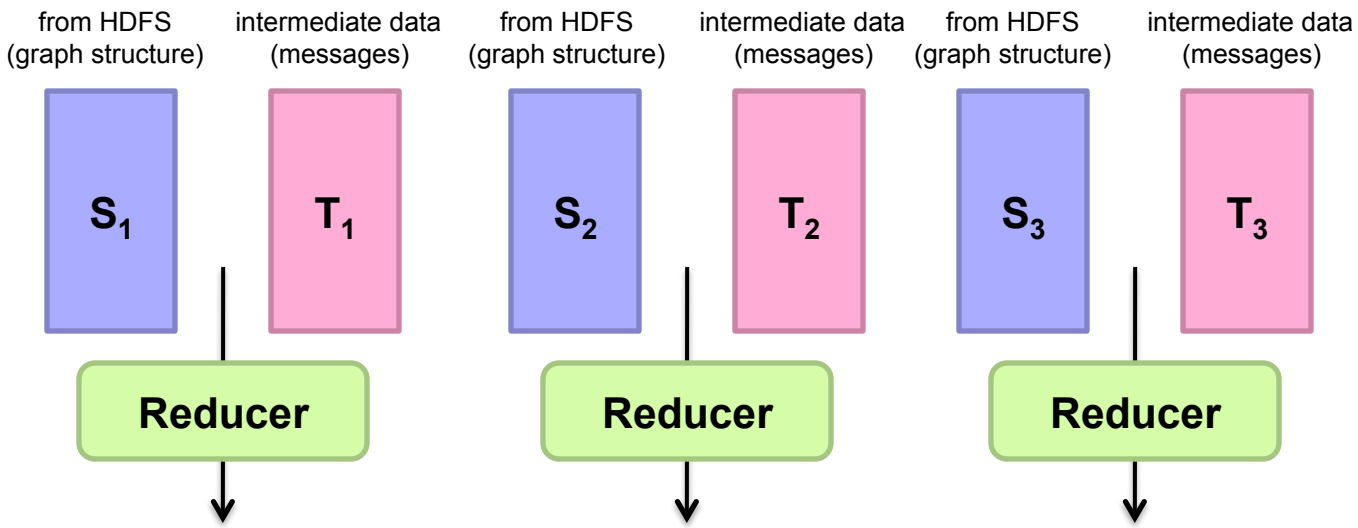
Schimmy Design Pattern

- Basic implementation contains two dataflows:
  - Messages (actual computations)
  - Graph structure ("bookkeeping")
- Schimmy: separate the two dataflows, shuffle only the messages
  - Basic idea: merge join between graph structure and messages



Do the Schimmy!

- Schimmy = reduce side parallel merge join between graph structure and messages
  - Consistent partitioning between input and intermediate data
  - Mappers emit only messages (actual computation)
  - Reducers read graph structure directly from HDFS

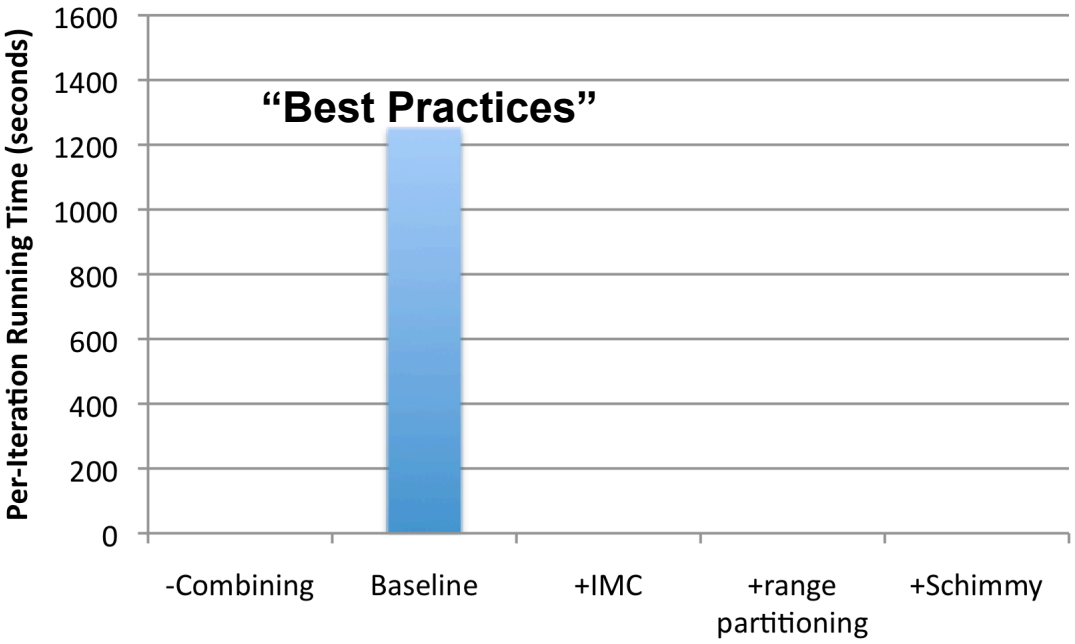


Experiments

- Cluster setup:
  - 10 workers, each 2 cores (3.2 GHz Xeon), 4GB RAM, 367 GB disk
  - Hadoop 0.20.0 on RHELS 5.3
- Dataset:
  - First English segment of ClueWeb09 collection
  - 50.2m web pages (1.53 TB uncompressed, 247 GB compressed)
  - Extracted webgraph: 1.4 billion links, 7.0 GB
  - Dataset arranged in crawl order
- Setup:
  - Measured per-iteration running time (5 iterations)
  - 100 partitions

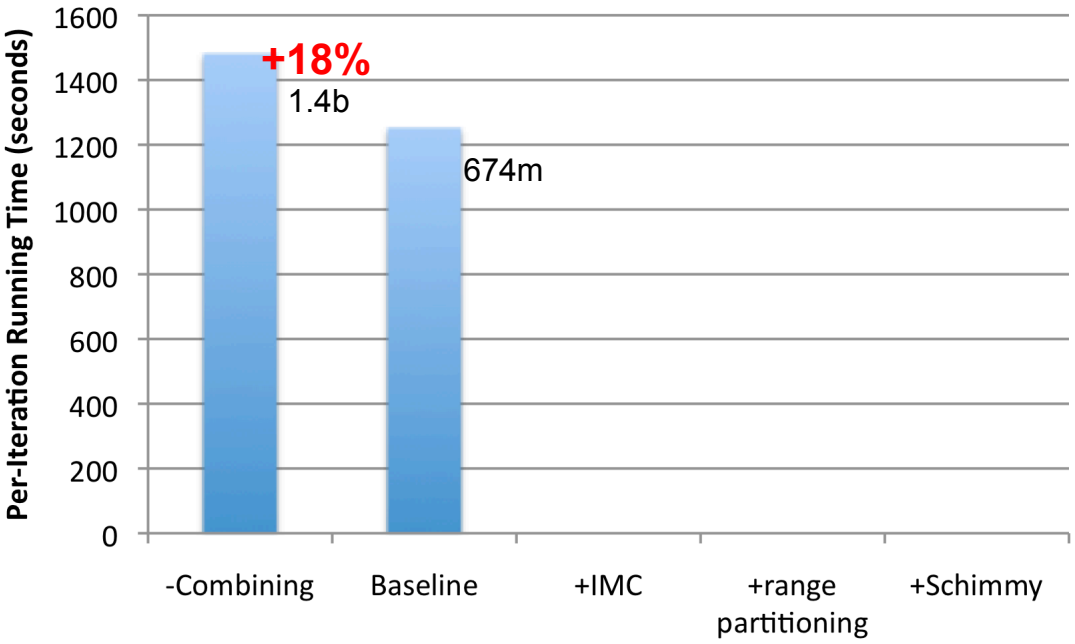
From: Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. Proceedings of the Eighth Workshop on Mining and Learning with Graphs Workshop (MLG-2010)

Results



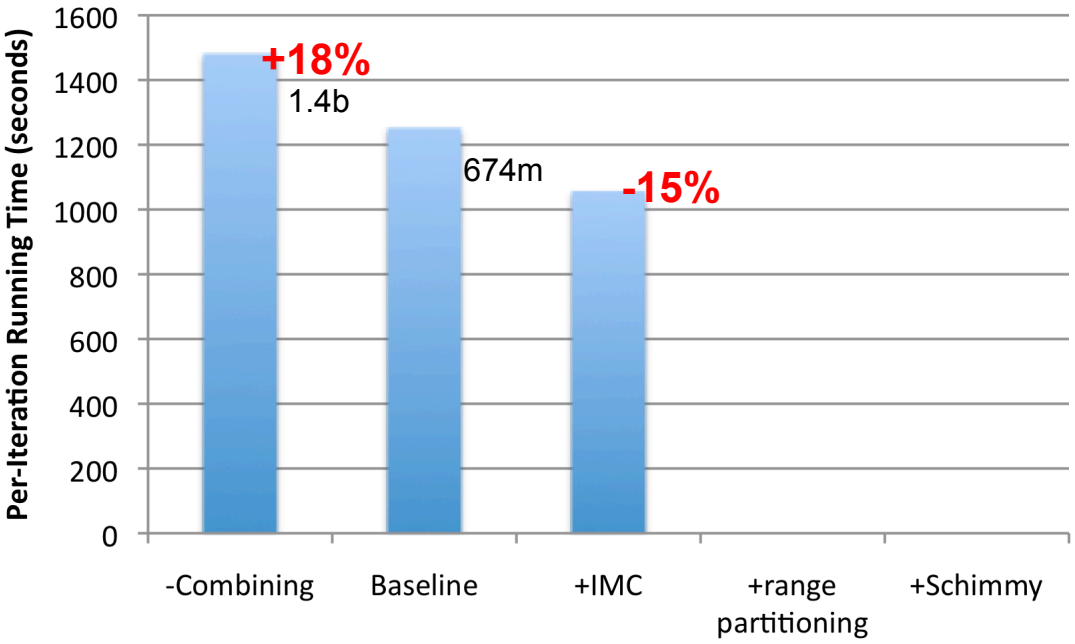
From: Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. Proceedings of the Eighth Workshop on Mining and Learning with Graphs Workshop (MLG-2010)

Results



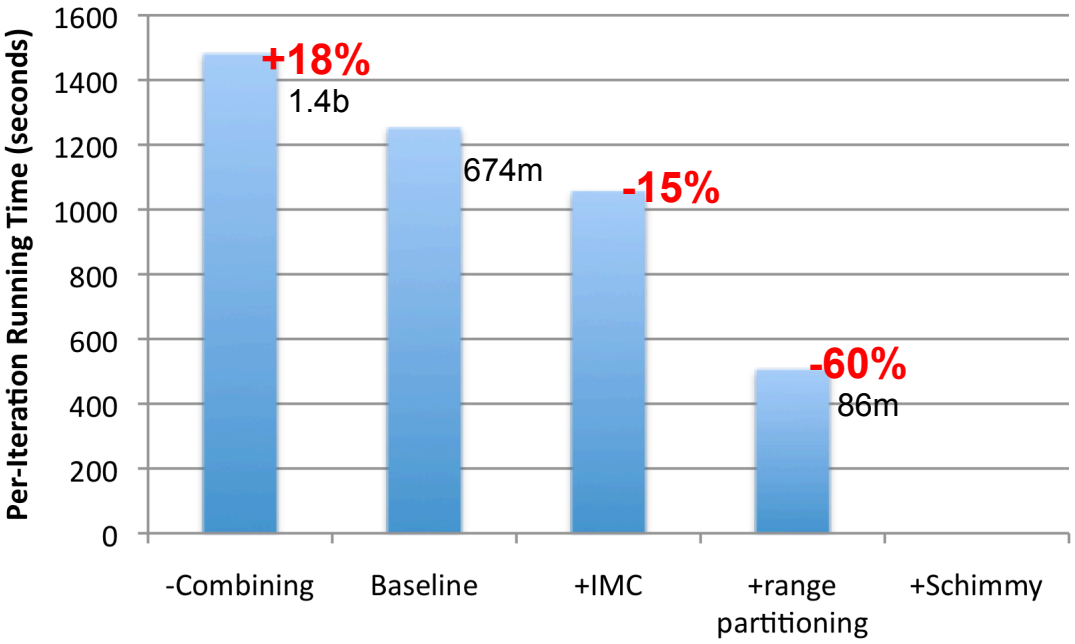
From: Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. Proceedings of the Eighth Workshop on Mining and Learning with Graphs Workshop (MLG-2010)

Results



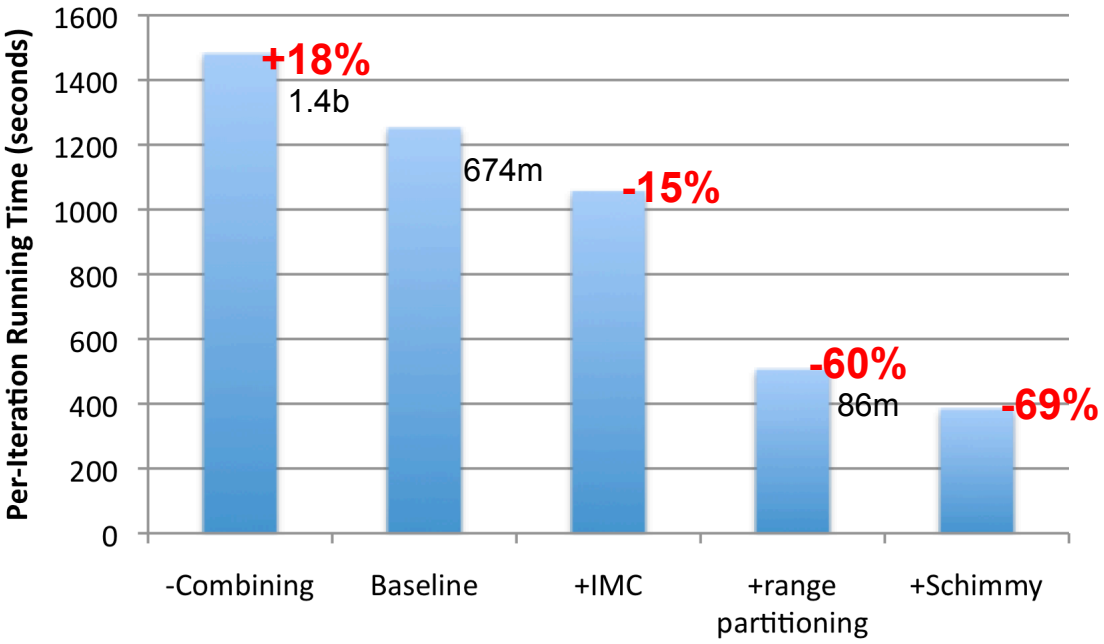
From: Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. Proceedings of the Eighth Workshop on Mining and Learning with Graphs Workshop (MLG-2010)

Results



From: Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. Proceedings of the Eighth Workshop on Mining and Learning with Graphs Workshop (MLG-2010)

Results



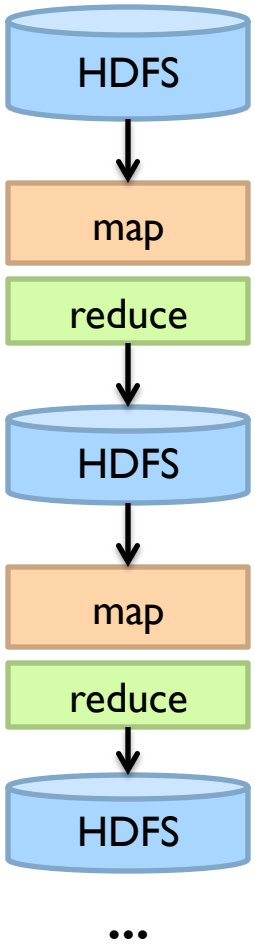
From: Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. Proceedings of the Eighth Workshop on Mining and Learning with Graphs Workshop (MLG-2010)

MapReduce Sucks

- Java verbosity
- Hadoop task startup time
- Stragglers
- Needless graph shuffling
- Checkpointing at each iteration

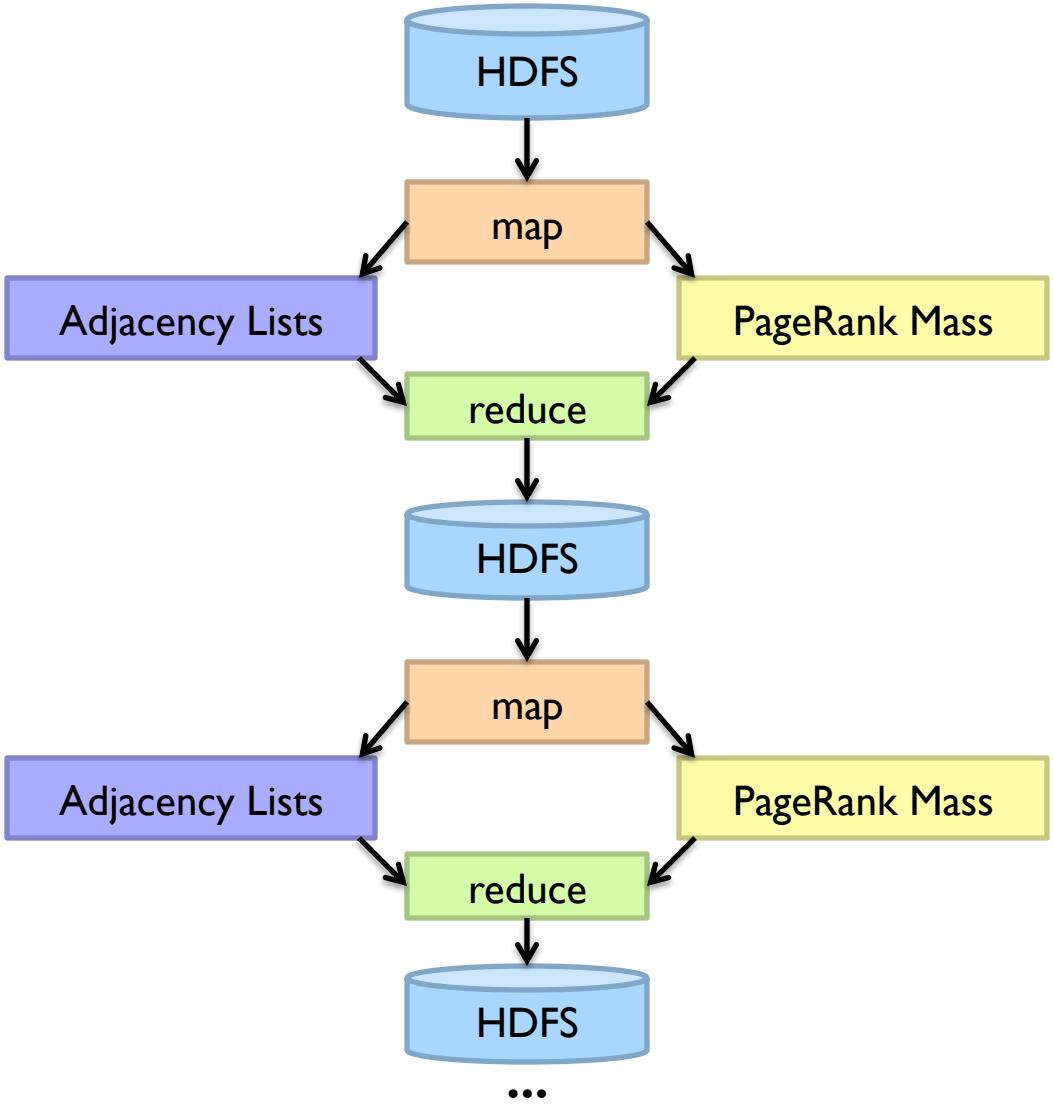
What have we fixed?

From MapReduce to Spark

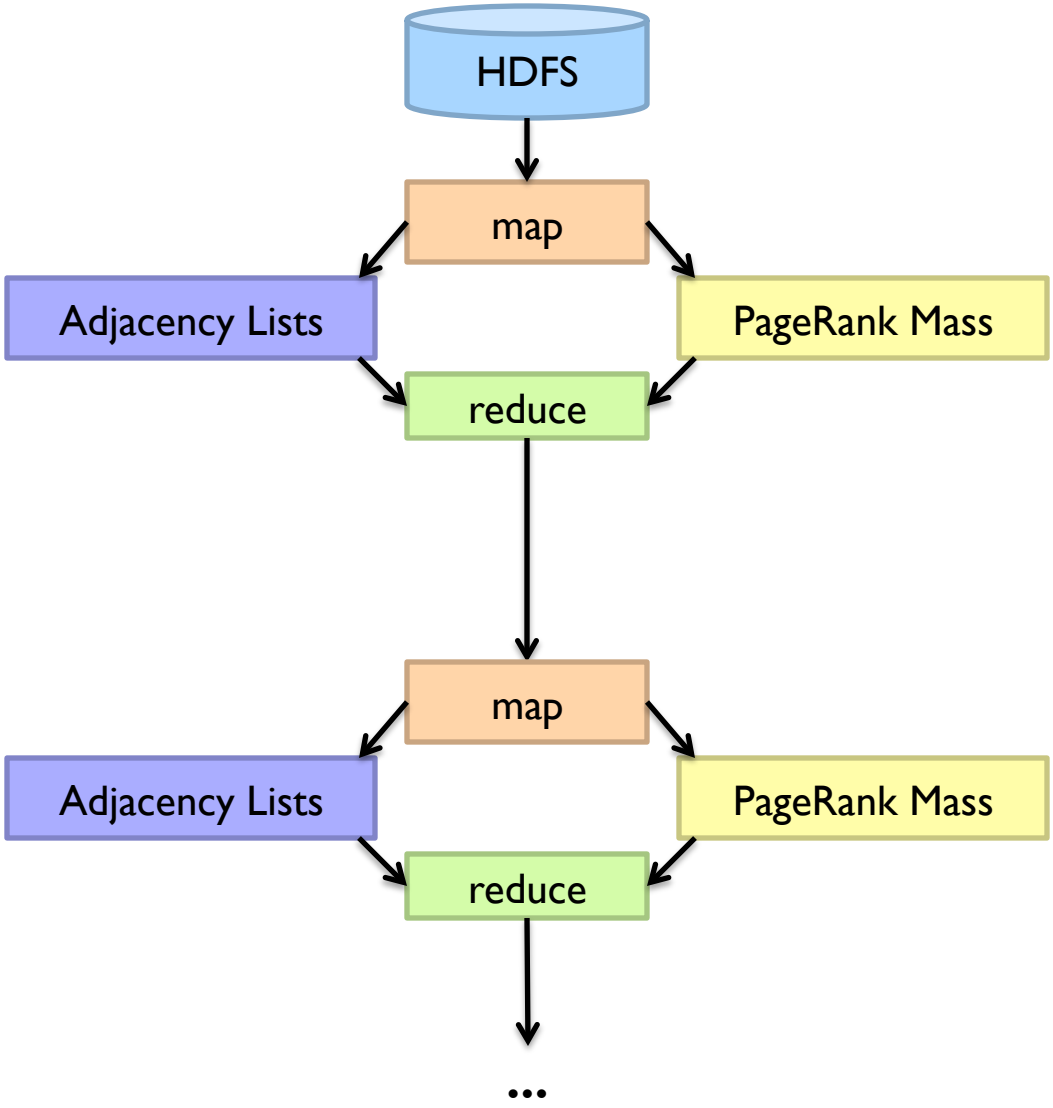


(omitting the second MapReduce job for simplicity; no handling of dangling links)

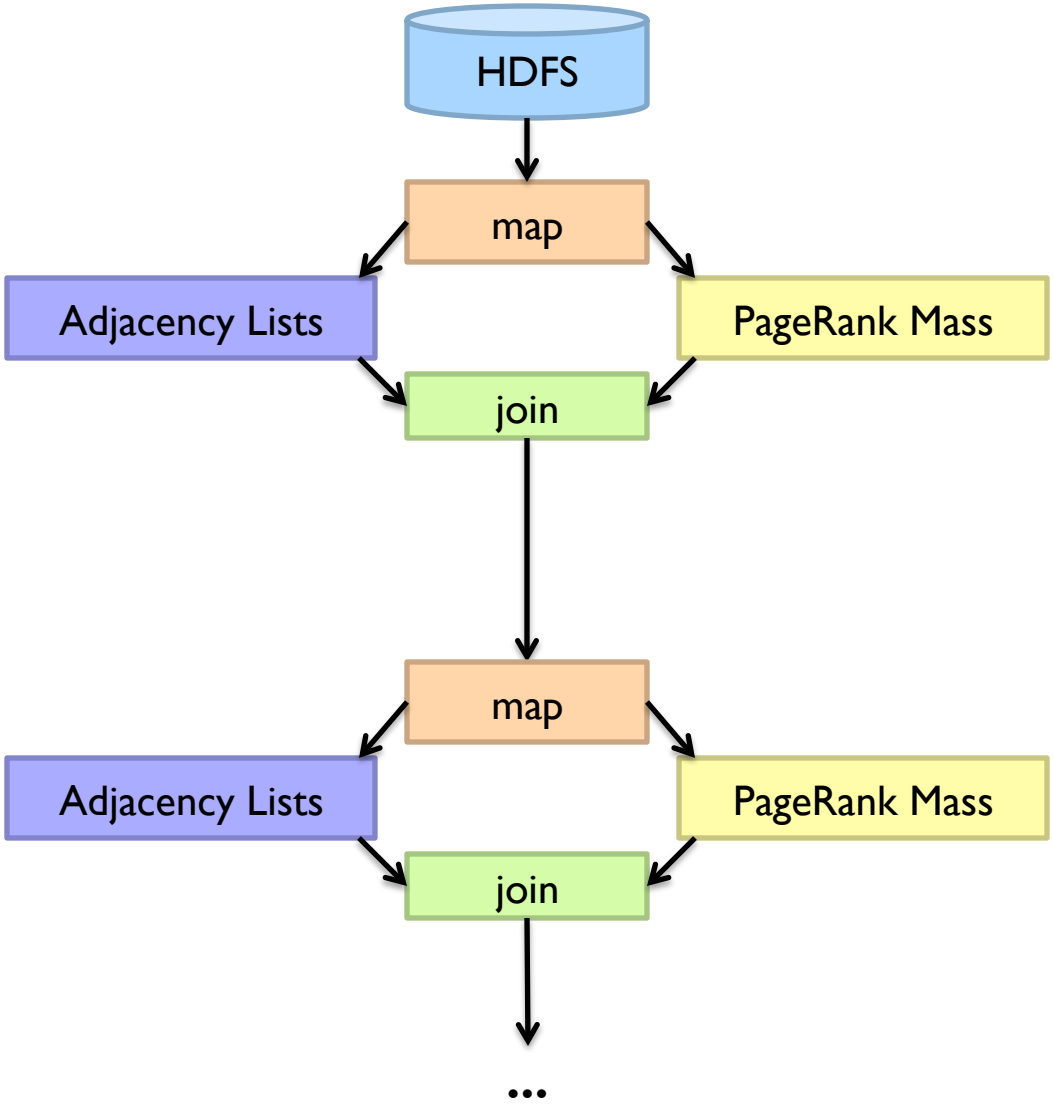
From MapReduce to Spark



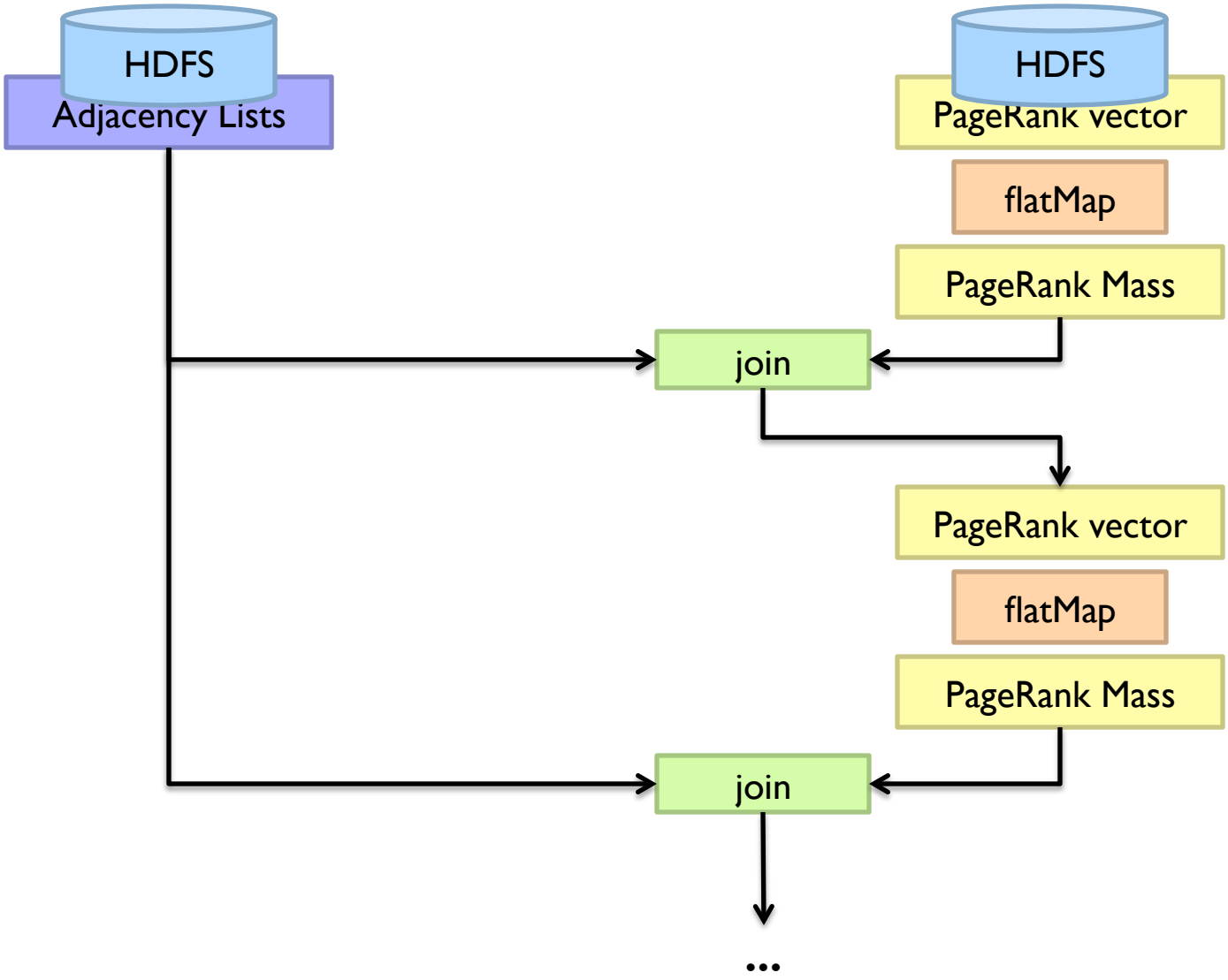
From MapReduce to Spark



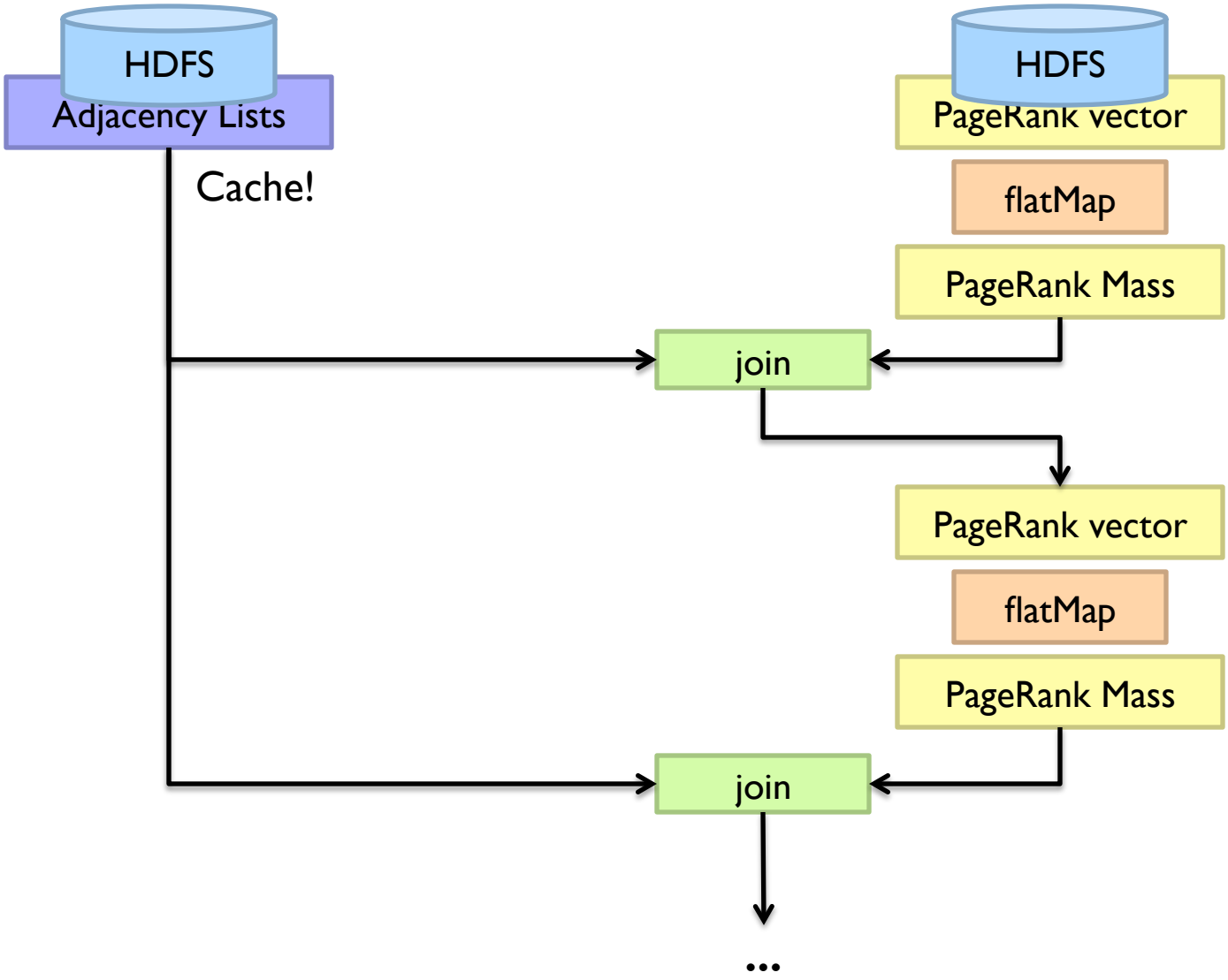
From MapReduce to Spark



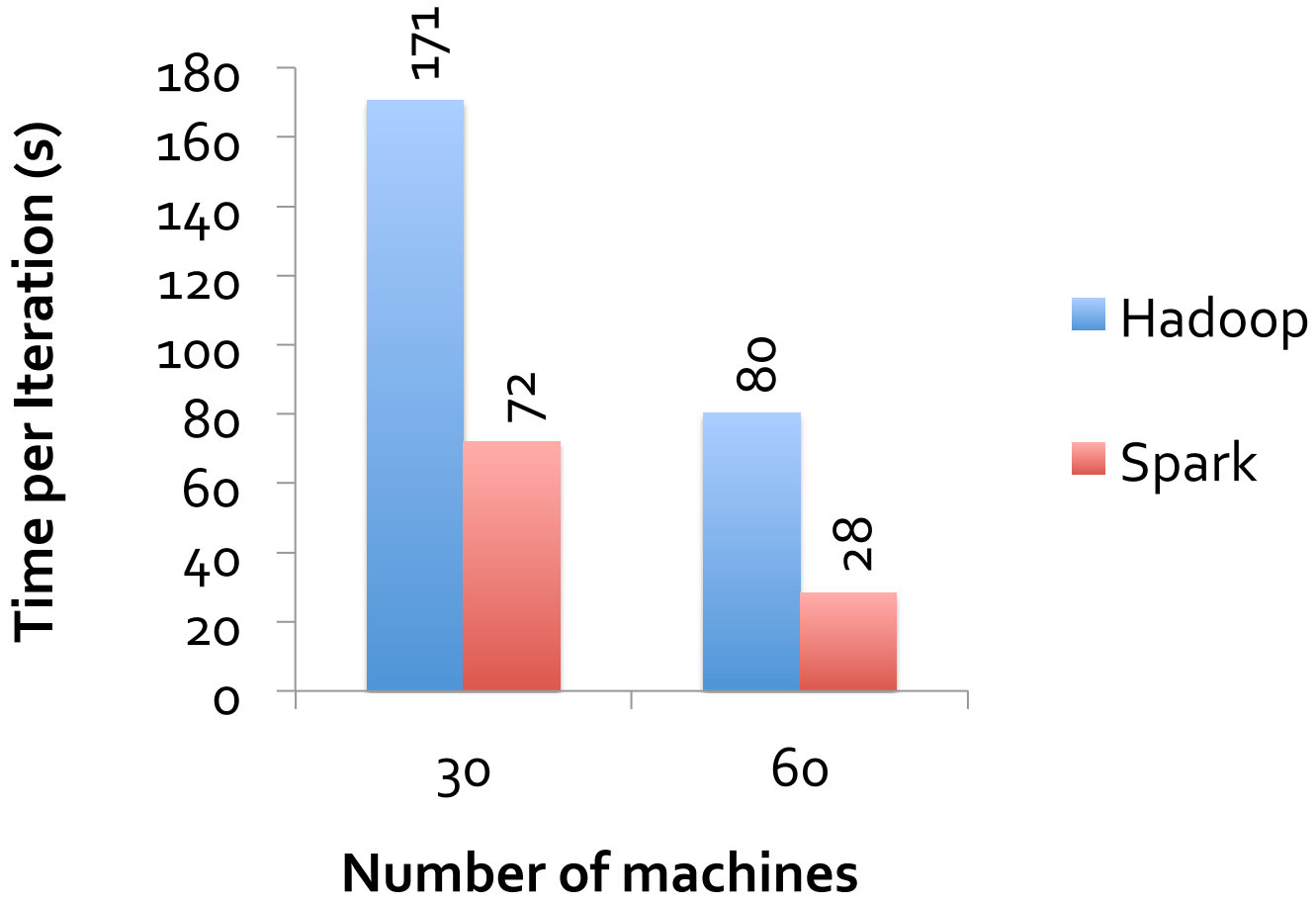
From MapReduce to Spark



From MapReduce to Spark



MapReduce vs. Spark



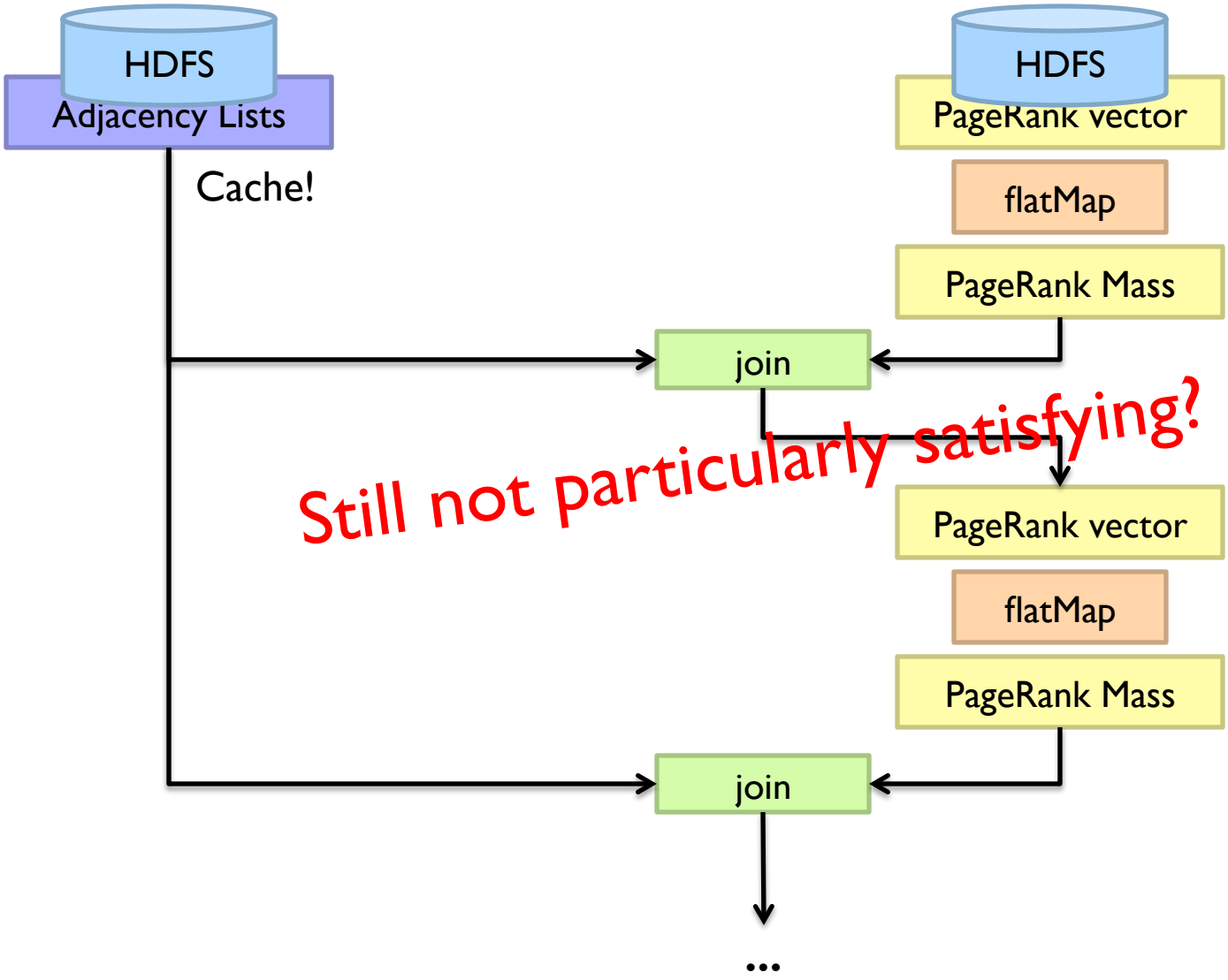
Source: http://ampcamp.berkeley.edu/wp-content/uploads/2012/06/matei-zaharia-part-2-amp-camp-2012-standalone-programs.pdf

Spark to the Rescue!

- Java verbosity
- Hadoop task startup time
- Stragglers
- Needless graph shuffling
- Checkpointing at each iteration

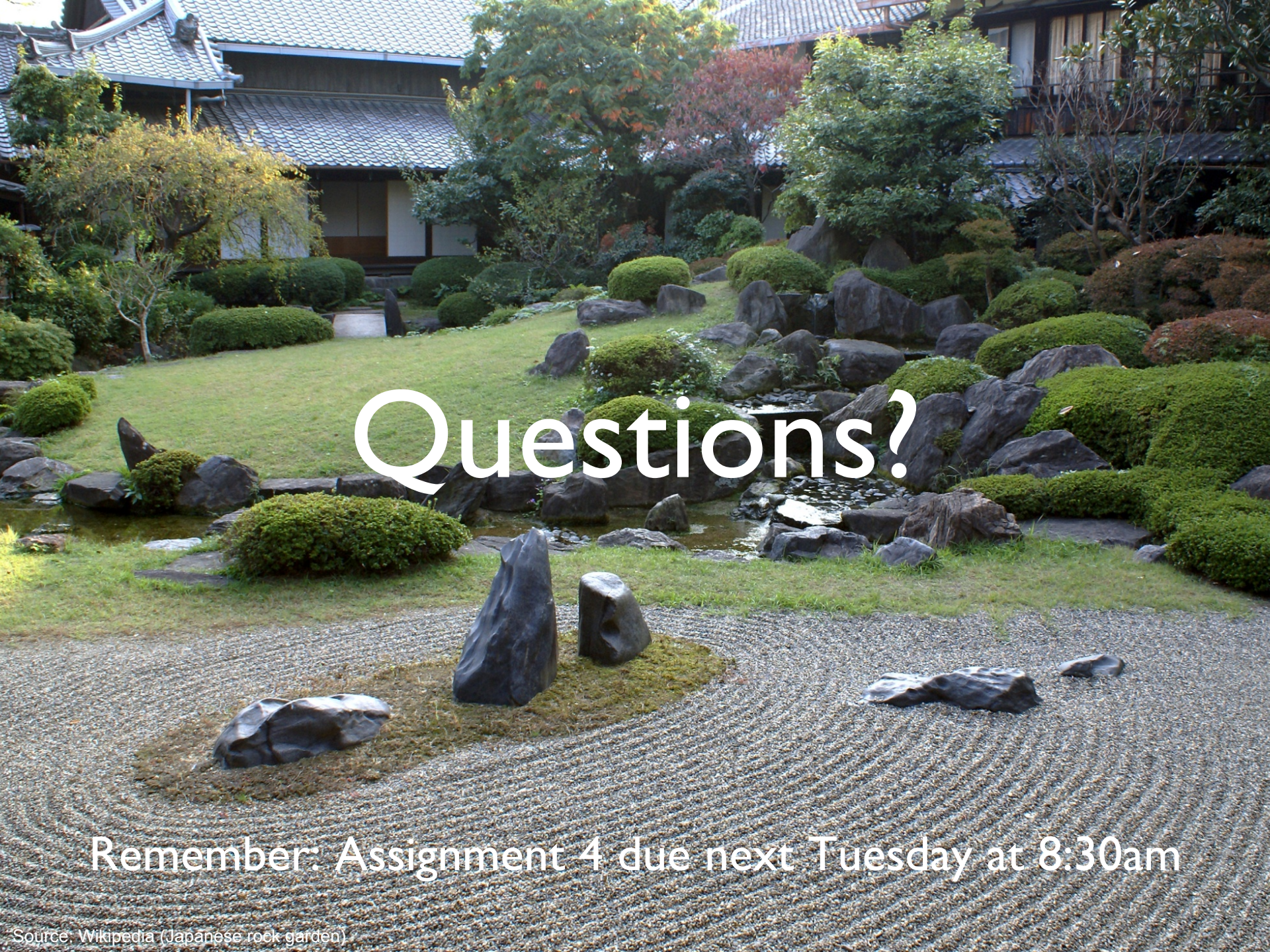
What have we fixed?

From MapReduce to Spark



Stay Tuned!

Source: https://www.flickr.com/photos/smuzz/4350039327/



Source: Wikipedia (Japanese rock garden)