Data-Intensive Distributed Computing

CS 451/651 431/631 (Winter 2018)

Part 4: Analyzing Graphs (2/2)

February 6, 2018

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

These slides are available at http://lintool.github.io/bigdata-2018w/

Parallel BFS in MapReduce

Data representation:

Key: node n

Value: d (distance from start), adjacency list

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

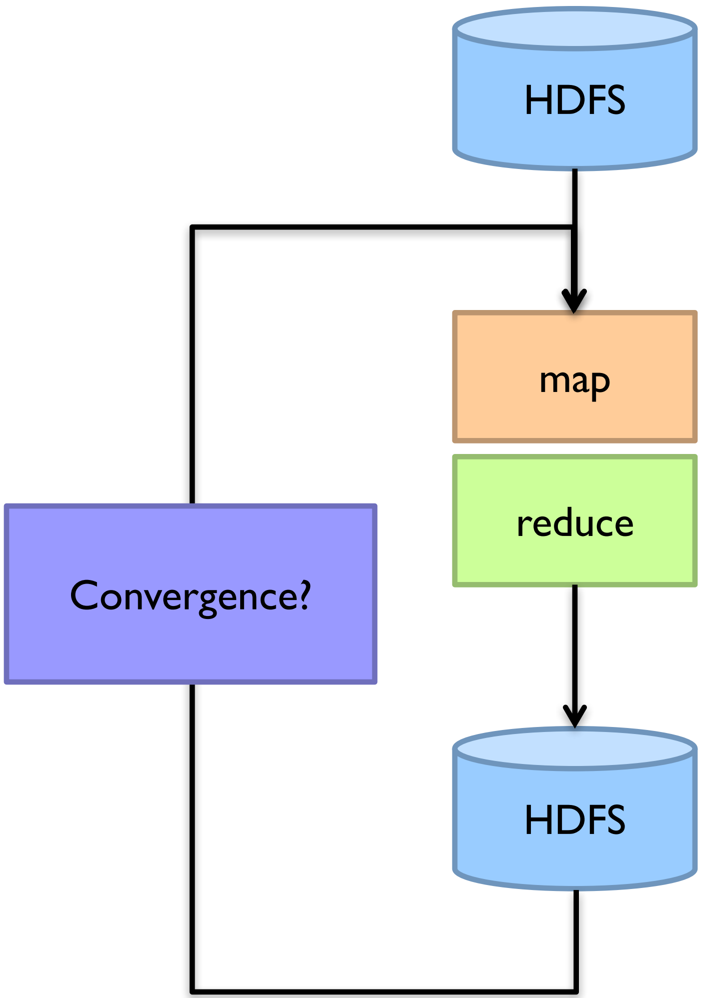
Remember to pass along the graph structure!

BFS Pseudo-Code

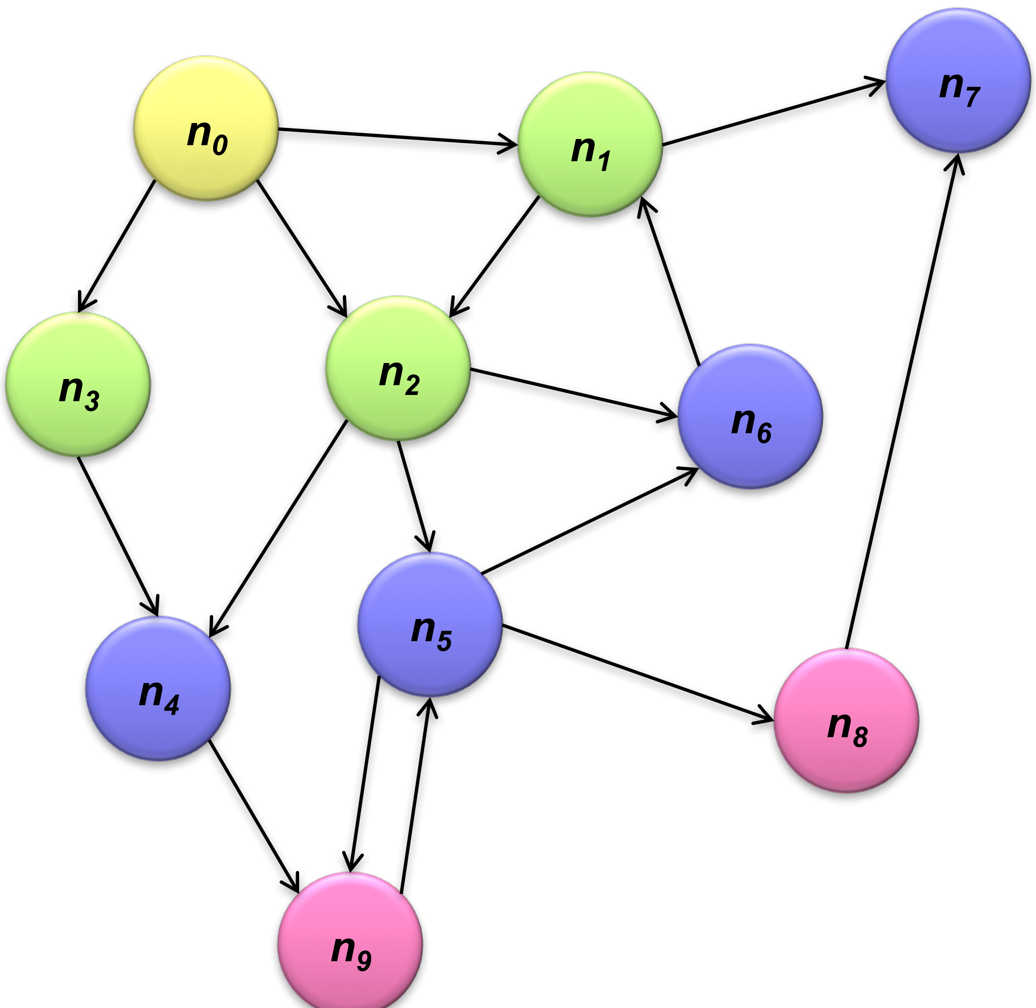
```
class Mapper {
  def map(id: Long, n: Node) = {
    emit(id, n)
    val d = n.distance
    emit(id, d)
    for (m <- n.adjacenyList) {
      emit(m, d+1)
    }
}

class Reducer {
  def reduce(id: Long, objects: Iterable[Object]) = {
    var min = infinity
    var n = null
    for (d <- objects) {
      if (isNode(d))    n = d
      else if d < min   min = d
    }
    n.distance = min
    emit(id, n)
  }
}
```

Implementation Practicalities



Visualizing Parallel BFS



Non-toy?

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 , … s_n }

Instead of emitting a single distance, emit an array of distances, wrt each source

Reducer selects minimum for each element in array

Does this scale?

Landmark Approach (aka sketches)

Select n seeds { s_0, s_1, … 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

Run multi-source parallel BFS in MapReduce!

Lots of details:

How to more tightly bound distances
How to select landmarks (random isn't the best…)

Graphs and MapReduce (and Spark)

A large class of graph algorithms involve:

Local computations at each node

Propagating results: “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

Use in web ranking

Correspondence to human intuition?

One of thousands of features used in web search

PageRank: Defined

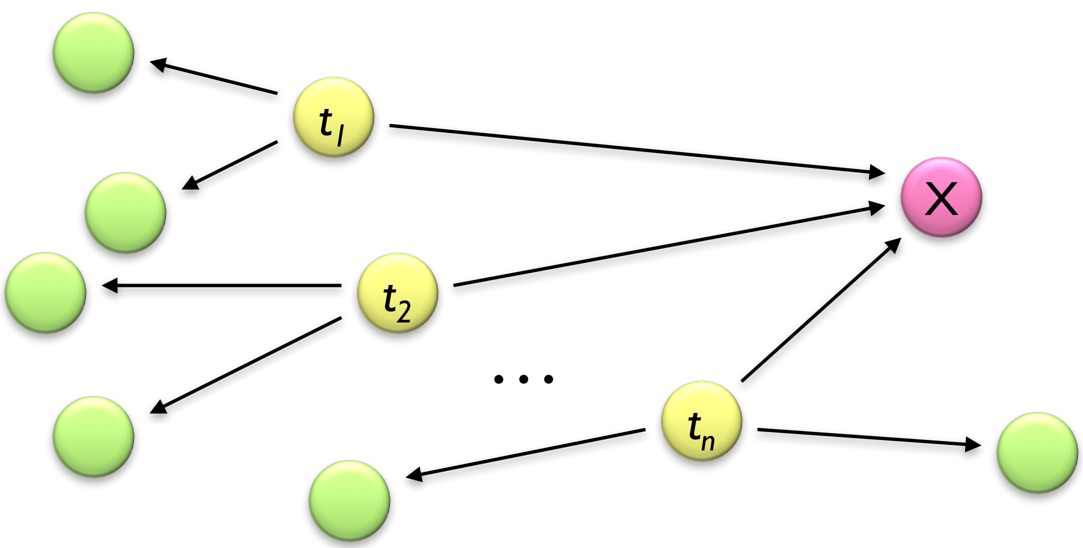
Given page *x* with inlinks $t_1 \ldots t_n$, where

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

Remember this?

A large class of graph algorithms involve:

Local computations at each node

Propagating results: “traversing” the graph

Sketch of algorithm:

Start with seed PR_i values

Each page distributes PR_i mass to all pages it links to

Each target page adds up mass from in-bound links to compute PR_{i+1}

Iterate until values converge

Simplified PageRank

First, tackle the simple case:

No random jump factor

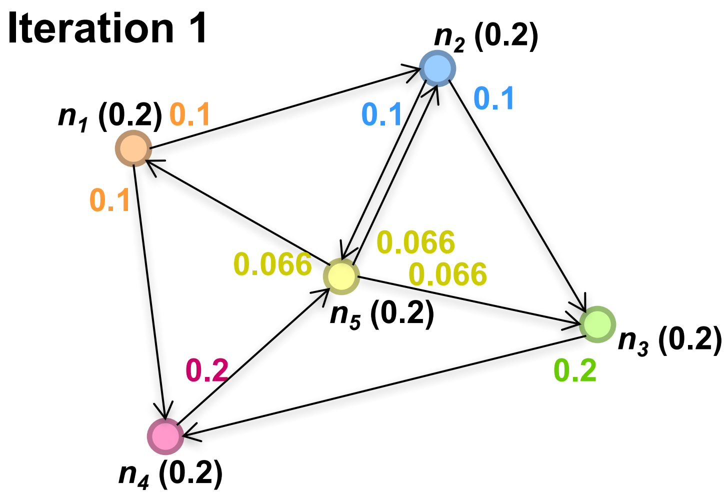
No dangling nodes

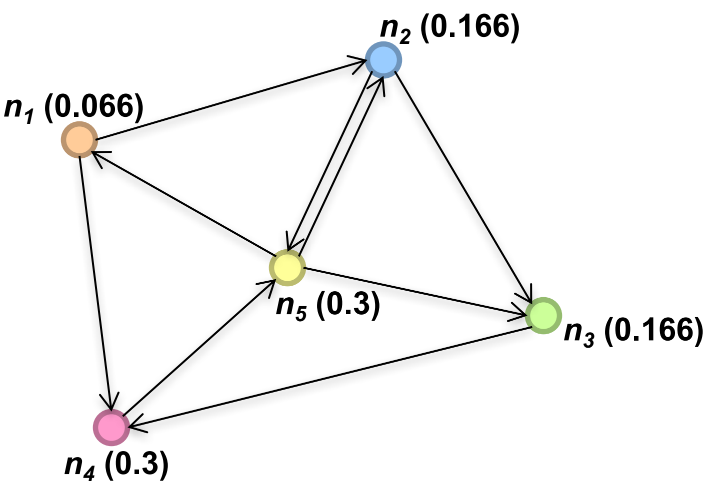
Then, factor in these complexities…

Why do we need the random jump?

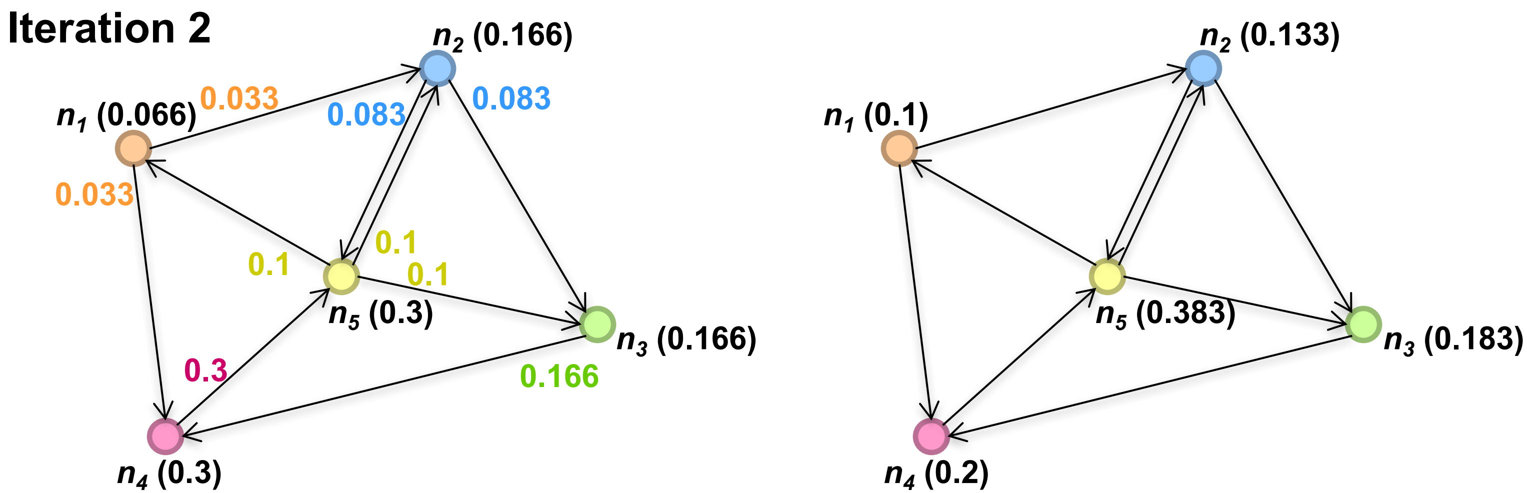
Where do dangling nodes come from?

Sample PageRank Iteration (1)

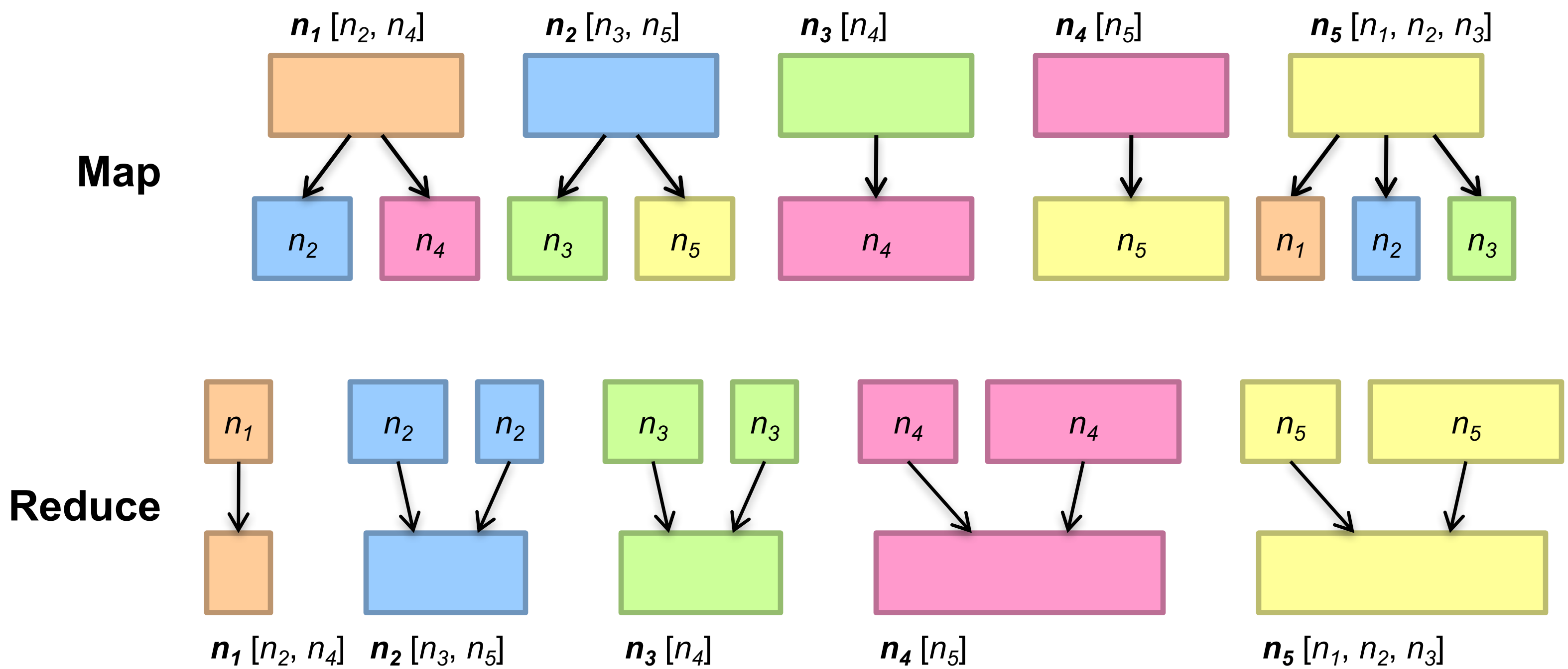




Sample PageRank Iteration (2)



PageRank in MapReduce



PageRank Pseudo-Code

```
class Mapper {
  def map(id: Long, n: Node) = {
    emit(id, n)
    p = n.PageRank / n.adjacenyList.length
    for (m <- n.adjacenyList) {
      emit(m, p)
    }
}

class Reducer {
  def reduce(id: Long, objects: Iterable[Object]) = {
    var s = 0
    var n = null
    for (p <- objects) {
      if (isNode(p))    n = p
      else              s += p
    }
    n.PageRank = s
    emit(id, n)
  }
}
```

PageRank vs. BFS

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

A large class of graph algorithms involve:

Local computations at each node

Propagating results: “traversing” the graph

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

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

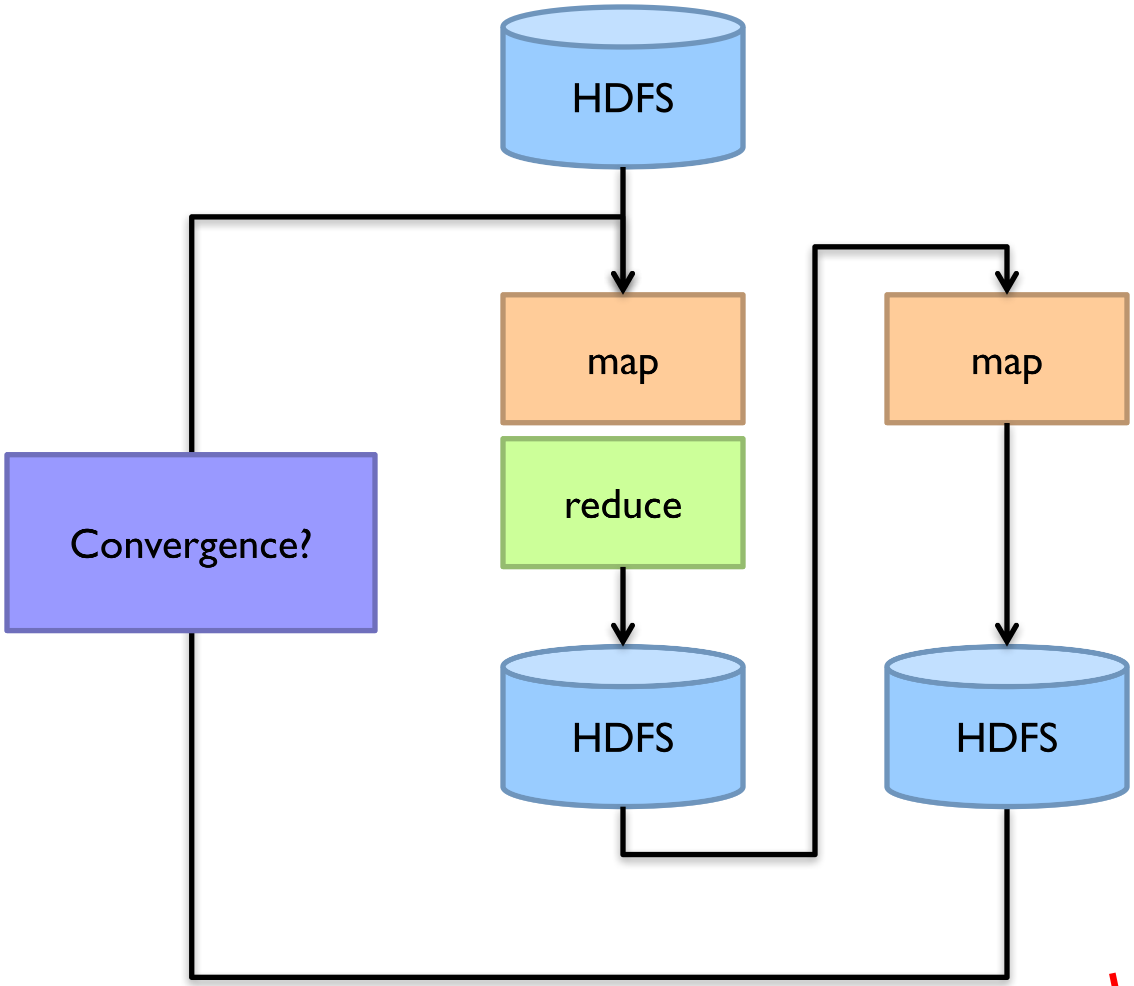
p is PageRank value from before, *p*' is updated PageRank value

N is the number of nodes in the graph

m is the missing PageRank mass

One final optimization: fold into a single MR job

Implementation Practicalities



What's the optimization?

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

…

Log Probs

PageRank values are *really* small…

Solution?

Product of probabilities = Addition of log probs

Addition of probabilities?

$$a \oplus b = \begin{cases} b + \log(1 + e^{a-b}) & a < b \\ a + \log(1 + e^{b-a}) & a \geq b \end{cases}$$

More Implementation Practicalities

How do you even extract the webgraph?

Lots of details…

Beyond PageRank

Variations of PageRank

Weighted edges

Personalized PageRank

Variants on graph random walks

Hubs and authorities (HITS)

SALSA

Applications

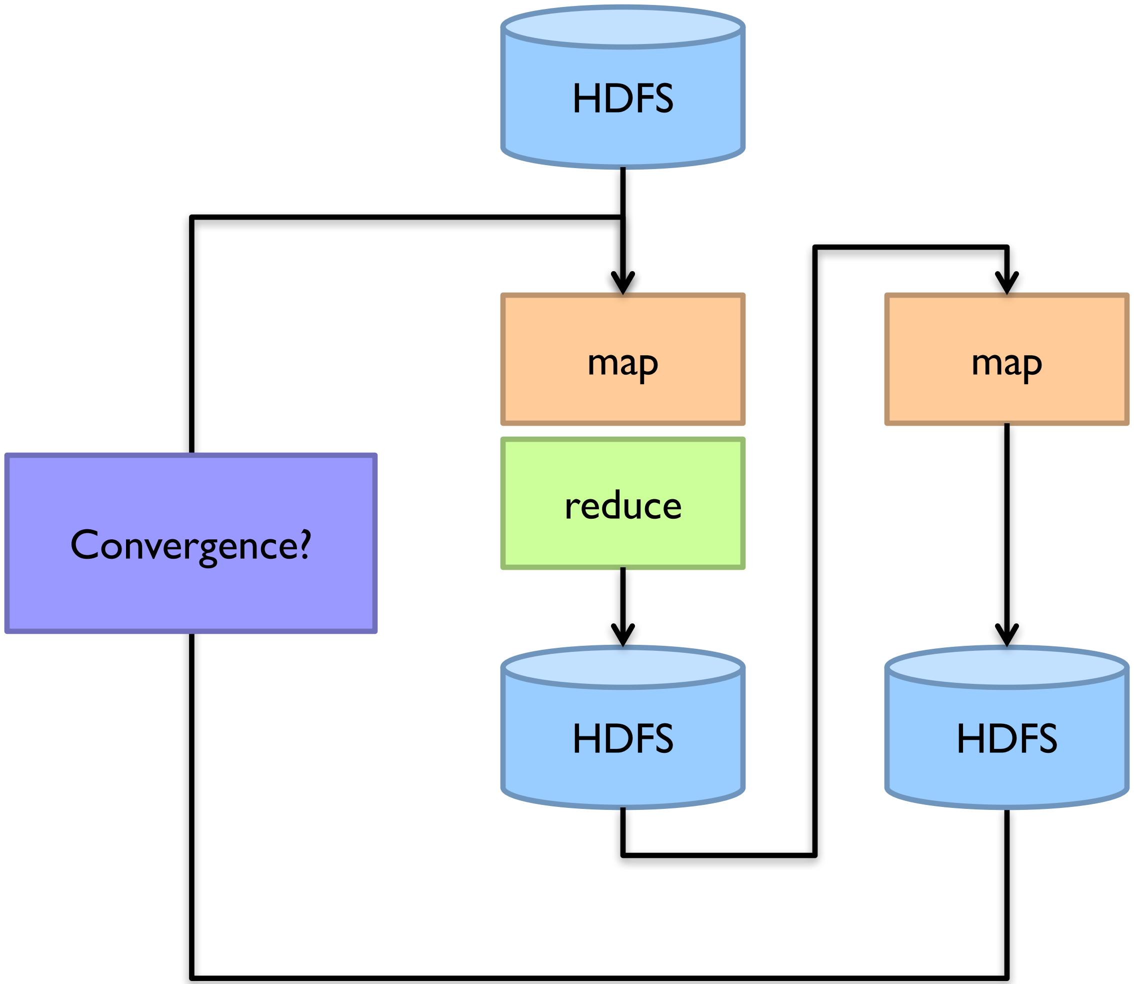
Static prior for web ranking

Identification of "special nodes" in a network

Link recommendation

Additional feature in any machine learning problem

Implementation Practicalities



MapReduce Sucks

Java verbosity

Hadoop task startup time

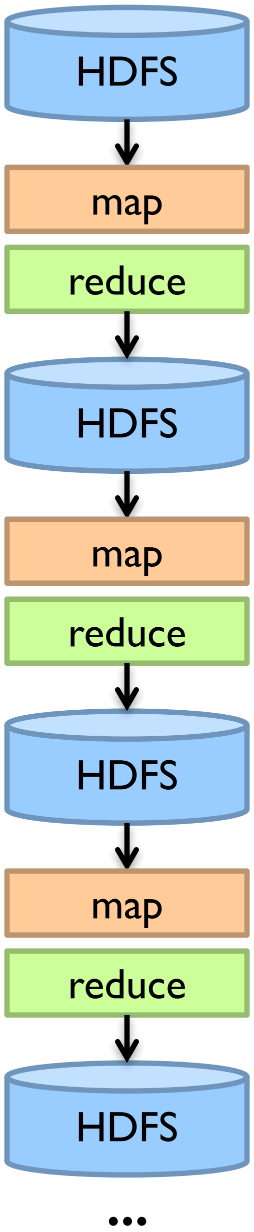
Stragglers

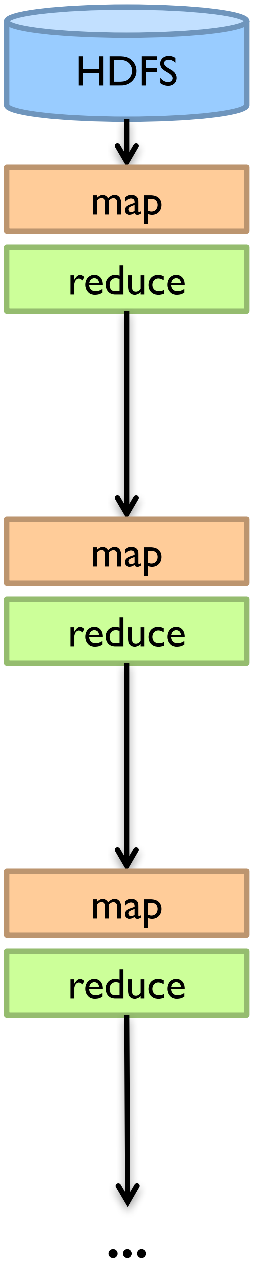
Needless graph shuffling

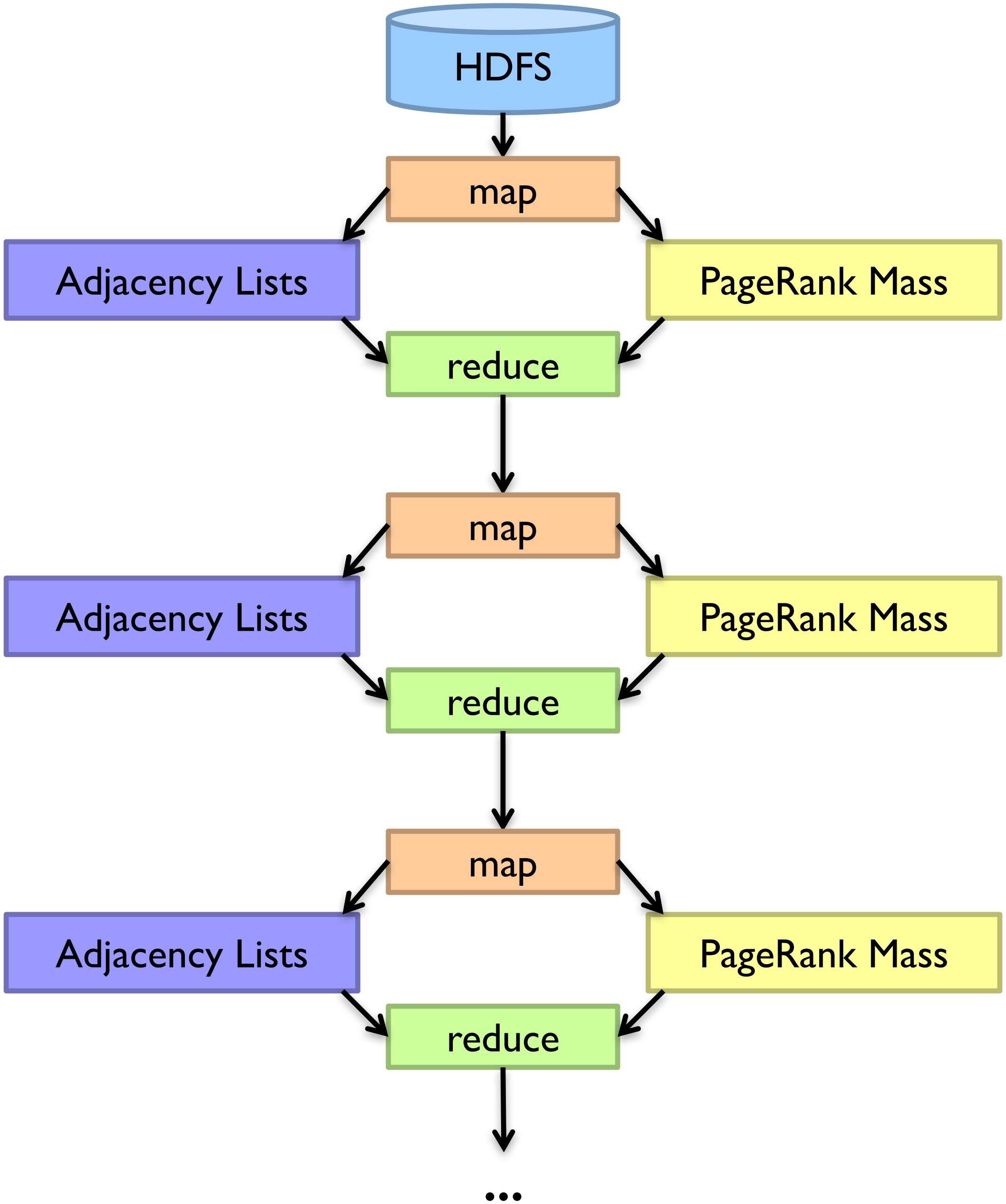
Checkpointing at each iteration

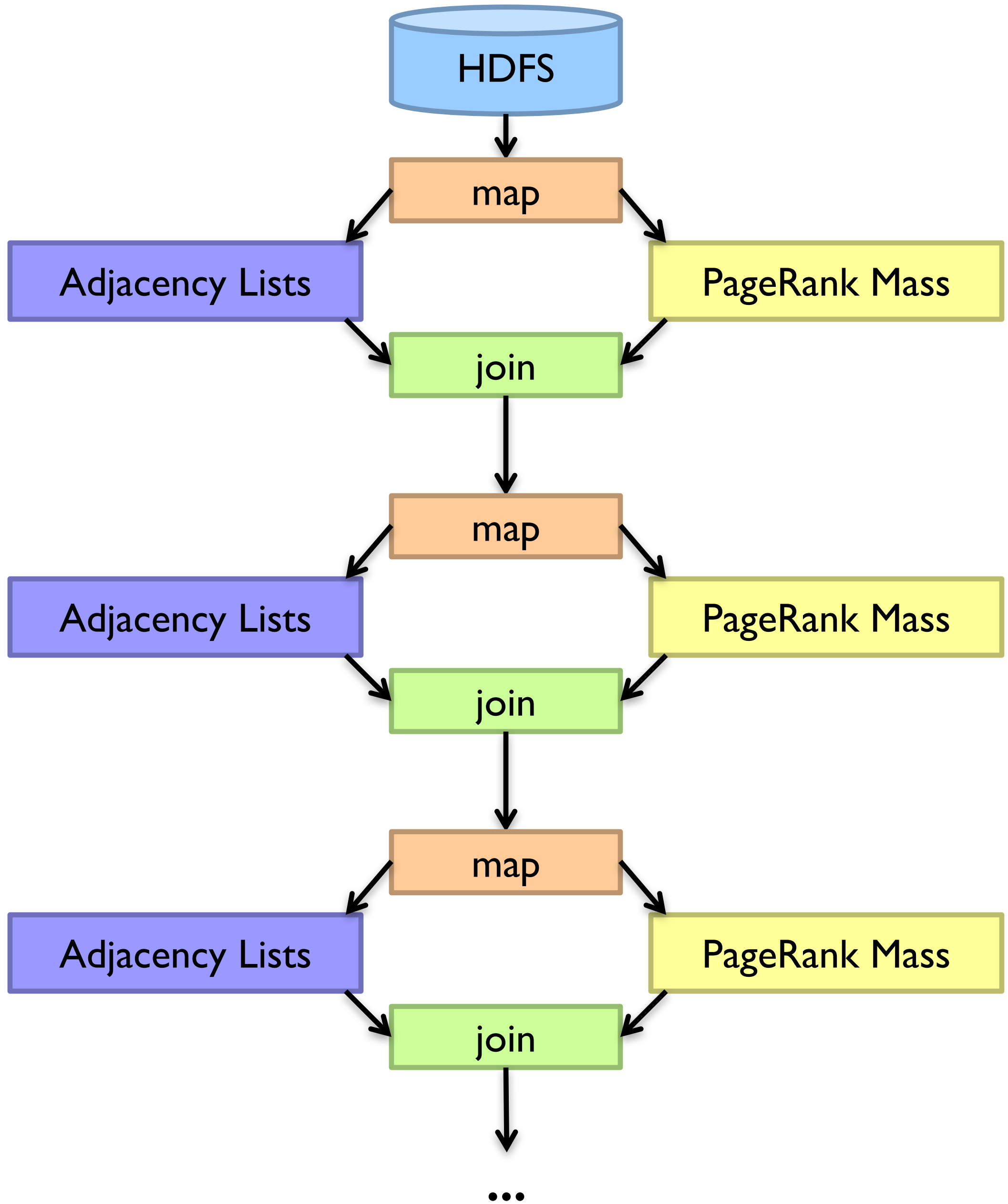
Spark to the rescue?

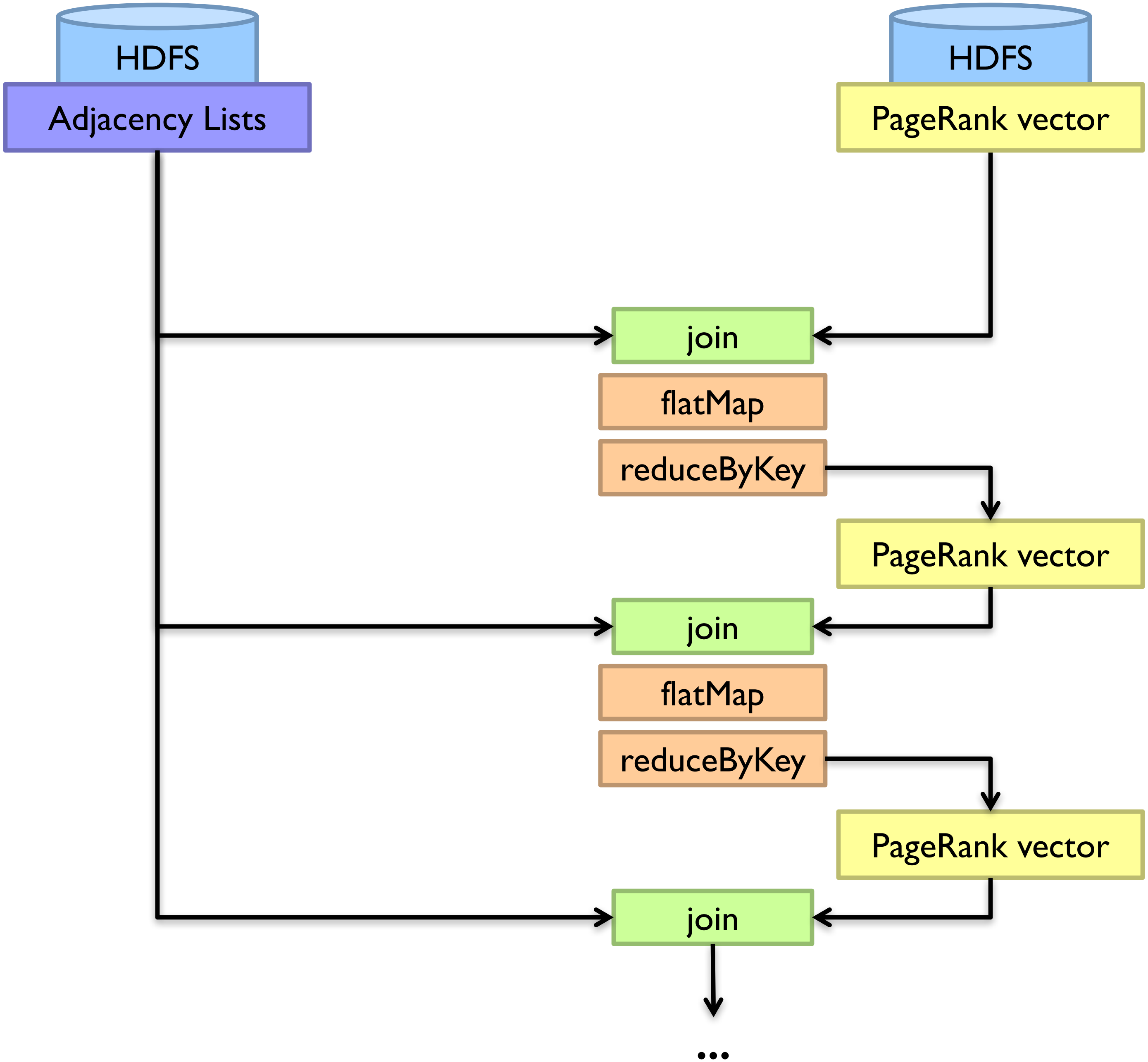
Let's Spark!



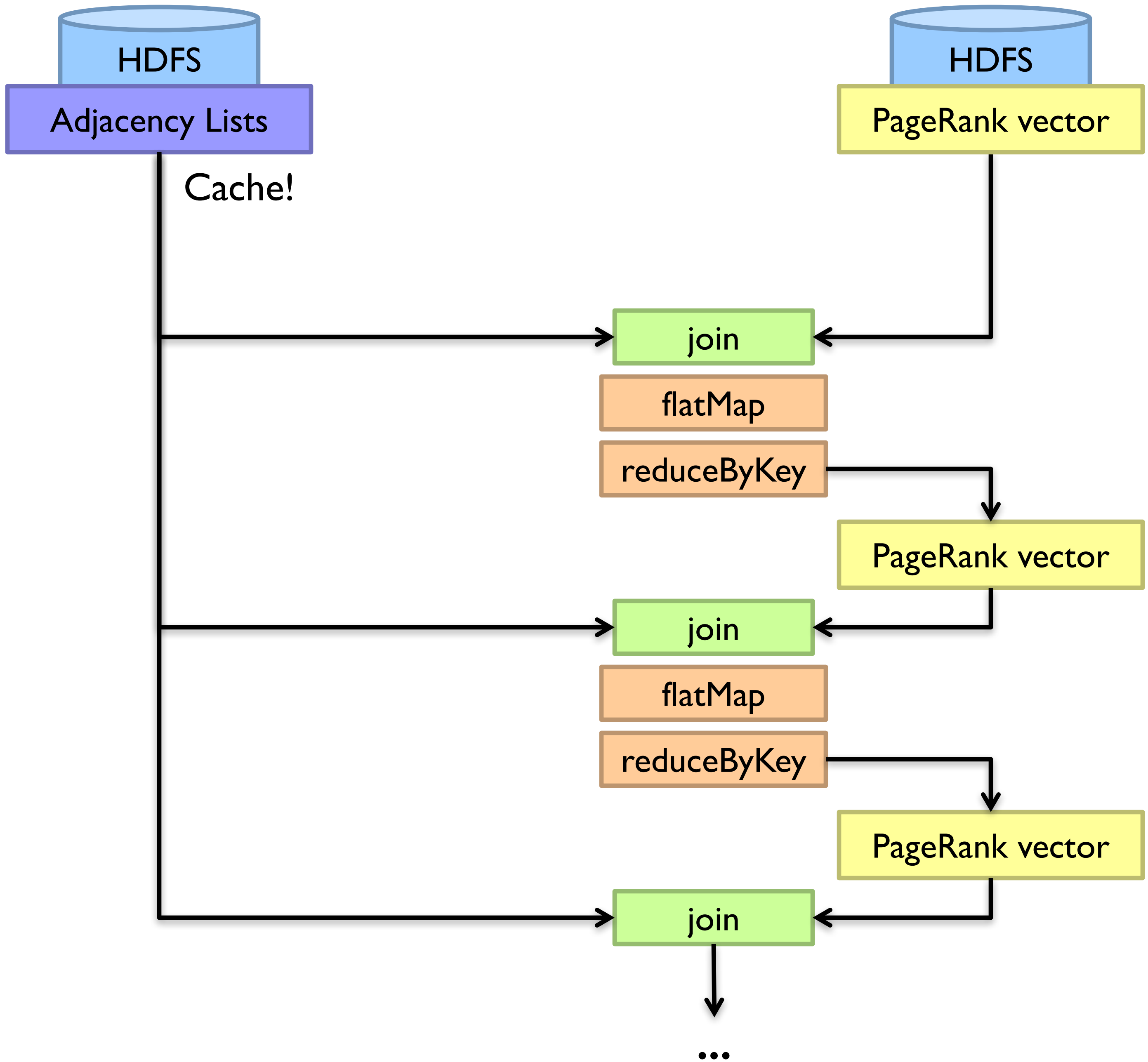

HDFS
map
reduce
map
reduce
map
reduce
...


HDFS
map
Adjacency Lists
PageRank Mass
reduce
map
Adjacency Lists
PageRank Mass
reduce
map
Adjacency Lists
PageRank Mass
reduce
…


HDFS
map
Adjacency Lists
PageRank Mass
join
map
Adjacency Lists
PageRank Mass
join
map
Adjacency Lists
PageRank Mass
join
…

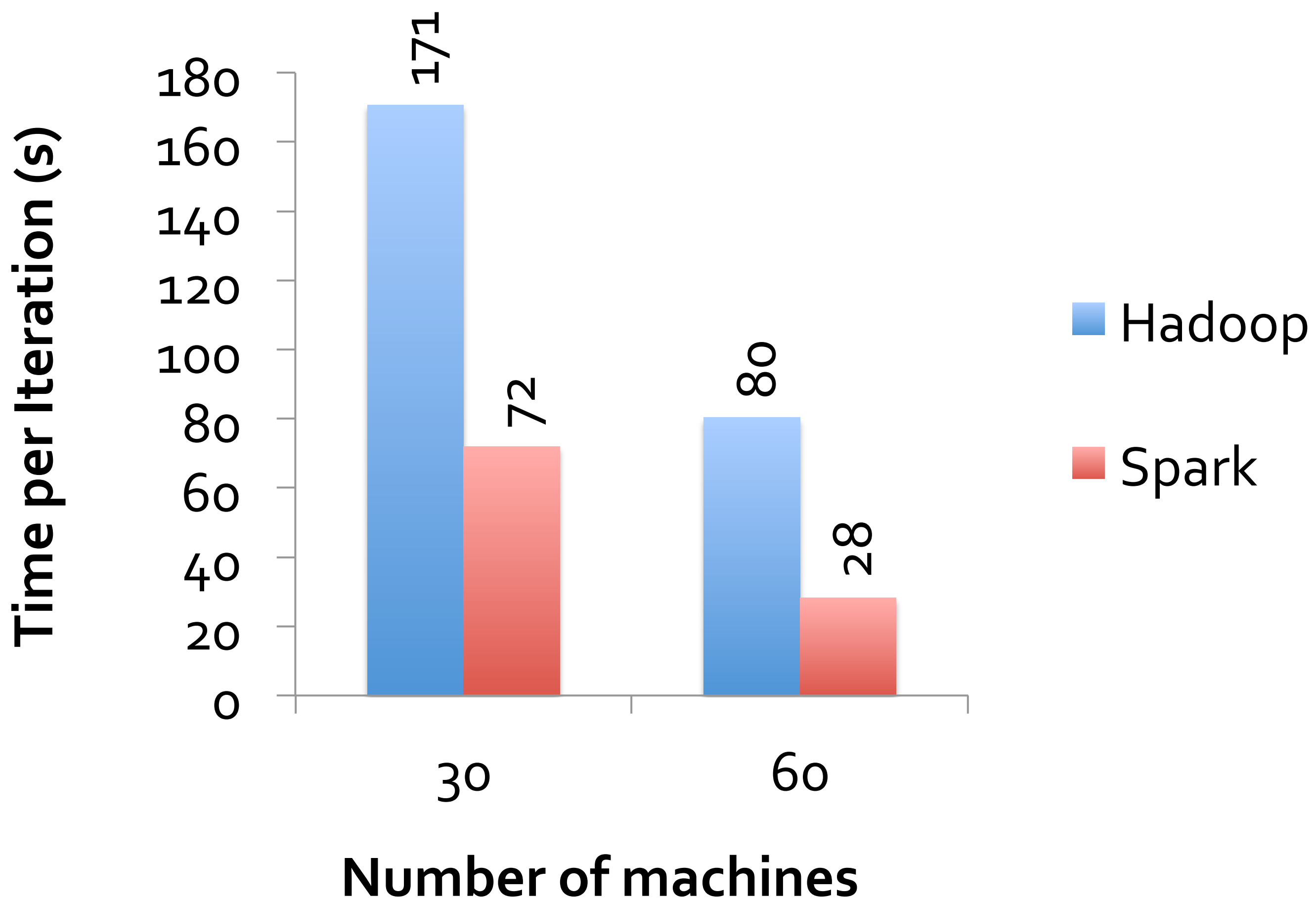


HDFS
Adjacency Lists
HDFS
PageRank vector
join
flatMap
reduceByKey
PageRank vector
join
flatMap
reduceByKey
PageRank vector
join
…



HDFS
Adjacency Lists
HDFS
PageRank vector
Cache!
join
flatMap
reduceByKey
PageRank vector
join
flatMap
reduceByKey
PageRank vector
join
…

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

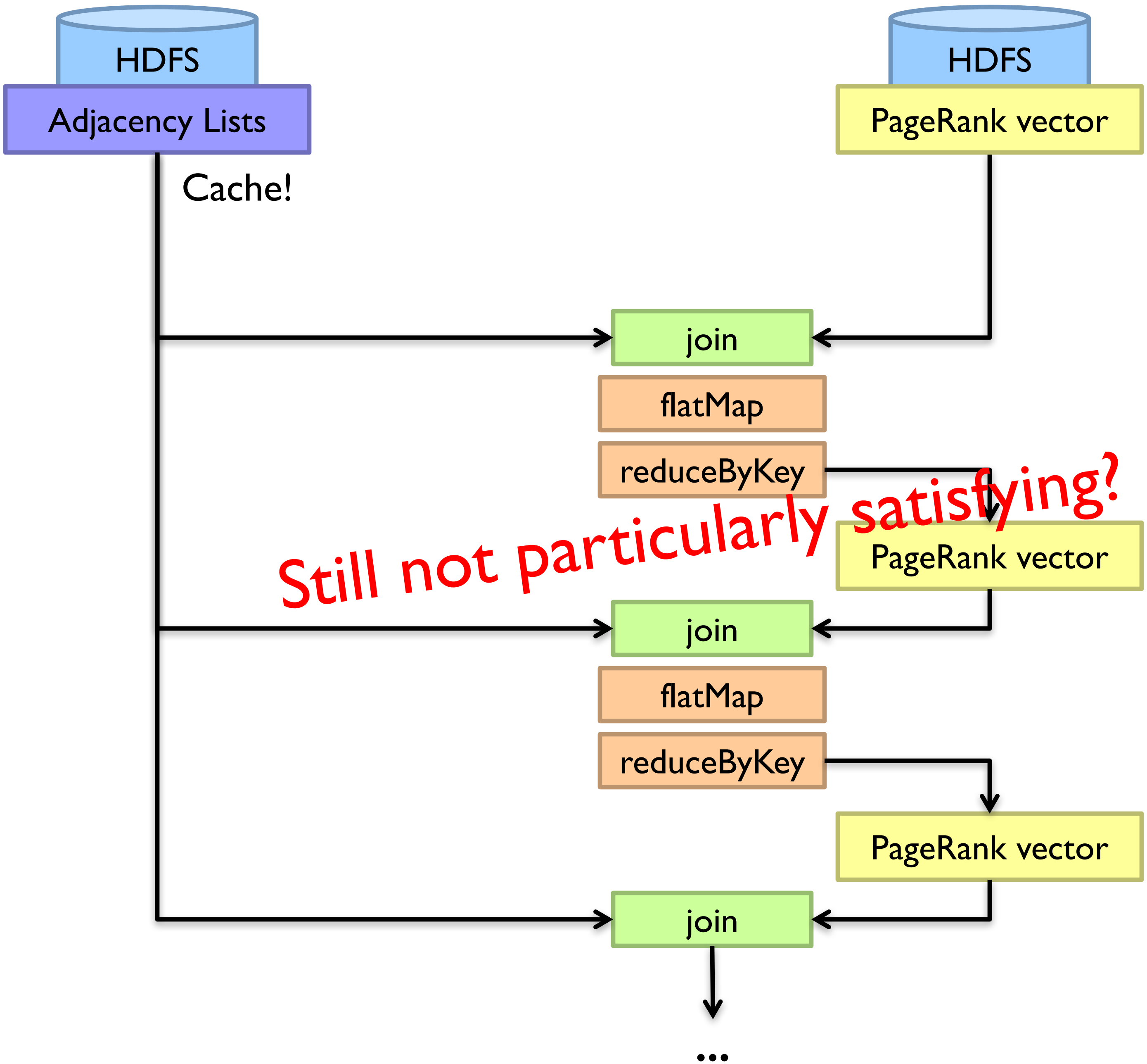
Hadoop task startup time

Stragglers

Needless graph shuffling

Checkpointing at each iteration

What have we fixed?


HDFS
Adjacency Lists
Cache!
HDFS
PageRank vector
join
flatMap
reduceByKey
PageRank vector
Still not particularly satisfying?
join
flatMap
reduceByKey
PageRank vector
join
…

Stay Tuned!

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

Source: Wikipedia (Japanese rock garden)