Data-Intensive Distributed Computing
CS 451/651 (Fall 2025)

UNIVERSITY OF WATERLOO

Graph Processing
(v1.00)

Week 12: November 20, 2025

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

These slides are available at http://lintool.github.io/bigdata-2025f/

Key Questions

What are alternative approaches to representing graphs?

Why are graph algorithms challenging in MapReduce/Spark?

What is the general structure of graph traversals in MapReduce/Spark?

Why MapReduce/Spark?

Graphs and MapReduce (and Spark)

A large class of graph algorithms involve:

Local computations at each node

Propagating results: "traversing" the graph

Key questions:

How do you represent graph data in MapReduce (and Spark)?

How do you traverse a graph in MapReduce (and Spark)?

Single-Source Shortest Path (SSSP)

Parallel BFS

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:

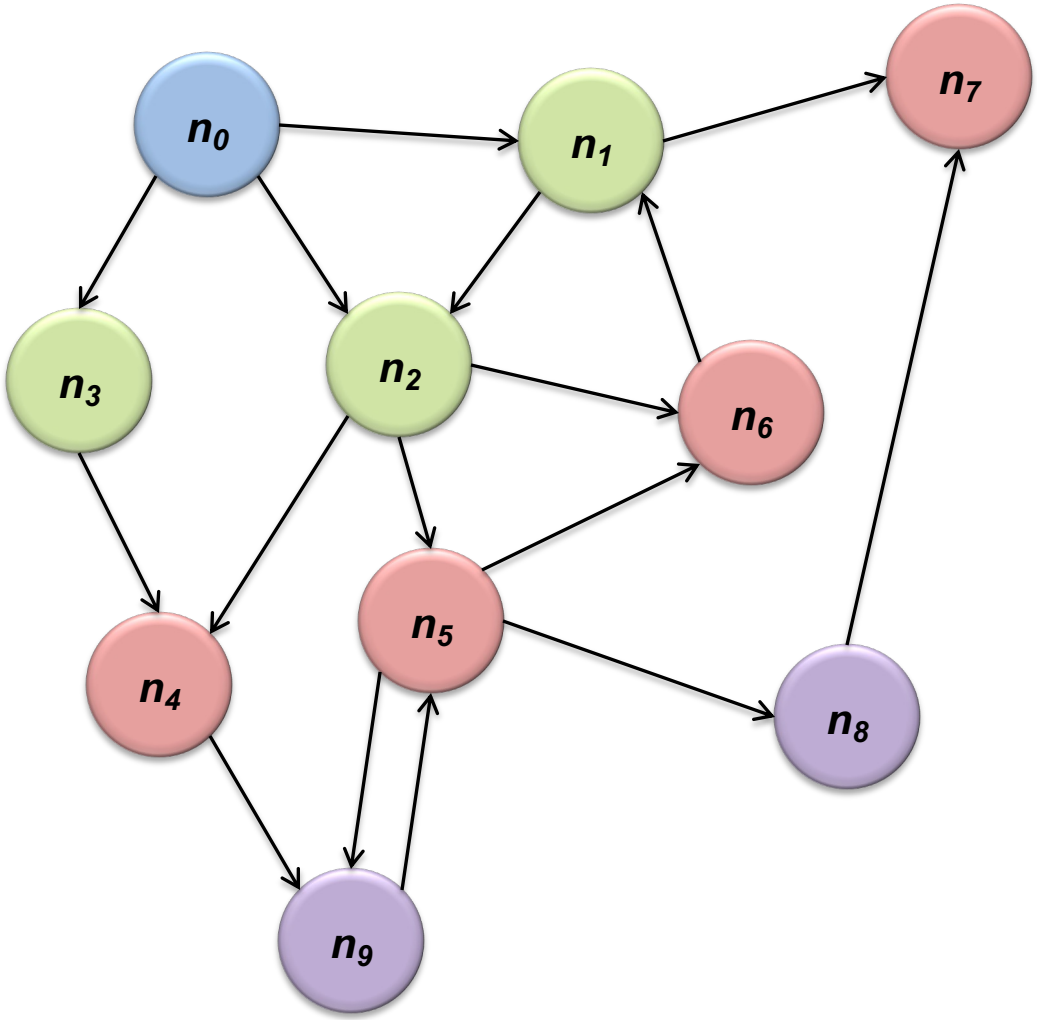
Groups distances by reachable nodes

Reducer:

Selects minimum distance path for each reachable node

Additional bookkeeping needed to keep track of actual path

Visualizing Parallel BFS



Comparison to Dijkstra

Dijkstra's algorithm is more efficient

At each step, only pursues edges from minimum-cost path inside frontier

MapReduce explores all paths in parallel

Lots of "waste"

Useful work is only done at the "frontier"

Why can't we do better using MapReduce?

Multiple-Source Shortest Path (MSSP)

Multiple-Source Parallel BFS

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, wrt each source

Reducer selects minimum for each element in array

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

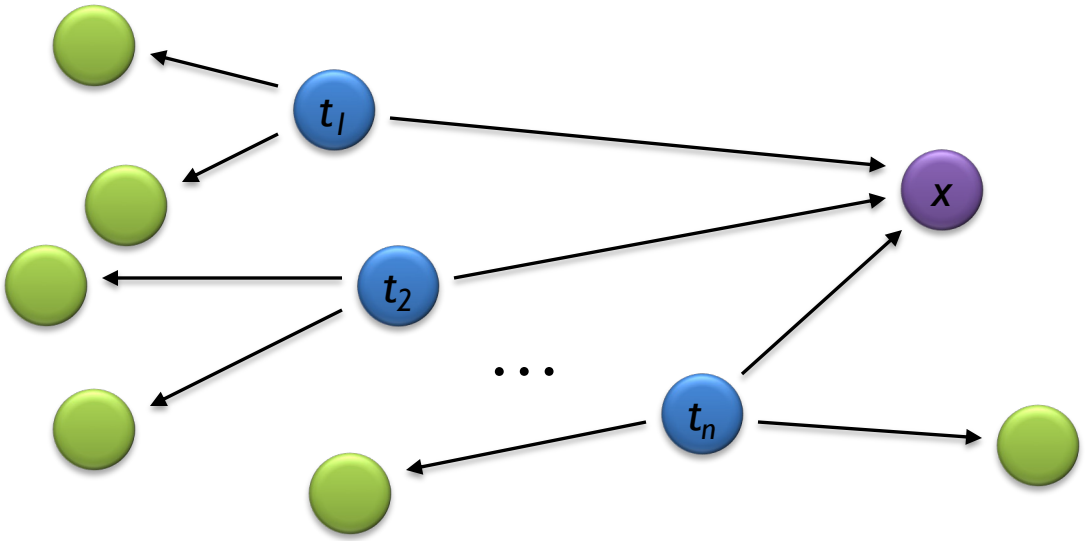
Given page *x* with inlinks $t_1 \ldots t_n$, where

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

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

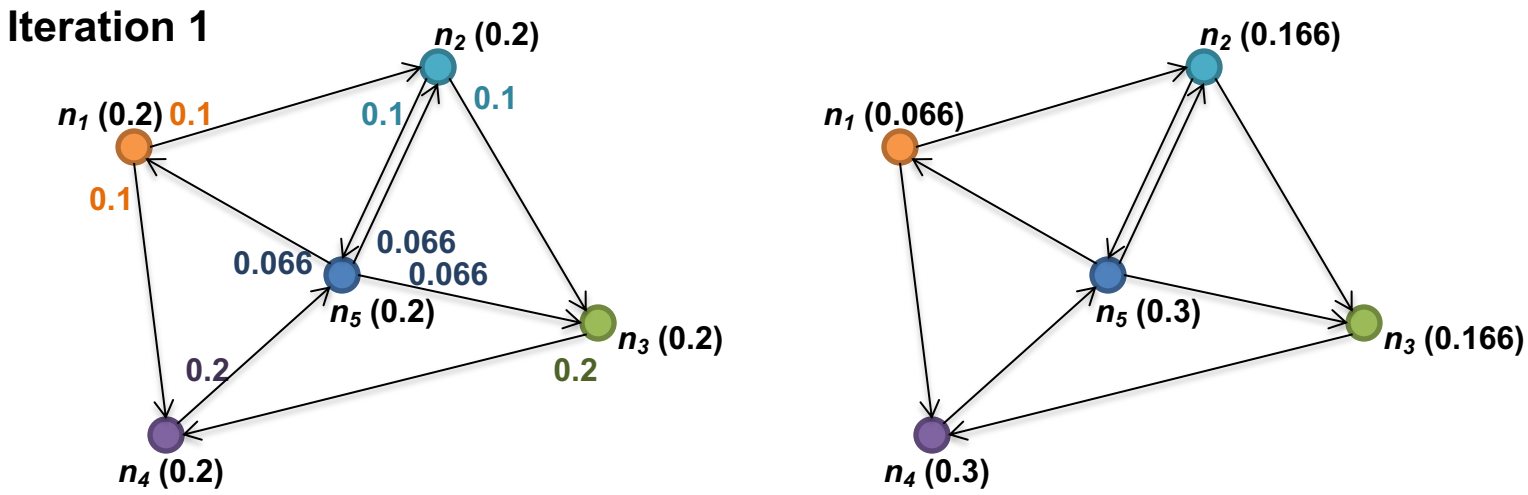
No dangling nodes

Then, factor in these complexities…

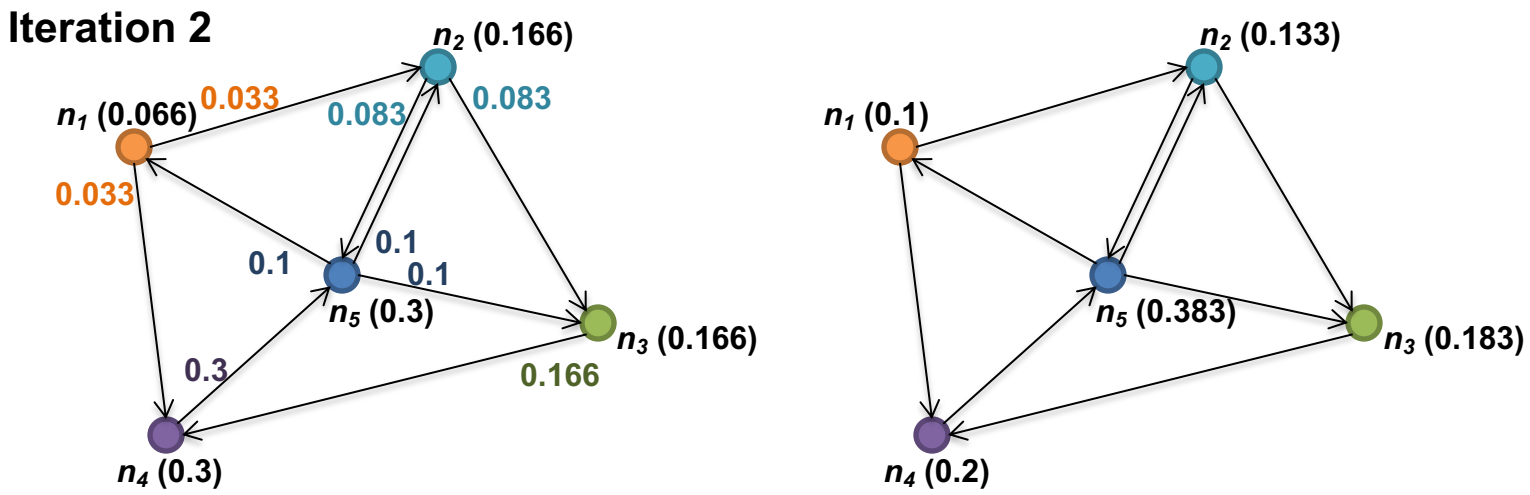
Why do we need the random jump?

Where do dangling nodes come from?

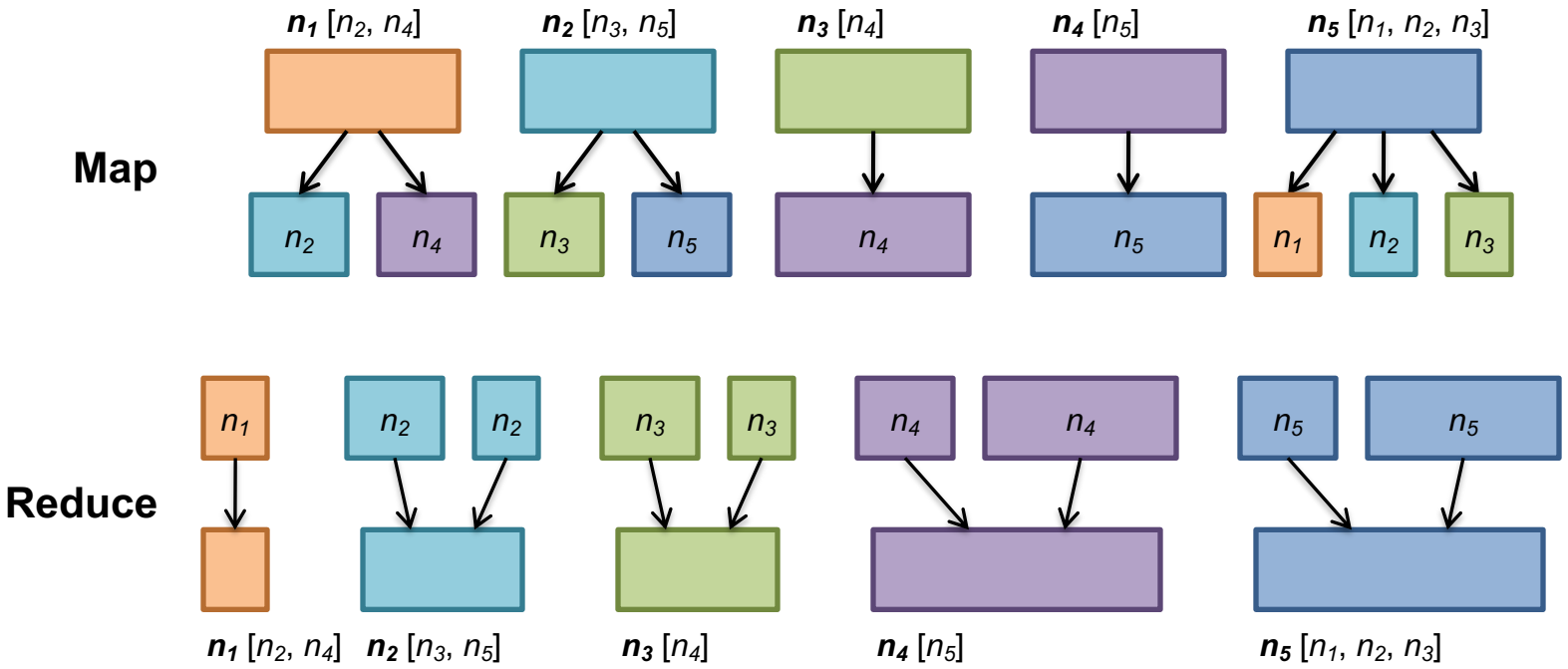
Sample PageRank Iteration (1)



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

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

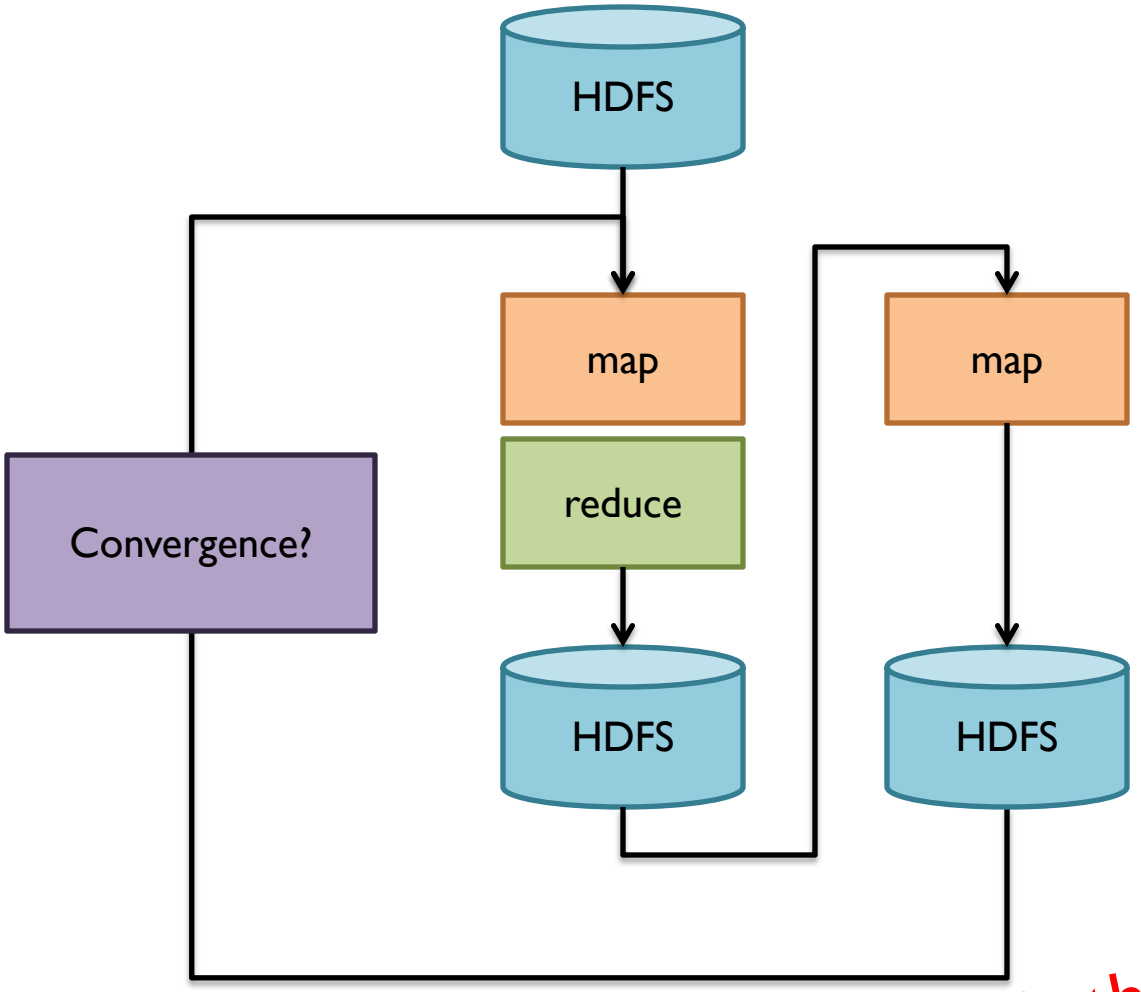
p is PageRank value from before, *p'* is updated PageRank value

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

…

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

WTF: the who to follow service at Twitter

Authors: Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, Reza Zadeh | Authors Info & Claims

WWW '13: Proceedings of the 22nd international conference on World Wide Web • Pages 505 - 514
https://doi.org/10.1145/2488388.2488433

Abstract

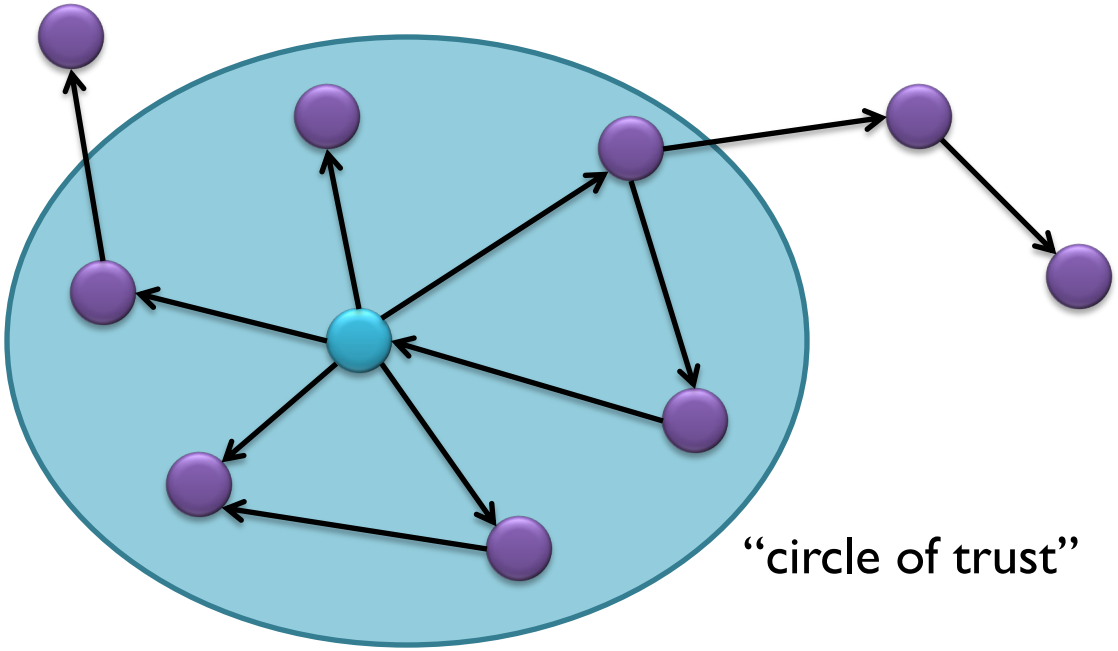
WTF ("Who to Follow") is Twitter's user recommendation service, which is responsible for creating millions of connections daily between users based on shared interests, common connections, and other related factors. This paper provides an architectural overview and shares lessons we learned in building and running the service over the past few years. Particularly noteworthy was our design decision to process the entire Twitter graph in memory on a single server, which significantly reduced architectural complexity and allowed us to develop and deploy the service in only a few months. At the core of our architecture is Cassovary, an open-source in-memory graph processing engine we built from scratch for WTF. Besides powering Twitter's user recommendations, Cassovary is also used for search, discovery, promoted products, and other services as well. We describe and evaluate a few graph recommendation algorithms implemented in Cassovary, including a novel approach based on a combination of random walks and SALSA. Looking into the future, we revisit the design of our architecture and comment on its limitations, which are presently being addressed in a second-generation system under development.

“Circle of Trust”

Ordered set of important neighbors for a user

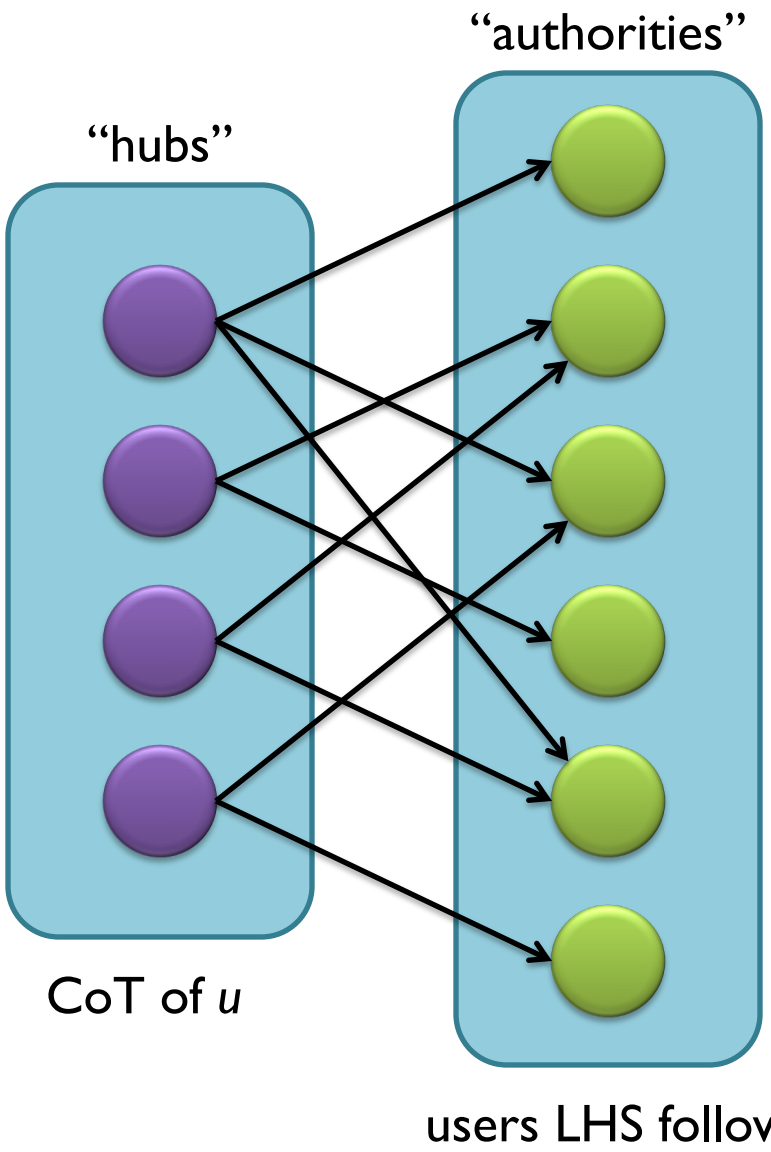
Result of egocentric random walk: Personalized PageRank!

Computed online based on various input parameters



One of the features used in search

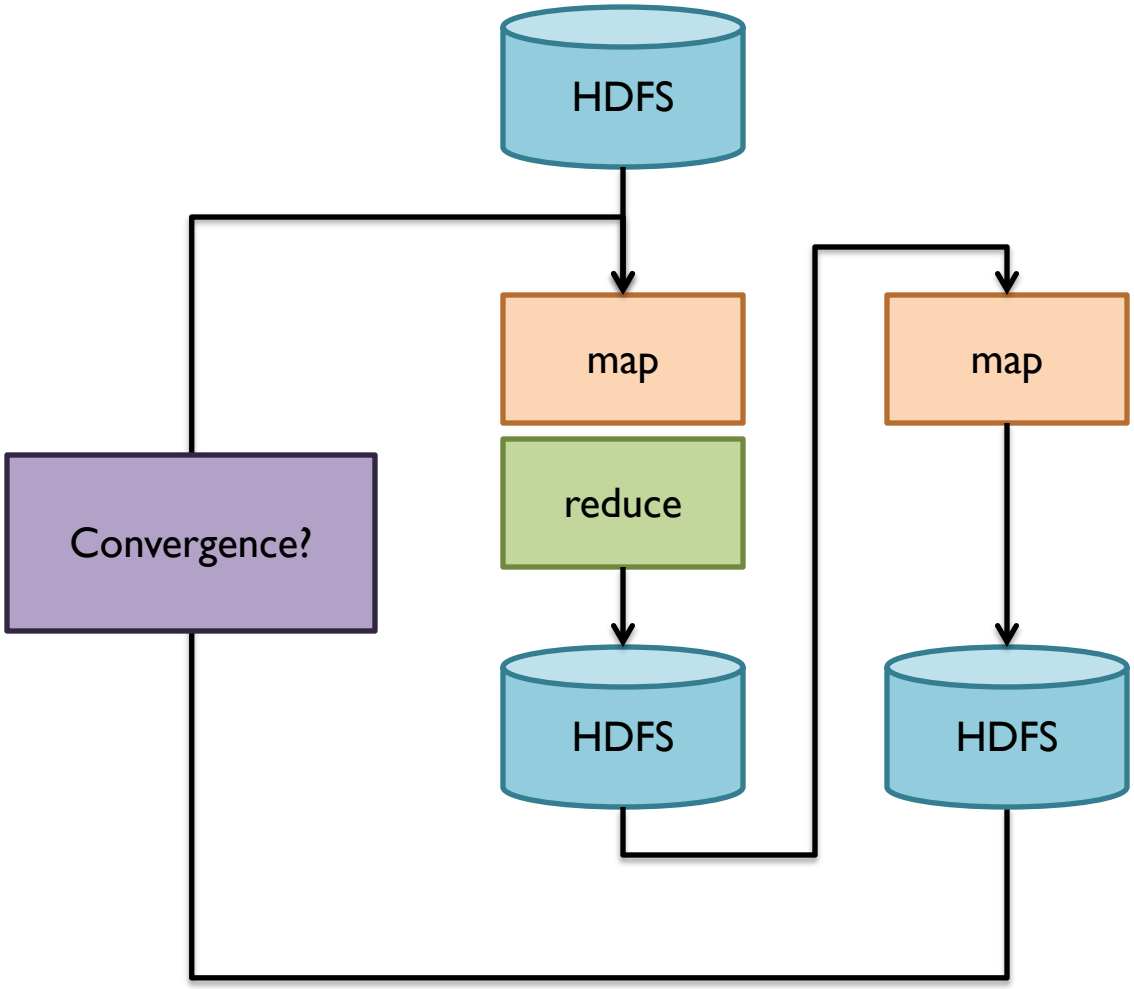
SALSA for Recommendations



hubs scores:
similarity scores to *u*

authority scores:
recommendation scores for *u*

Implementation Practicalities



MapReduce Sucks

Java verbosity

Hadoop task startup time

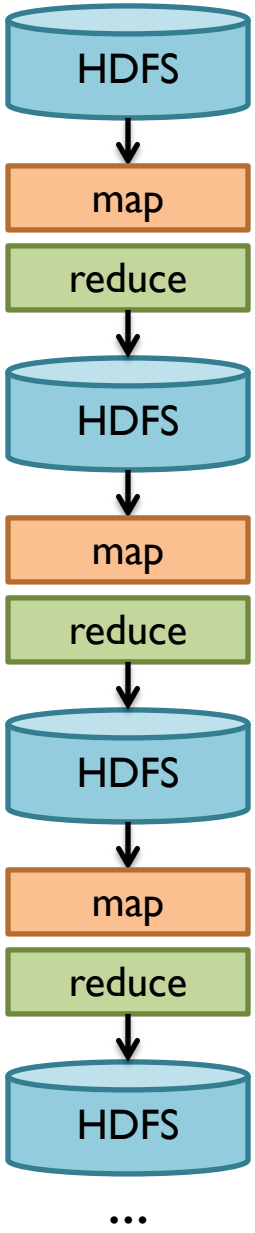
Stragglers

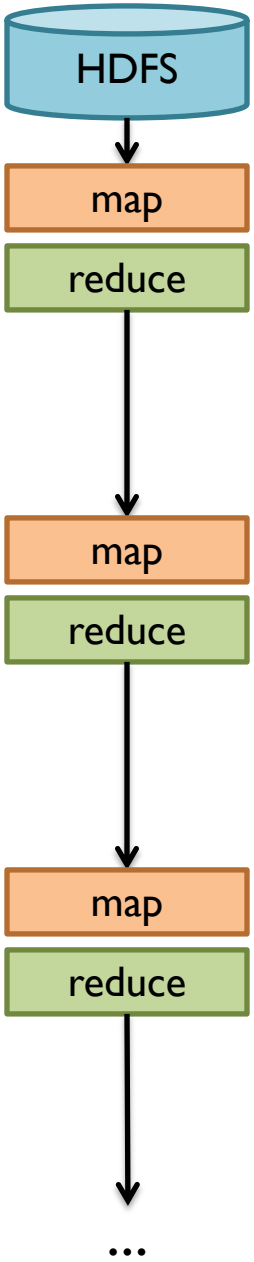
Needless graph shuffling

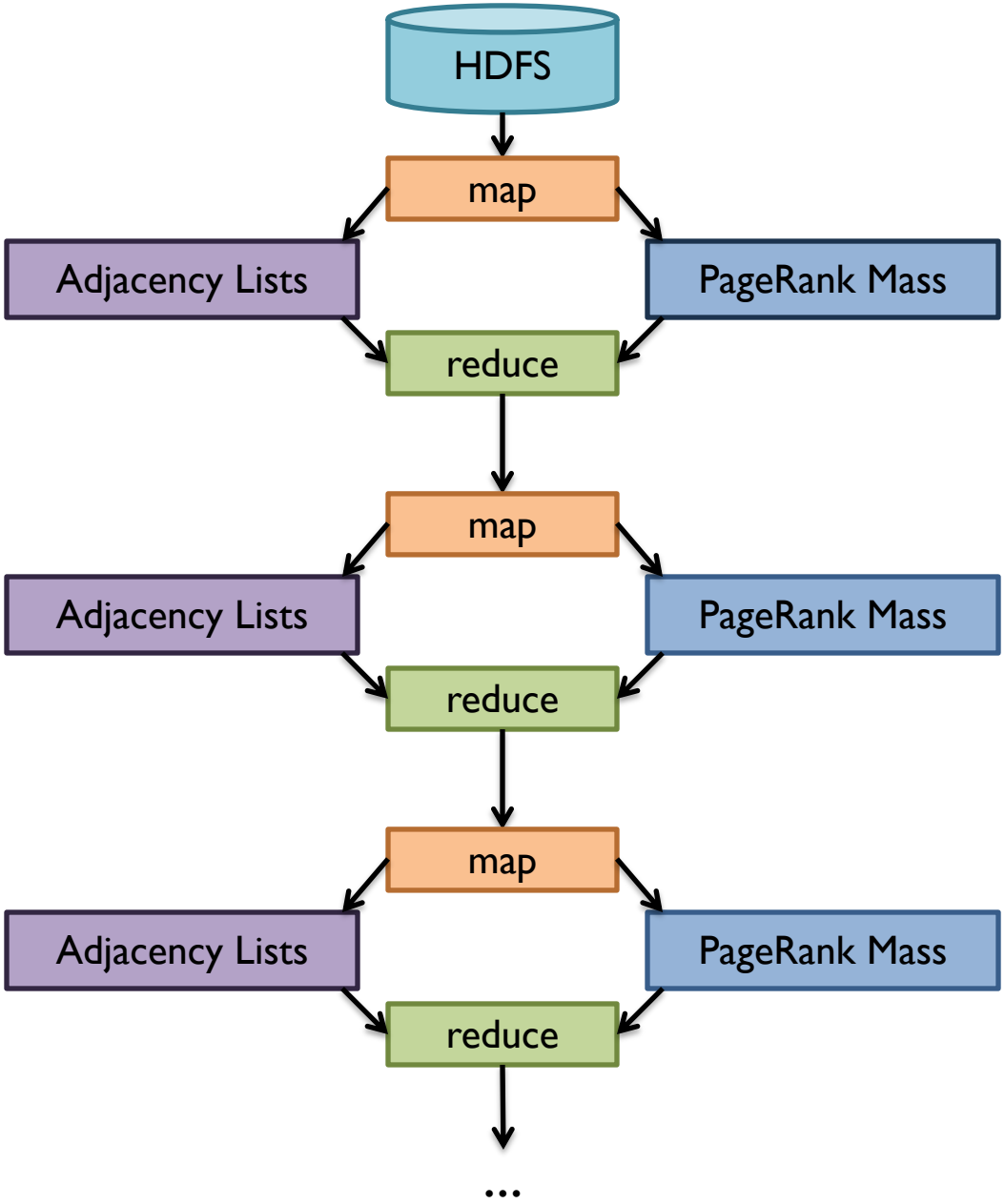
Checkpointing at each iteration

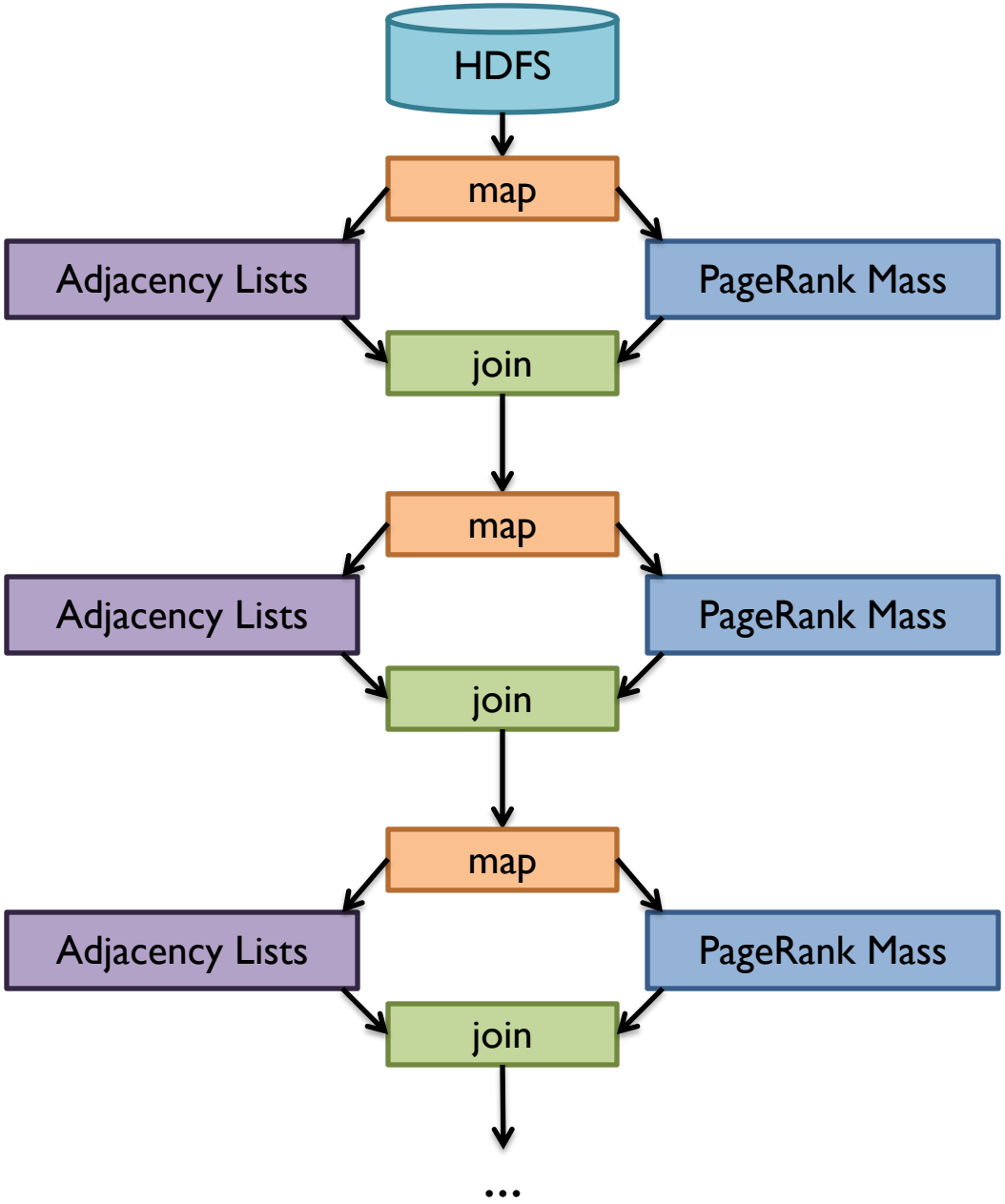
Spark to the rescue?

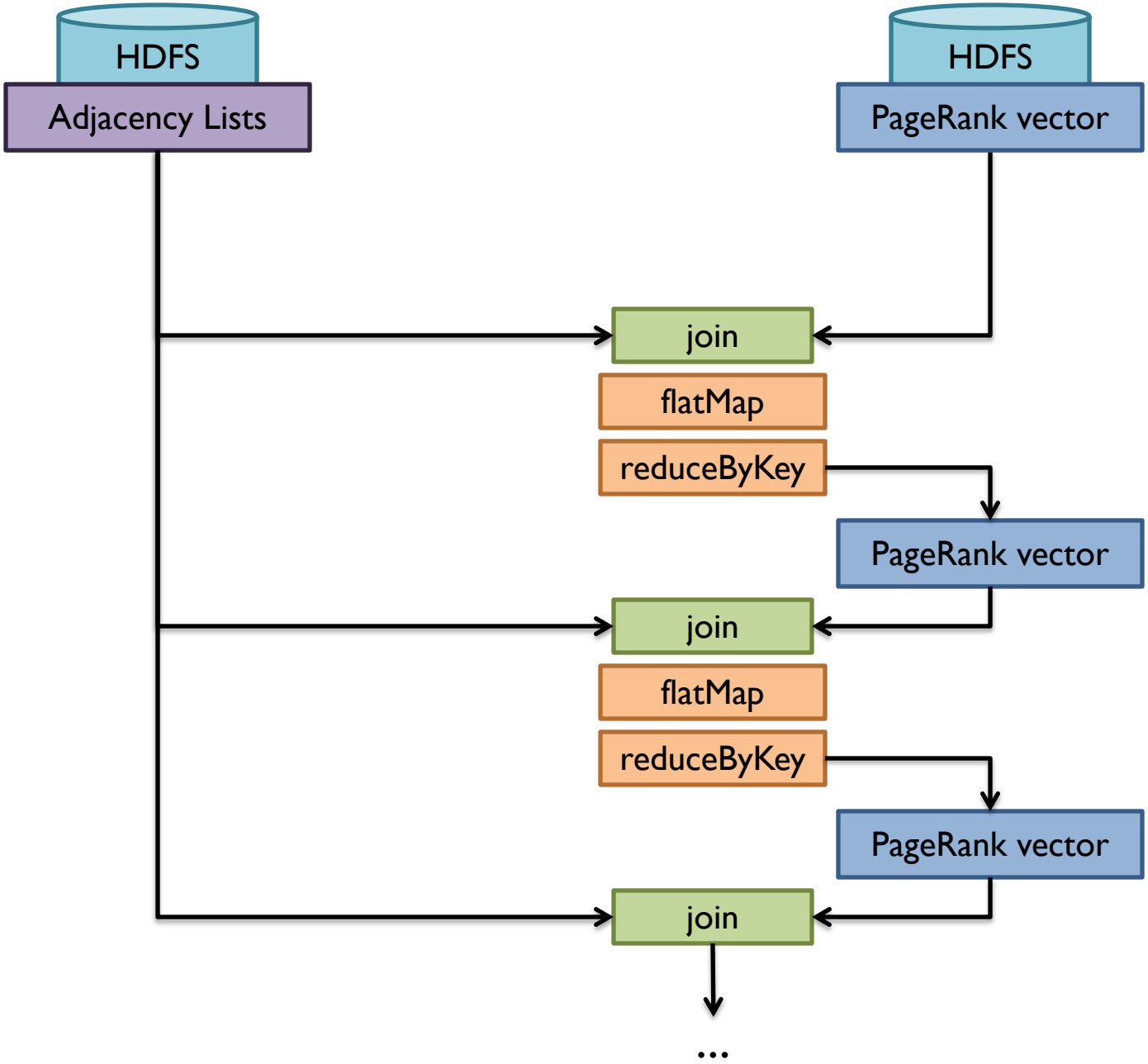
Let's Spark!

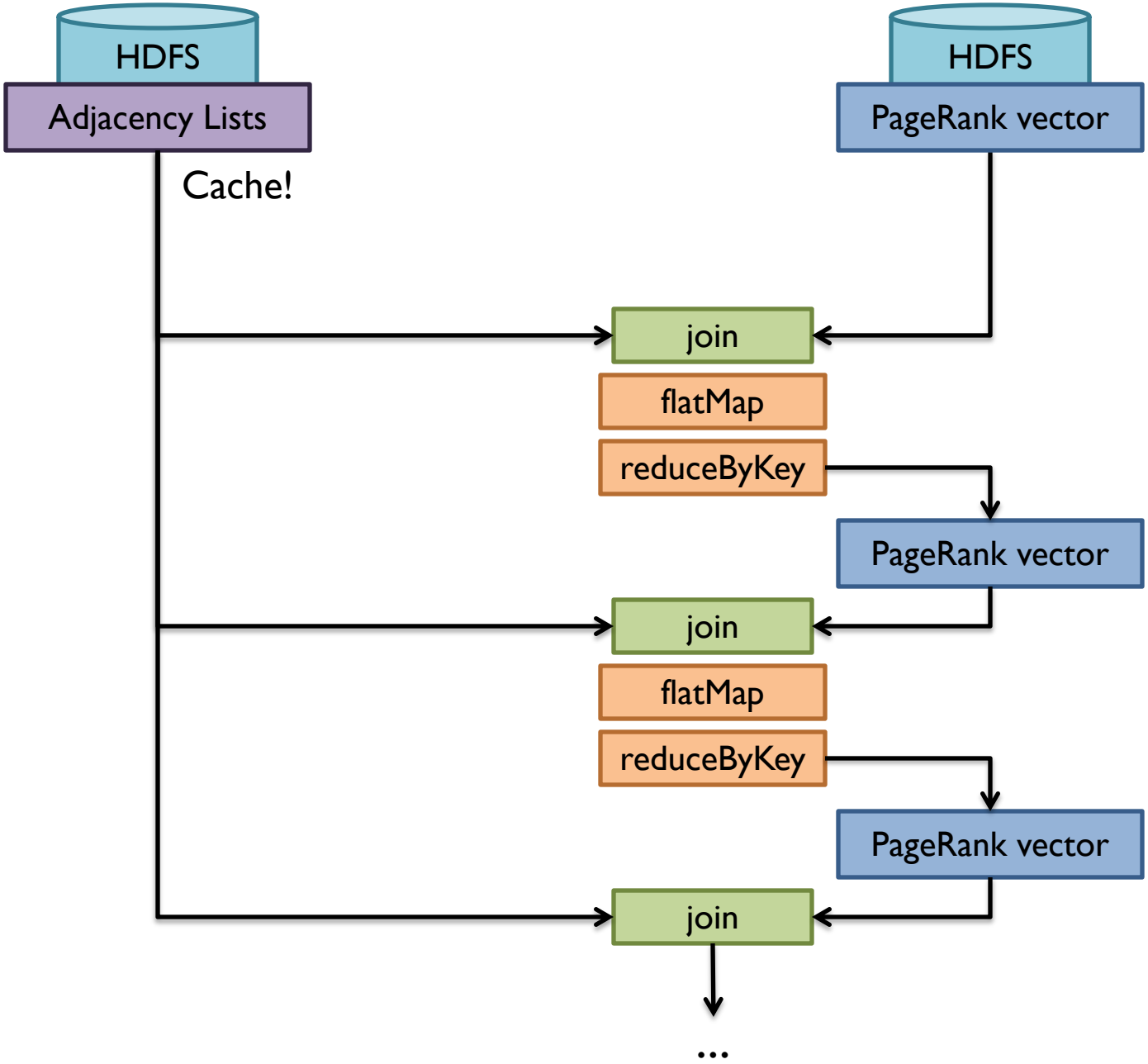



HDFS
map
reduce
map
reduce
map
reduce
...

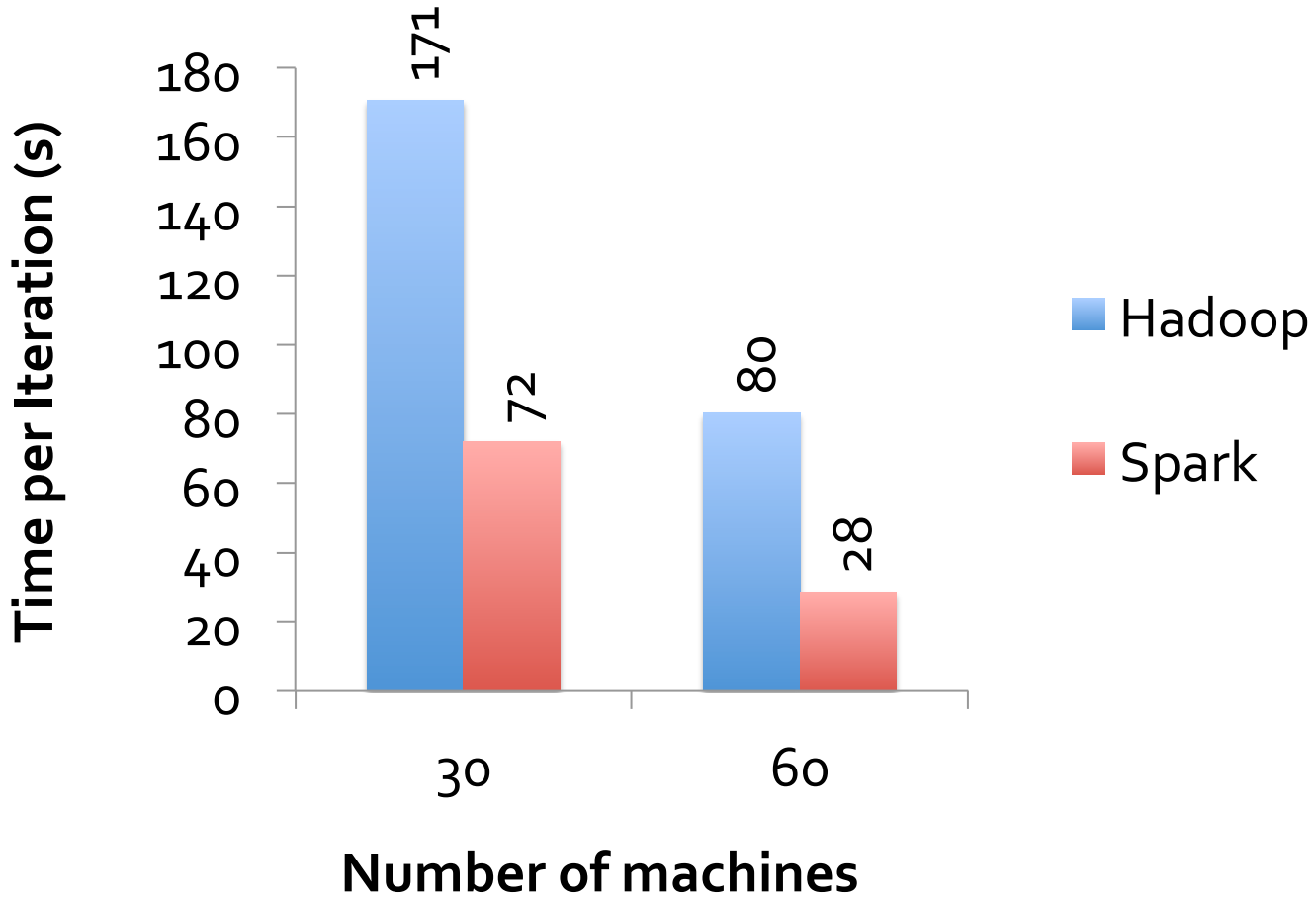

HDFS
map
Adjacency Lists
PageRank Mass
reduce
map
Adjacency Lists
PageRank Mass
reduce
map
Adjacency Lists
PageRank Mass
reduce
...


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
Cache!
HDFS
PageRank vector
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?

Pregel: “think like a vertex”

Pregel: Computational Model

Based on Bulk Synchronous Parallel (BSP)

Computational units encoded in a directed graph

Computation proceeds in a series of supersteps

Message passing architecture

Each vertex, at each superstep:

Receives messages directed at it from previous superstep

Executes a user-defined function (modifying state)

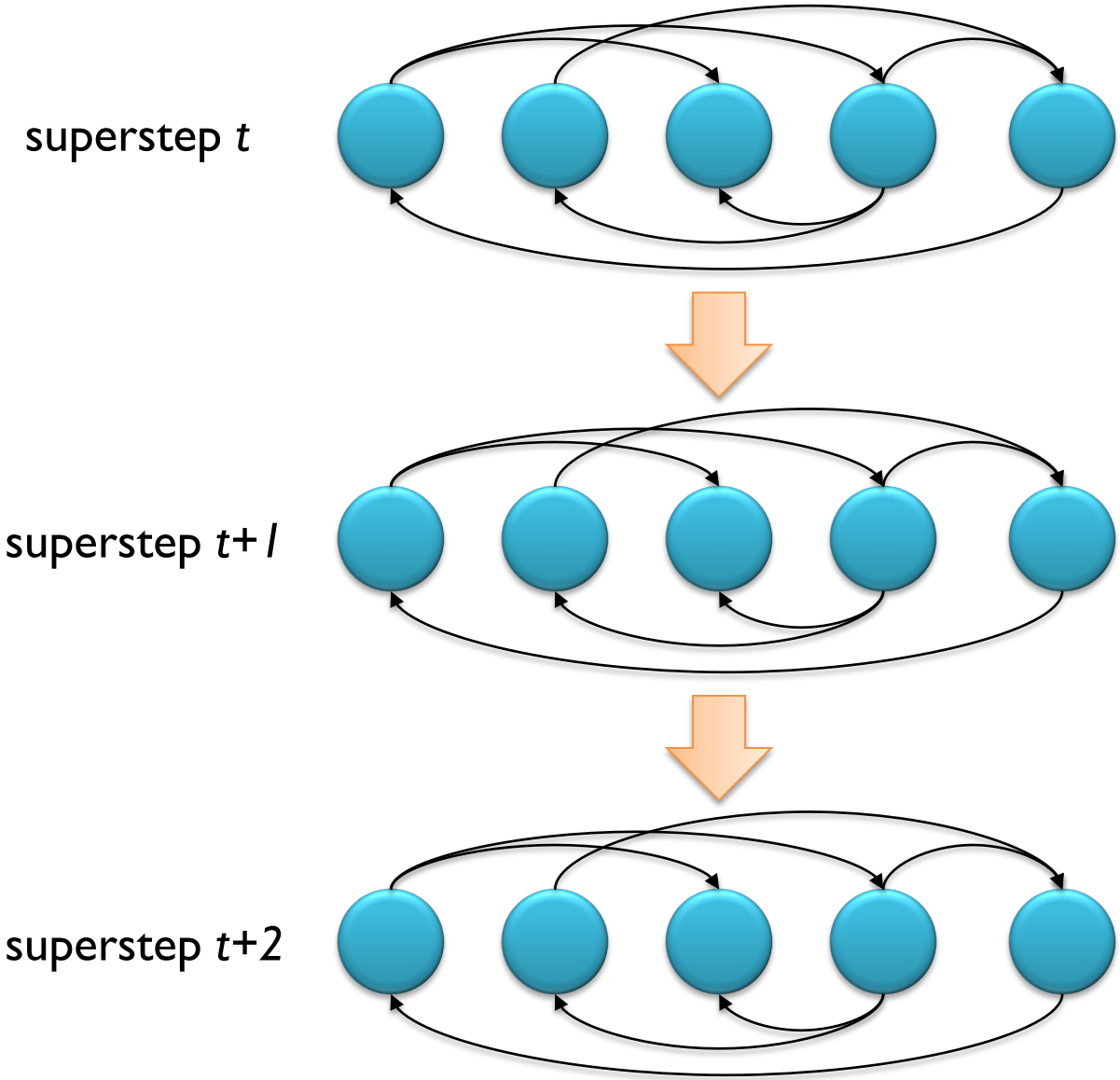
Emits messages to other vertices (for the next superstep)

Termination:

A vertex can choose to deactivate itself

Is "woken up" if new messages received

Computation halts when all vertices are inactive



Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.

Pregel: Implementation

Coordinator-Worker architecture

Vertices are hash partitioned (by default) and assigned to workers

Everything happens in memory

Processing cycle:

Coordinator tells all workers to advance a single superstep

Worker delivers messages from previous superstep, executing vertex computation

Messages sent asynchronously (in batches)

Worker notifies coordinator of number of active vertices

Fault tolerance

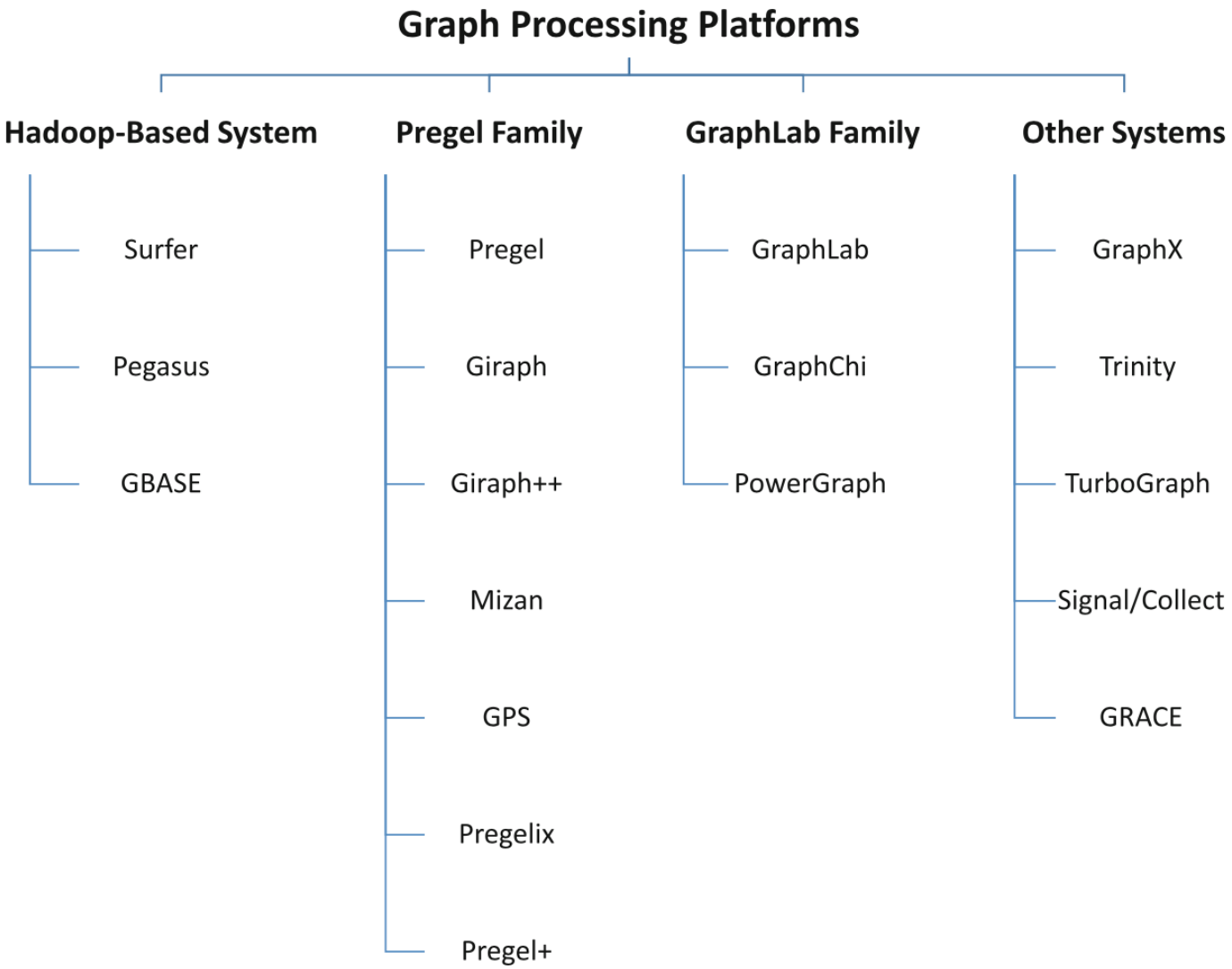
Checkpointing

Heartbeat/revert

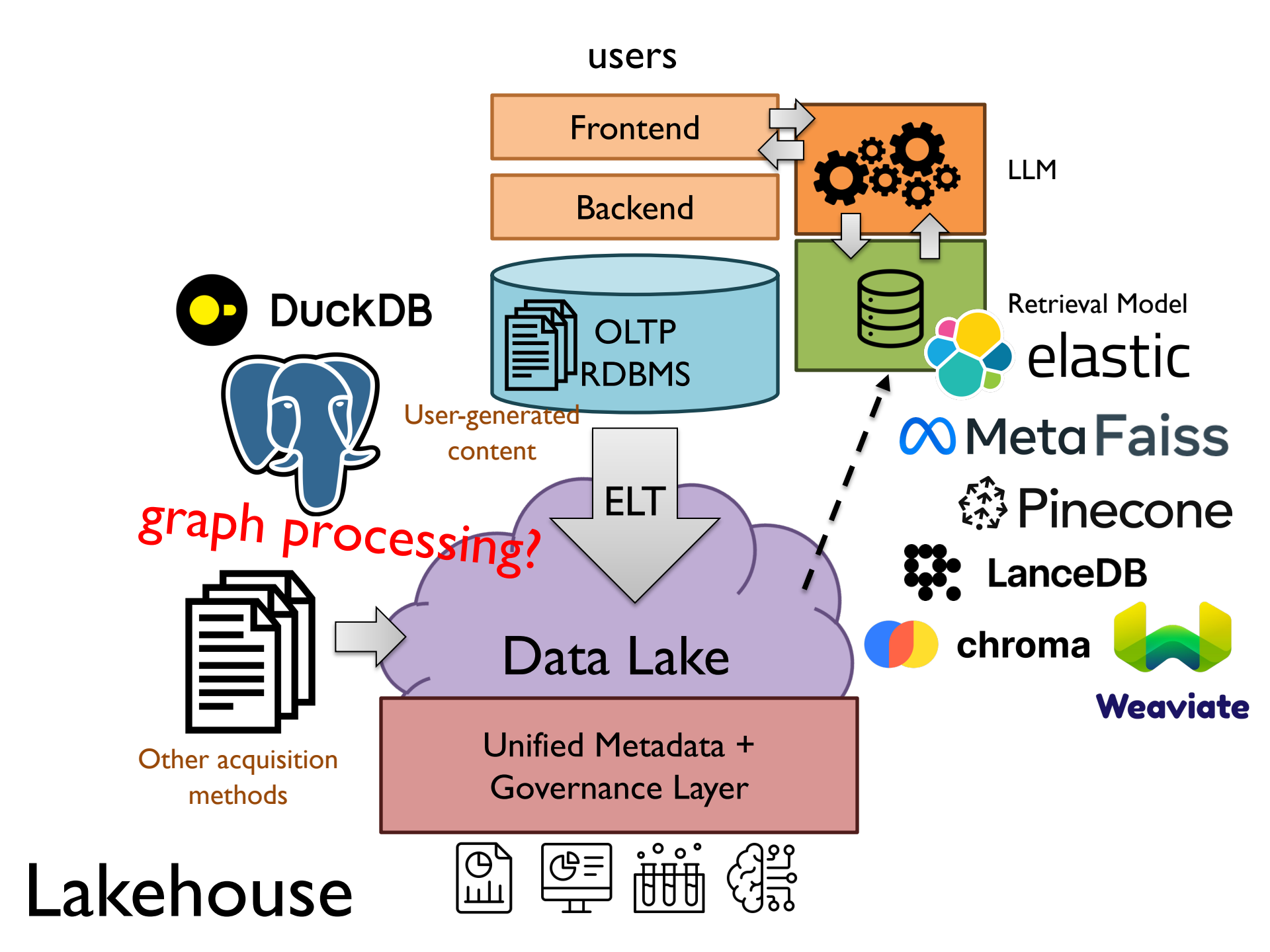
Pregel: SSSP

```
class ShortestPathVertex : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

Source: Malewicz et al. (2010) Pregel: A System for Large-Scale Graph Processing. SIGMOD.



Sherif Sakr. Big Data 2.0 Processing Systems. 2016.


users
Frontend
Backend
LLM
DuckDB
OLTP
RDBMS
Retrieval Model
elastic
User-generated
content
ELT
graph processing?
Meta Faiss
Pinecone
LanceDB
chroma
Weaviate
Data Lake
Unified Metadata +
Governance Layer
Other acquisition
methods
Lakehouse

Two Separate Requirements

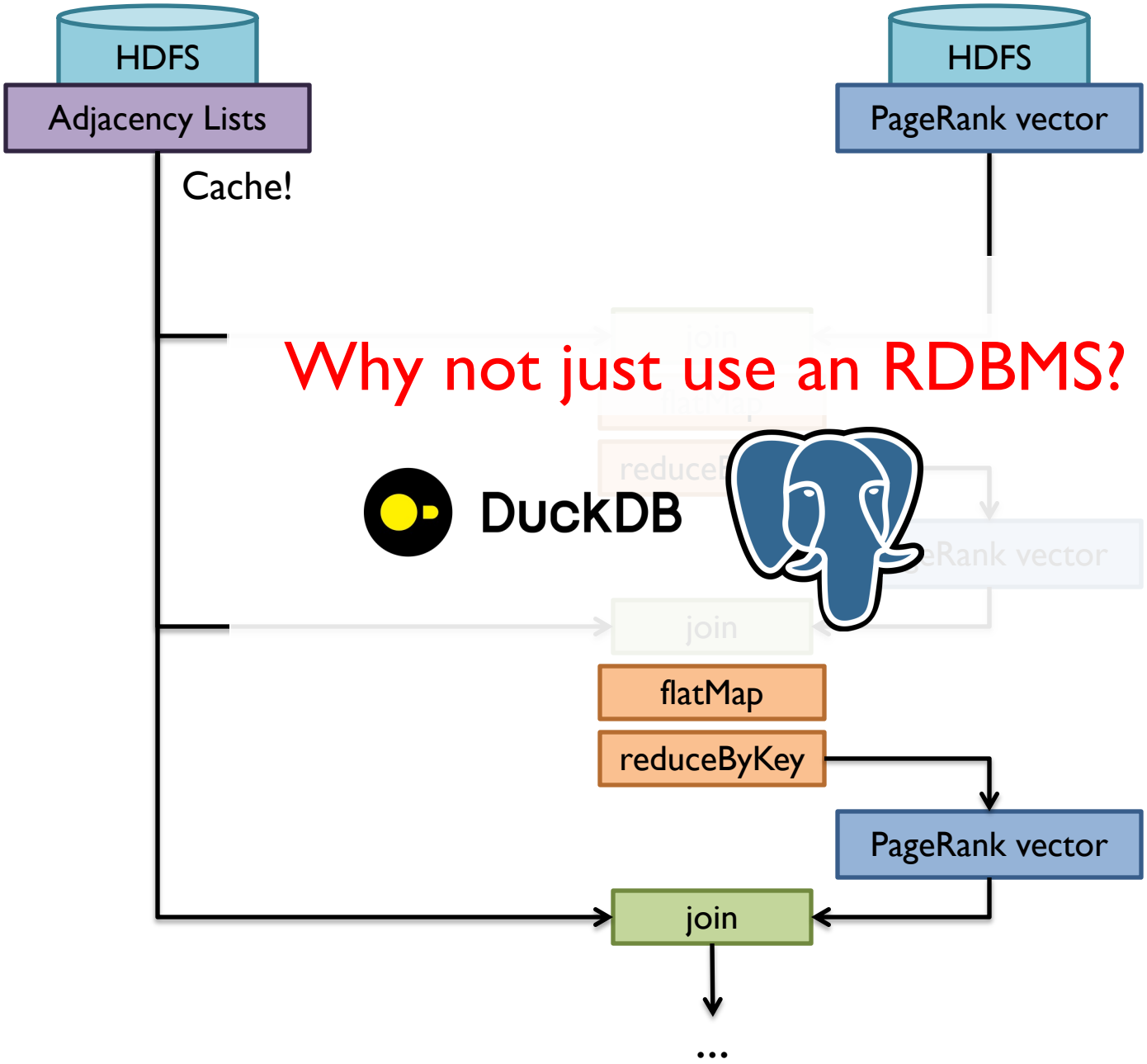
Data manipulation to build graphs

Running graph algorithms

How large are your graphs?

One billion edges as (src, dst) pairs occupies *only* 8GB!

But what else are we forgetting?


HDFS
Adjacency Lists
HDFS
PageRank vector
Cache!
Why not just use an RDBMS?
DuckDB
flatMap
reduceByKey
PageRank vector
join
...


富嶽三十六景
神奈川沖
浪裏