Important actions items during next week

Throughout this week and early next week (Sep20 to Sep 28) we will

- Release papers/tools for presentation and the signup sheet (First-come First-choose!)
- Release project member signup sheet (you can work in groups of two or as a single person for projects)
- Release project, paper, and tool presentation guidelines
- First presentation date is Oct 4 but reading papers might start sooner!
- Each group (of two) presents one-two papers and possibly a tool
- Look for presentation/project partners now and start discussing your project direction!

Look for announcements in Piazza!

Tentative Topics for Paper Presentations

- Domain-specific compilers: Halide, Sympiler, etc.: Structured Grids and Sparse Computations
- Cache Oblivious Methods and the Cilk runtime: Graph Traversal
- Fast Fourier Transforms: FFTW, Spiral, etc: Spectral
- Hierarchical Methods : N-Body
- Cloud Computing: Mapreduce
- Others: Roofline model, communication-avoiding methods, applications

Cloud Computing and Big Data Processing

Some slides from Shivaram Venkataraman and Matei Zaharia and Kathy Yelick

Cloud Computing, Big Data

Big data, Hardware, Software, Business

Data, Data, Data

"…Storage space must be used efficiently to store indices and, optionally, the documents themselves. The indexing system must process hundreds of gigabytes of data efficiently…"

The Anatomy of a Large-Scale Hypertextual Web Search Engine

Sergey Brin and Lawrence Page

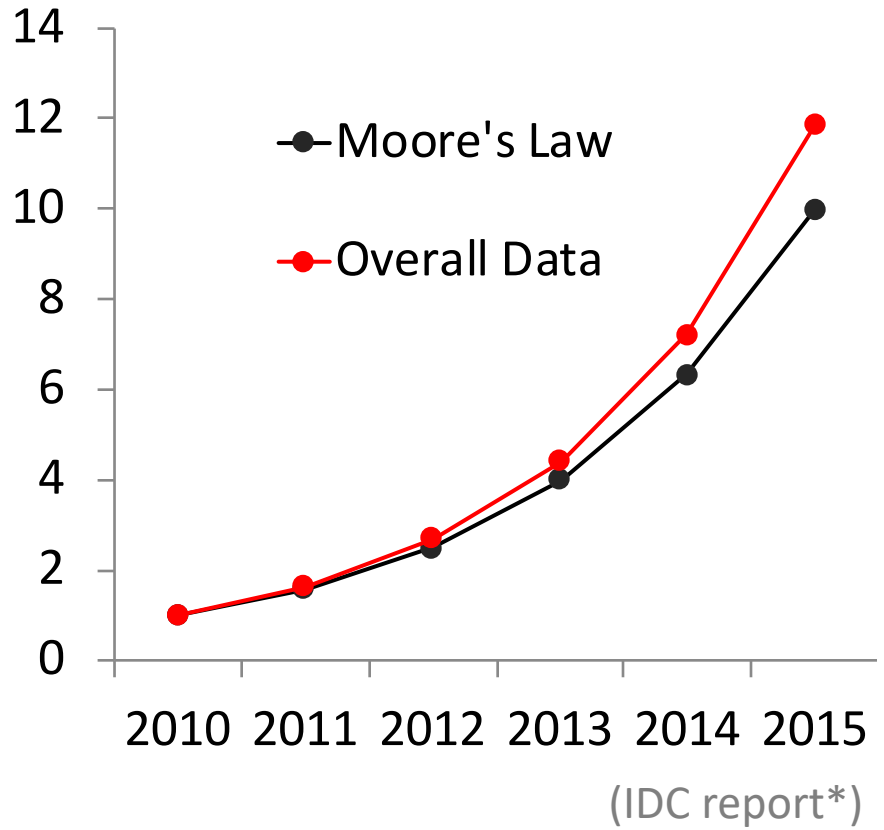
http://infolab.stanford.edu/~backrub/google.html

Datacenter Evolution

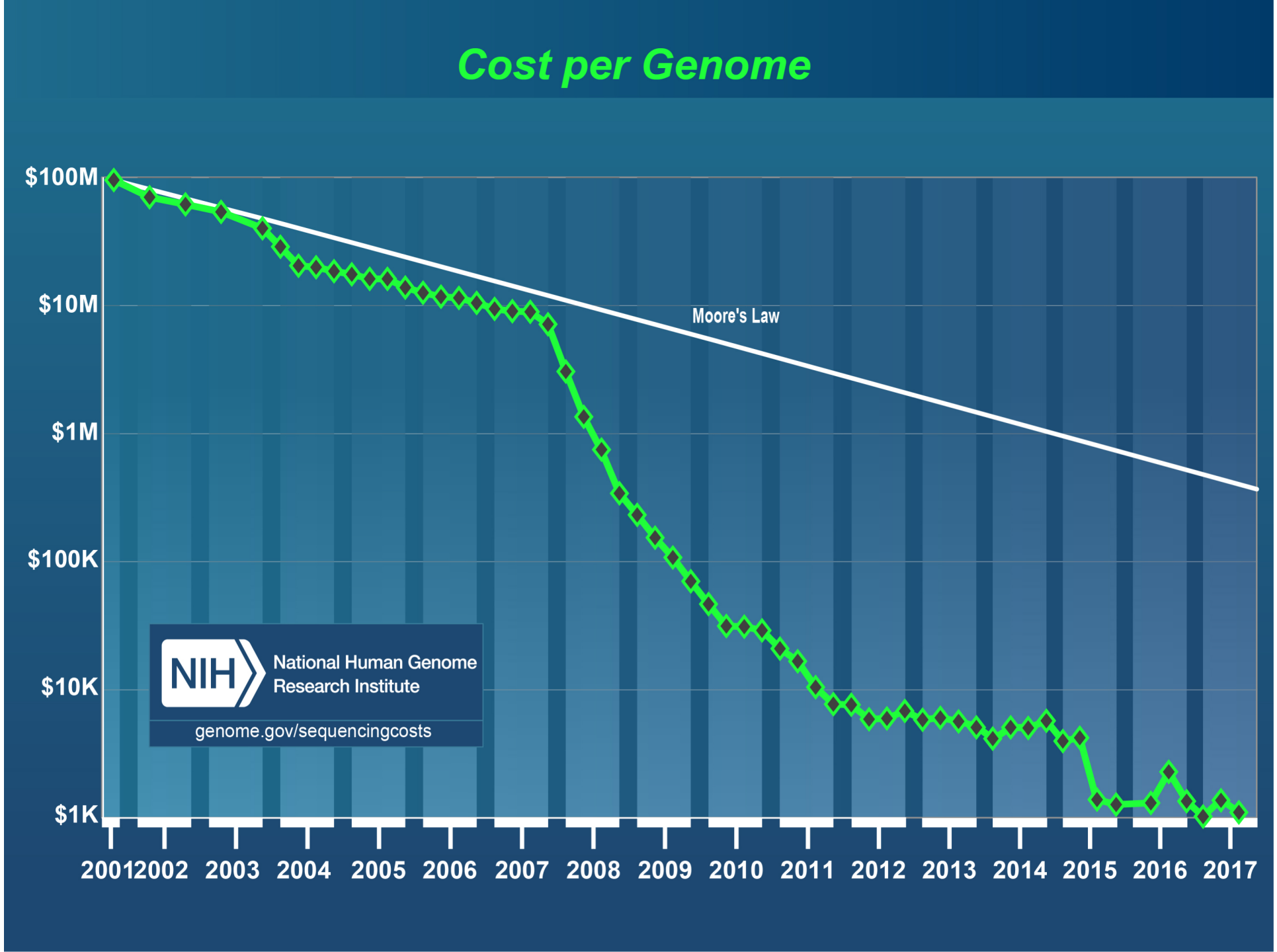
Facebook's daily logs:
60 TB

1000 genomes project:
200 TB

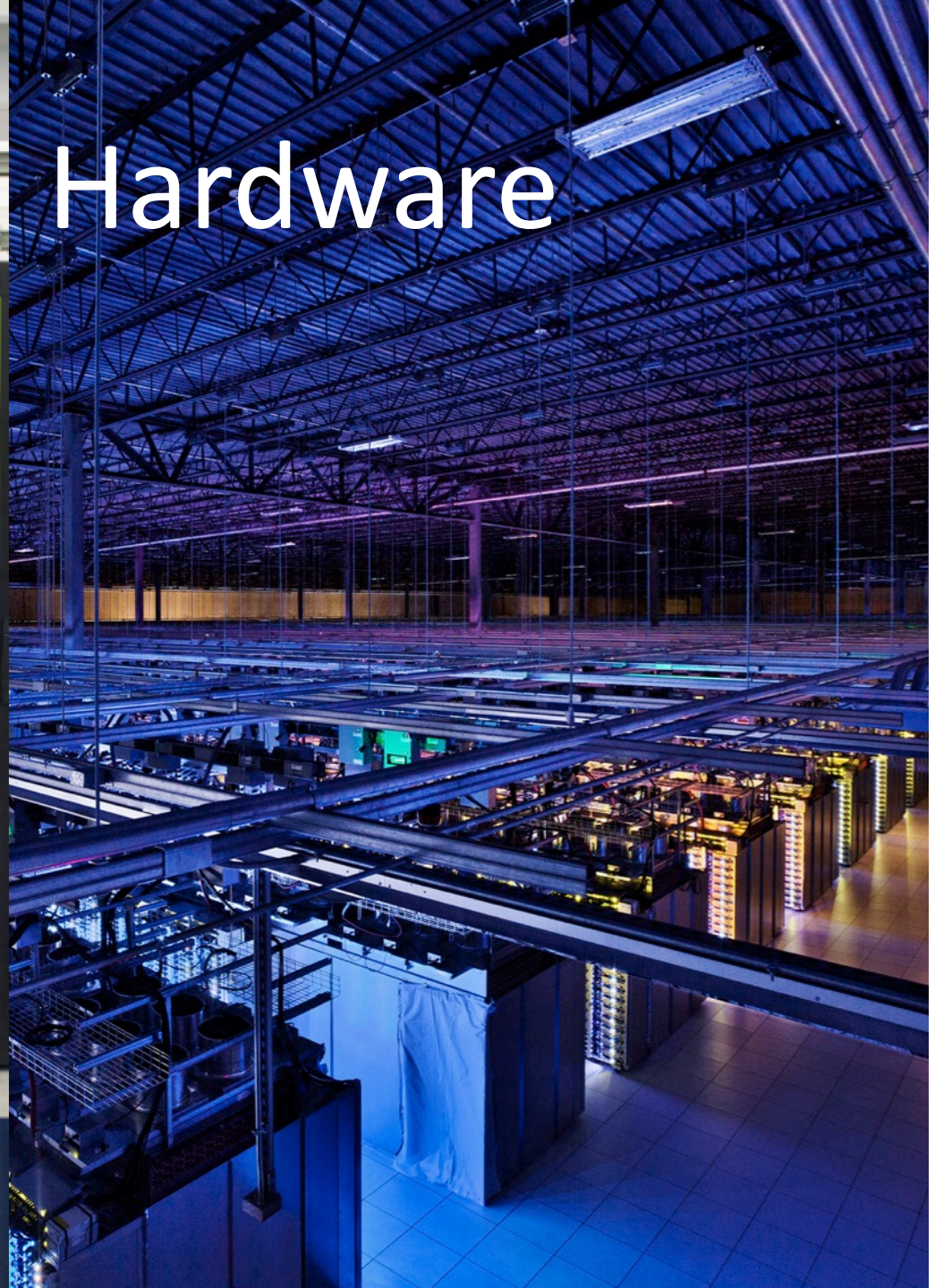
Google web index:
10+ PB



Data from Ion Stoica


Cost per Genome
$100M
$10M
$1M
$100K
$10K
$1K
Moore's Law
NIH
National Human Genome Research Institute
genome.gov/sequencingcosts
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017

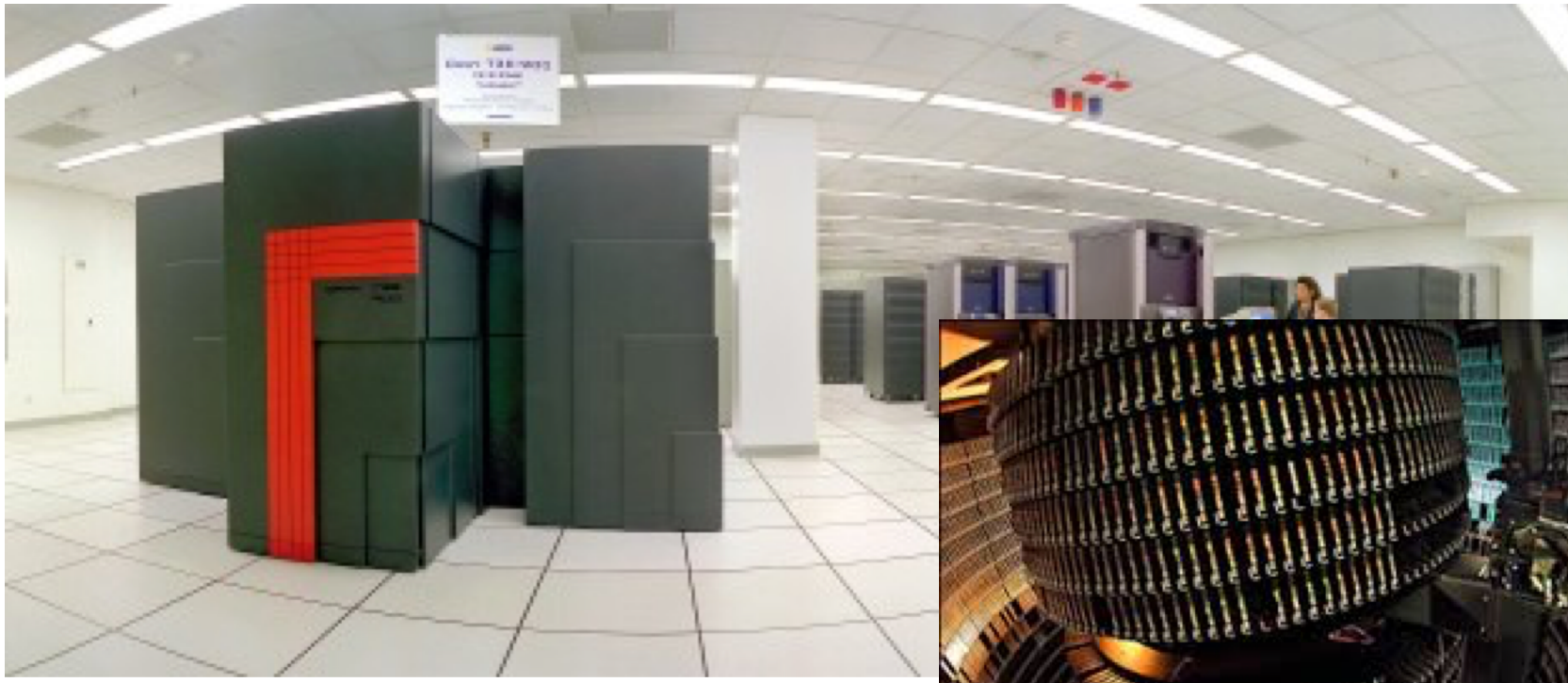
Hardware



Google 1997

10 GB of disk

NERSC 1996

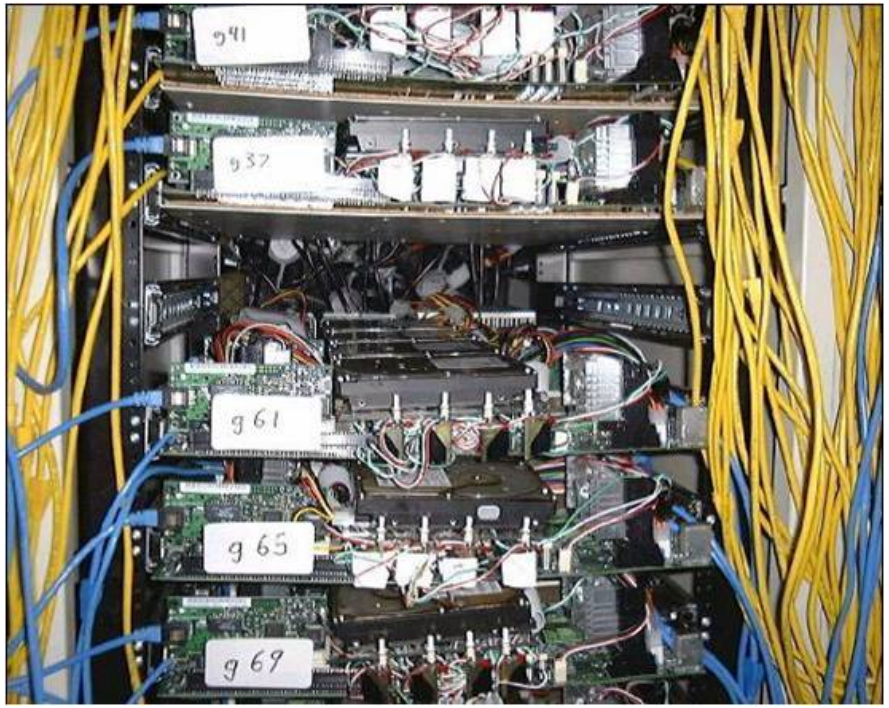


Cray T3E 900: 460 Gflop/s

850 GB of disk

106 TB Data archive

Google 2001



3 TB of disk

Commodity CPUs

Lots of disks

Low bandwidth network

Cheap !

NERSC 2001: Seaborg

2001 5 Tflop/s 3.3 TB upgraded in 2002 to 10 Tflop/s, 6.6 TB

Datacenter Evolution

Google data centers in The Dalles, Oregon

- 200K SF + 164K sf in (3 buildings total)
- $1.2 B investment in site
- 175 people employed on site
- 70 Megawatts (when it was 200K SF)

Datacenter Evolution

Capacity:

~10000 machines

Bandwidth:

12-24 disks per node

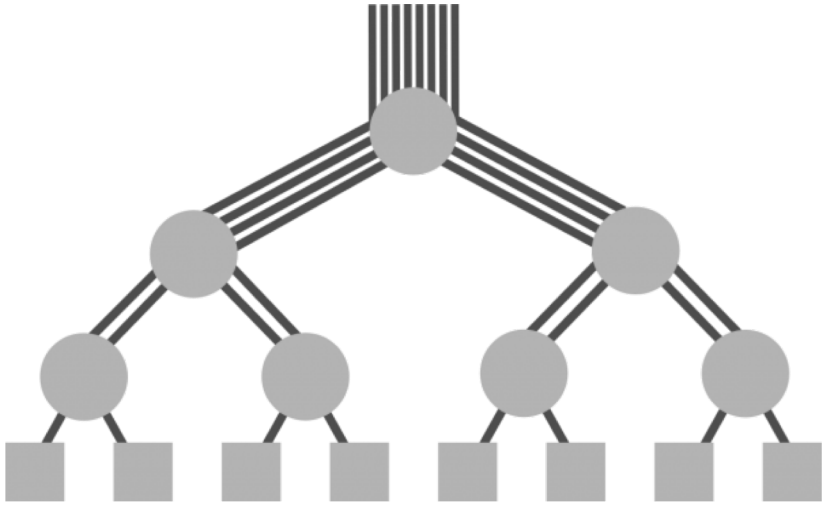
Latency:

256GB RAM cache

Datacenter Networking

Initially tree topology

Over subscribed links



Fat tree, Bcube, VL2 etc.

Lots of research to get full bisection bandwidth

Datacenter Design

Goals

Power usage effectiveness (PUE)

Cost-efficiency

Custom machine design

Open Compute Project

(Facebook)

Datacenters → Cloud Computing

Above the Clouds: A Berkeley View of Cloud Computing

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz,
Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia
(Comments should be addressed to abovetheclouds@cs.berkeley.edu)

UC Berkeley Reliable Adaptive Distributed Systems Laboratory *
http://radlab.cs.berkeley.edu/

"…long-held dream of computing as a utility…"

From Mid 2006

Rent virtual computers in the “Cloud”

On-demand machines, spot pricing

Amazon EC2 (2014)

Machine	Memory (GB)	Compute Units (ECU)	Local Storage (GB)	Cost / hour
t1.micro	0.615	1	0	$0.02
m1.xlarge	15	8	1680	$0.48
cc2.8xlarge	60.5	88 (Xeon 2670)	3360	$2.40

1 ECU = CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor

Amazon EC2 (2015)

Machine	Memory (GB)	Compute Units (ECU)	Local Storage (GB)	Cost / hour
t2.micro	~~0.615~~ 1	1	0	$0.013
r3.xlarge	~~15~~ 30	~~8~~ 13	~~1680~~ 80(SSD)	$0.35
r3.8xlarge	~~60.5~~ 244	~~88~~ 104 (32 Ivy Bridge)	~~3360~~ 640(SSD)	$2.80

1 ECU = CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor

Amazon EC2 (2016)

Machine	Memory (GB)	Compute Units (ECU)	Local Storage (GB)	Cost / hour
t2.nano	0.5	1	0	$0.006
t2.micro	1	1	0	$0.013
r3.8xlarge	~~60.5~~ 244	~~88~~ 104 (32 Ivy Bridge)	~~3360~~ 640(SSD)	$2.80
x1 (TBA)	2 TB	4 * Xeon E7	?	?

Amazon EC2 (2017)

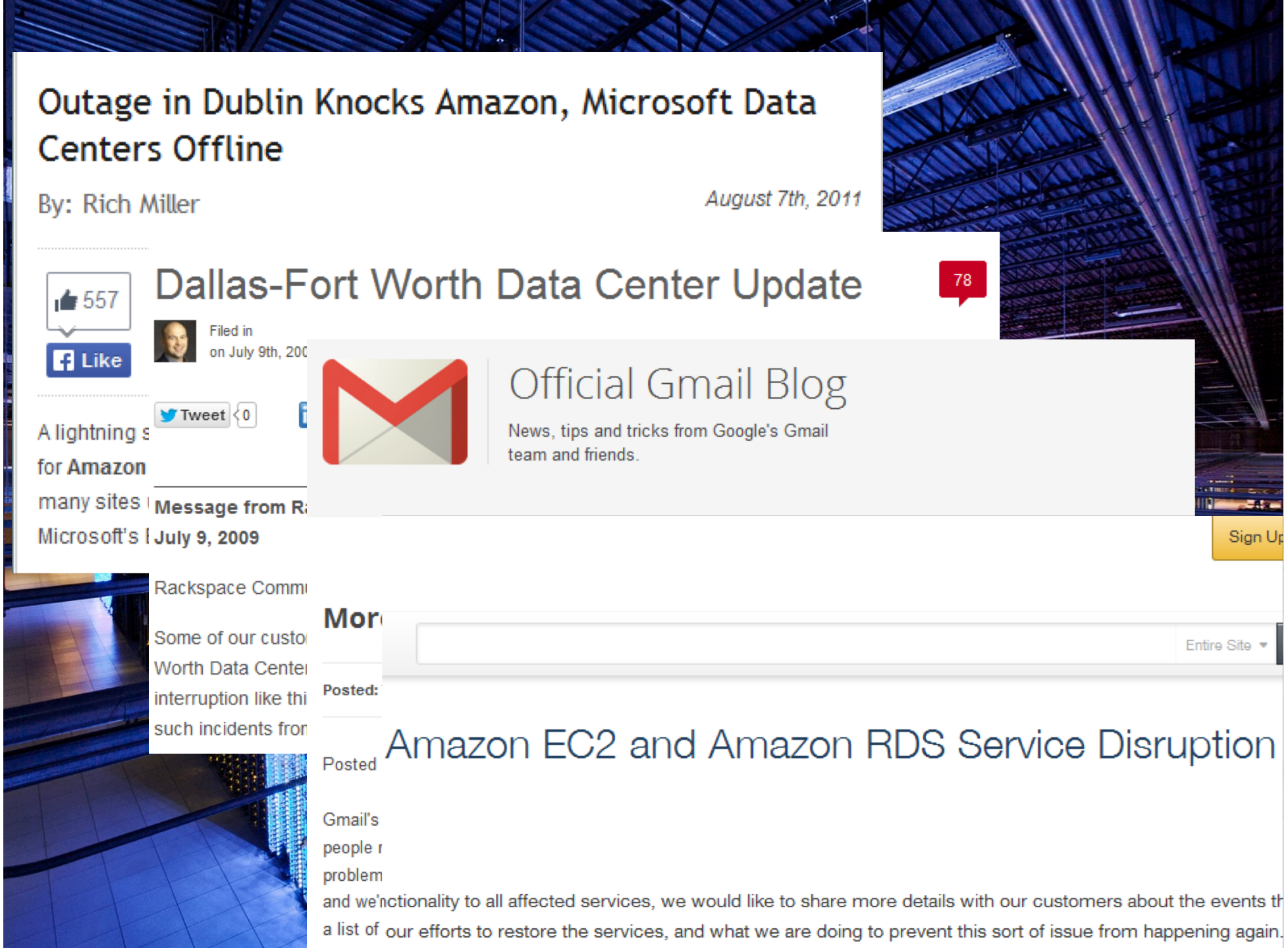
Machine	Memory (GB)	Compute Units (ECU)	Local Storage (GB)	Cost / hour
t2.nano	0.5	1	0	$0.006
r3.8xlarge	244	104 (32 Ivy Bridge)	640 (SSD)	~~$2.80~~ $2.66
x1.32xlarge	2 TB	4 * Xeon E7	3.8 TB (SSD)	$13.338
p2.16xlarge	732 GB	16 Nvidia K80 GPUs	0	$14.40

Amazon EC2 (2018)

Machine	Memory (GB)	Compute Units (ECU)	Local Storage (GB)	Cost / hour
t2.nano	0.5	1	0	$0.006
r3.8xlarge	244	104 (32 Ivy Bridge)	640 (SSD)	~~$2.66~~ $2.37
x1.32xlarge	2 TB	4 * Xeon E7	3.8 TB (SSD)	$13.338
p2.16xlarge	732 GB	16 Nvidia K80 GPUs	0	$14.40

Summary of Traditional Differences
(both are changing)

Cloud	HPC (e.g. Scinet)
Focus on storage	Focus on computing (flop/s)
Cheap(est) commodity component	High end components (some specialization)
Commodity networks	High performance networks
Pay as you go	Purchased for mission; pay in non-fungible “hours”
< 50% utilization	> 90% utilization
On-demand access	Large jobs wait in queues
On-node disks (air cooled)	Separate storage (compute is liquid cooled)



Outage in Dublin Knocks Amazon, Microsoft Data Centers Offline

By: Rich Miller

August 7th, 2011

557

Like

Dallas-Fort Worth Data Center Update

78

Filed in

on July 9th, 200

Tweet 0

A lightning s

for **Amazon**

many sites

Microsoft's

Message from R

July 9, 2009

Rackspace Commu

Some of our custo

Worth Data Cente

interruption like thi

such incidents fro

Official Gmail Blog

News, tips and tricks from Google's Gmail team and friends.

Sign Up

Mor

Entire Site

Posted:

Posted

Amazon EC2 and Amazon RDS Service Disruption

Gmail's

people r

problem

and we'nctionality to all affected services, we would like to share more details with our customers about the events th

a list of our efforts to restore the services, and what we are doing to prevent this sort of issue from happening again.

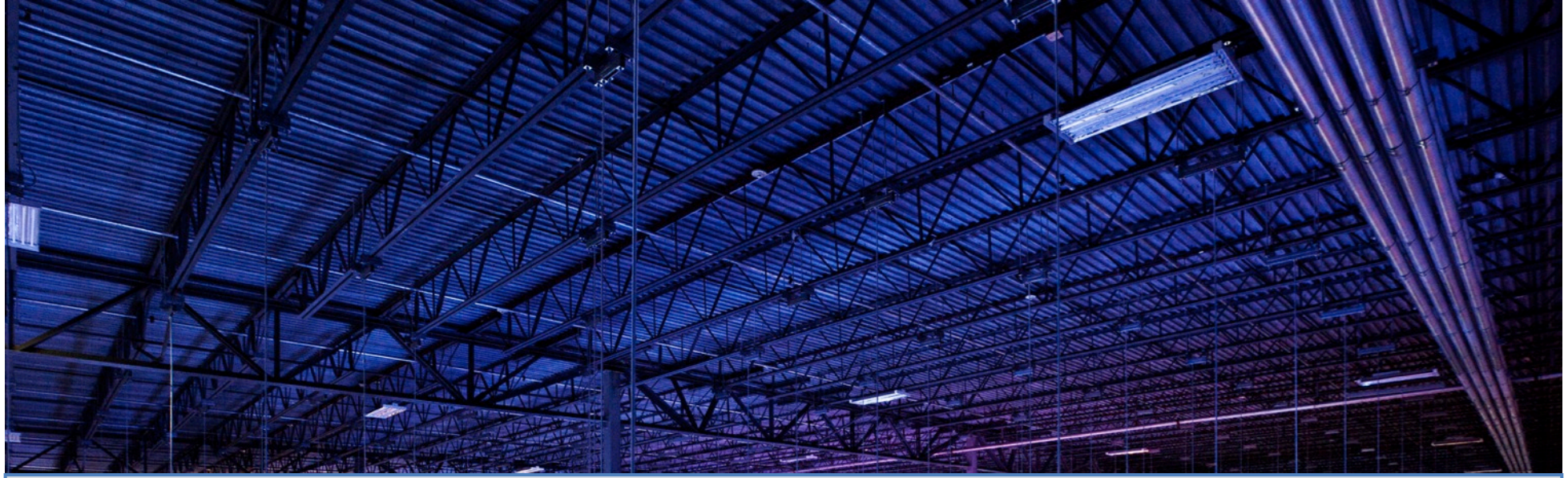
The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
~5 racks go wonky (40-80 machines see 50% packetloss)
~8 network maintenances (4 might cause ~30-minute random connectivity losses)
~12 router reloads (takes out DNS and external vips for a couple minutes)
~3 router failures (have to immediately pull traffic for an hour)
~dozens of minor 30-second blips for dns
~1000 individual machine failures
~thousands of hard drive failures
slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

Jeff Dean @ Google



How do we program this ?

Example Cloud Software

MapReduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

ce is a programming model and an associ-
entation for processing and generating large
ers specify a *map* function that processes a
r to generate a set of intermediate key/value
educe function that merges all intermediate
ated with the same intermediate key. Many
sks are expressible in this model, as shown

given day, etc. Most such computations a
ally straightforward. However, the input da
large and the computations have to be distri
hundreds or thousands of machines in orde
a reasonable amount of time. The issues of
allelize the computation, distribute the data
failures conspire to obscure the original sir
tation with large amounts of complex code
these issues.

Google 2004

Build search index
Compute
PageRank

Hadoop: Open-source
at Yahoo, Facebook

MapReduce Programming Model

Data type: Each record is (key, value)

Map function:

$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

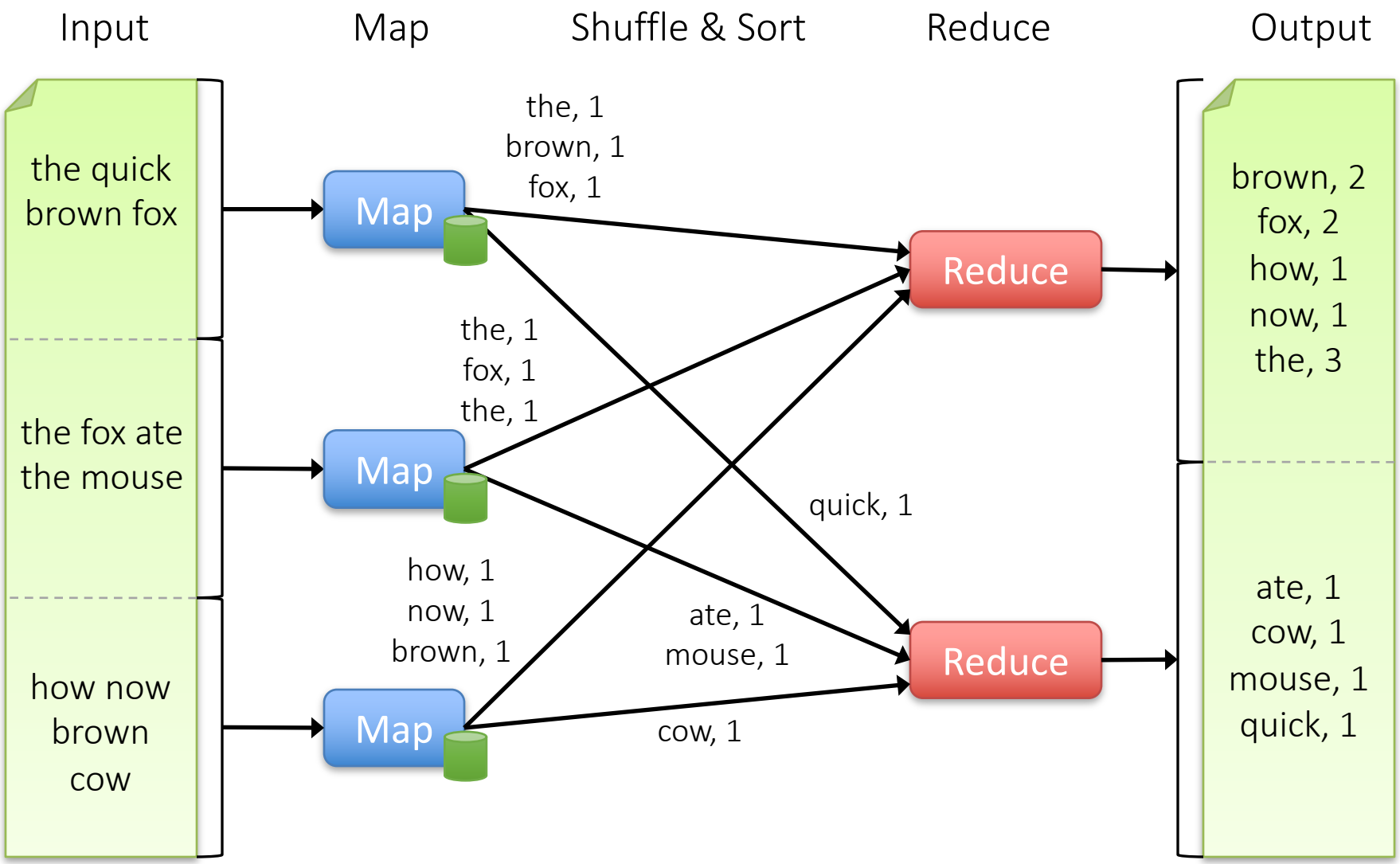
Reduce function:

$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

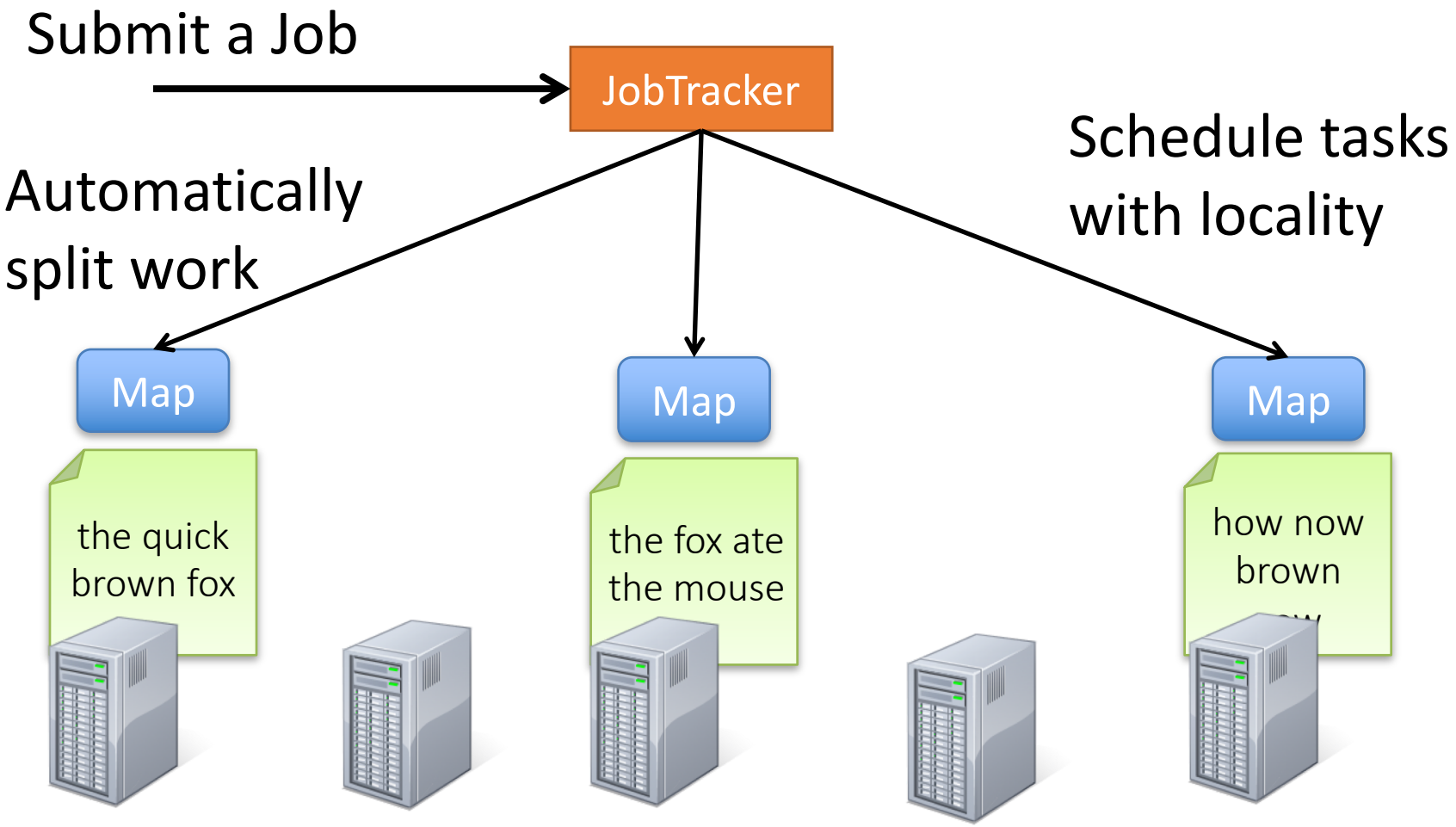
Example: Word Count

```
def mapper(line):
    for word in line.split():
        output(word, 1)


def reducer(key, values):
    output(key, sum(values))
```

Word Count Execution



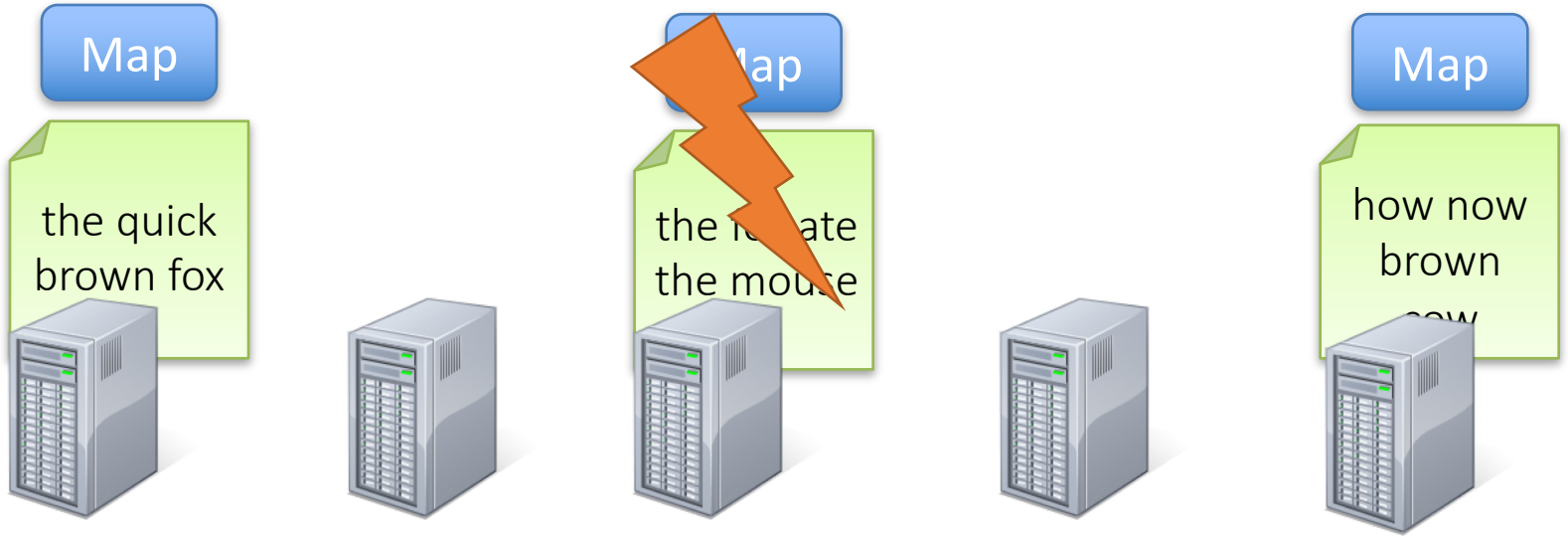
Word Count Execution



Fault Recovery

If a task crashes:

- Retry on another node
- If the same task repeatedly fails, end the job



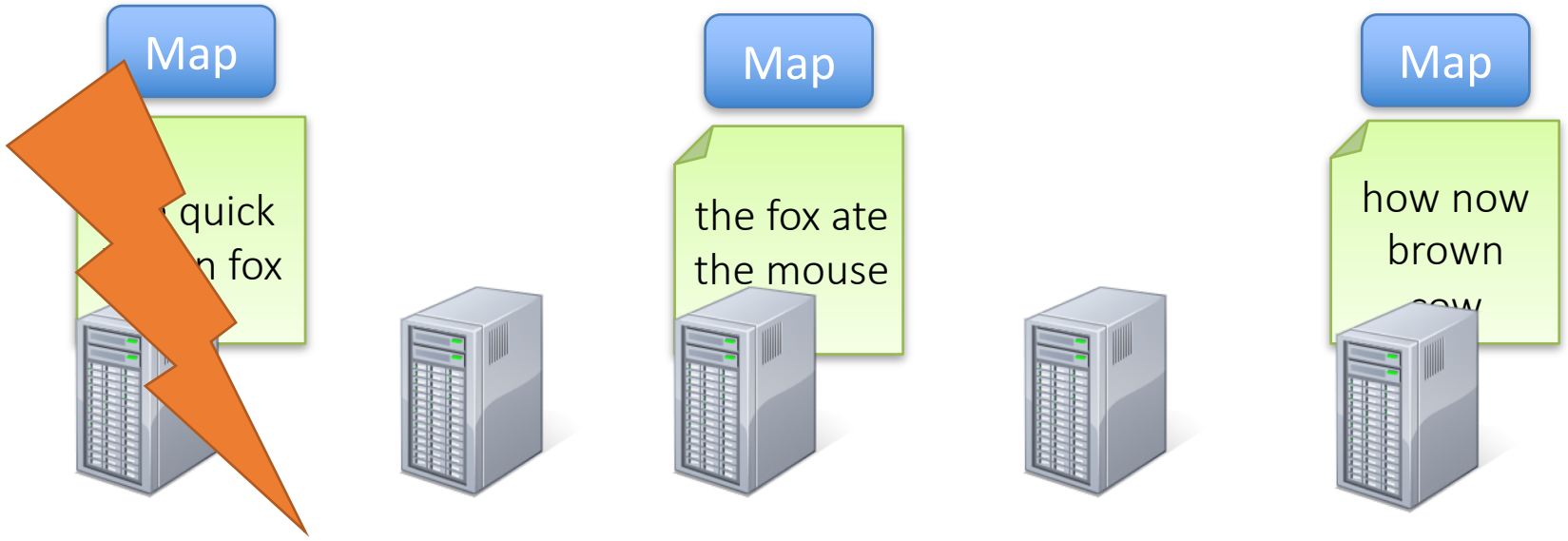
Requires user code to be **deterministic**

Fault Recovery

If a node crashes:

- Relaunch its current tasks on other nodes

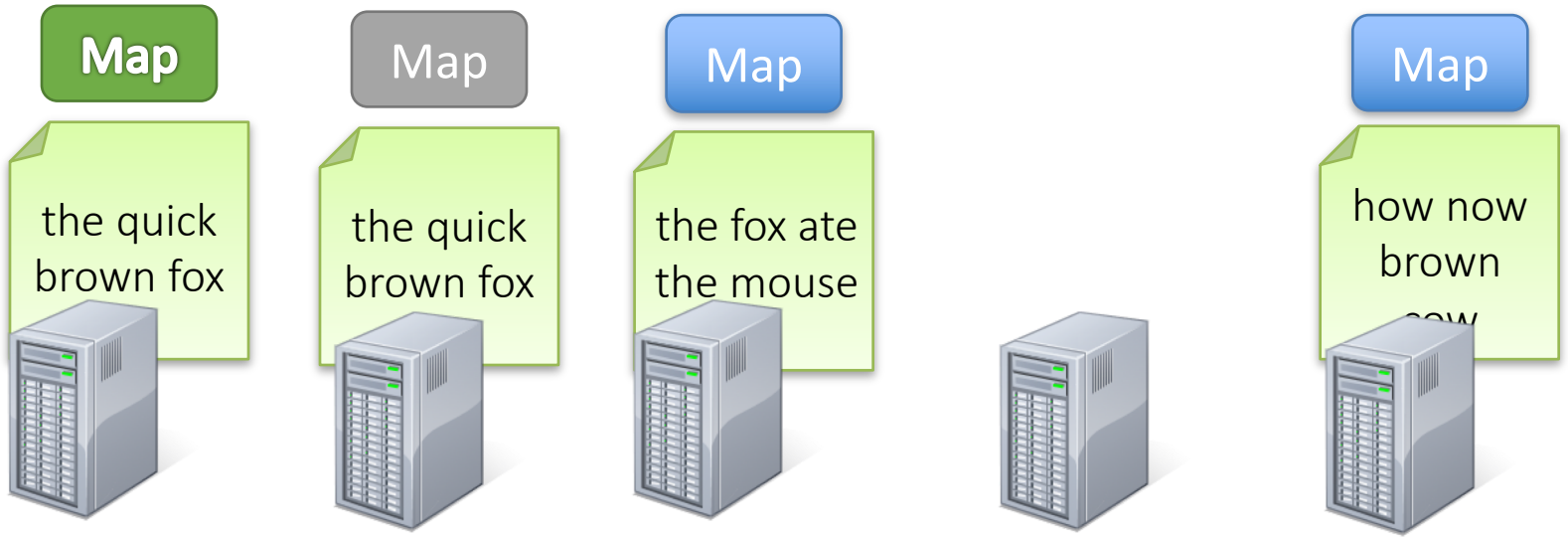
  What about task inputs ? File system replication



Fault Recovery

If a task is going slowly (straggler):

- Launch second copy of task on another node
- Take the output of whichever finishes first



What is Apache Hadoop?

- Open source software framework designed for storage and processing of large scale data on clusters of commodity hardware.

Who Uses Hadoop?

Hadoop uses HDFS

- The Hadoop Distributed File System (HDFS) is a specially designed file system for storing huge dataset switch cluster of commodity HW (low-end hardware) with streaming access.

- Normal file system
  - The file system is split into small blocks (e.g. 4KB), unused space in a block is wasted!
- Hadoop Distributed file system
  - On top of the OS and the normal file system. Blocks are large (64MB), unused space in a block will not go to waste.
  - Why does HDFS have a large block size?

Hadoop uses HDFS

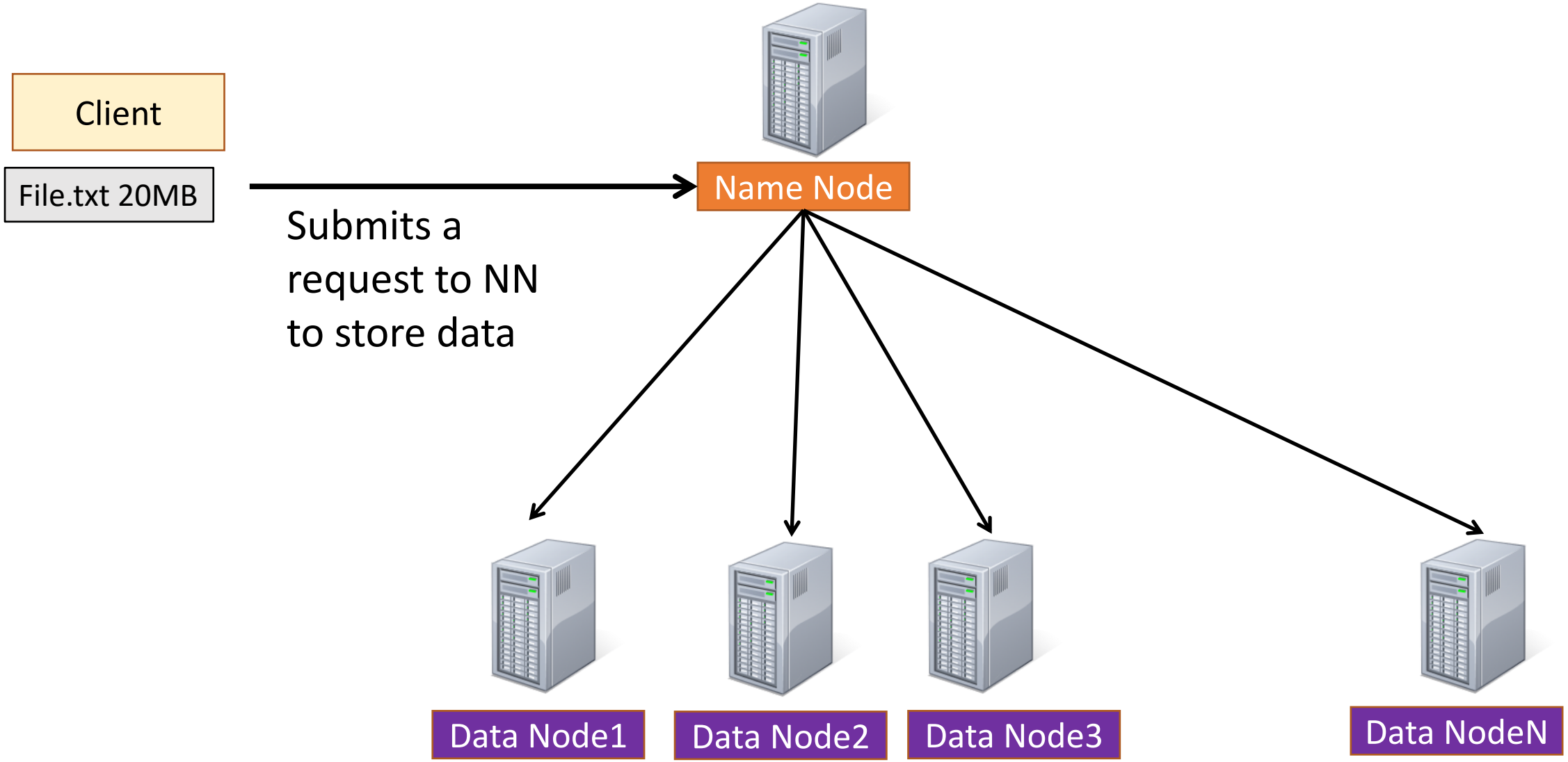
- The Hadoop Distributed File System (HDFS) is a specially designed file system for storing huge dataset switch cluster of commodity HW (low-end hardware) with streaming access.

- Normal file system
  - The file system is split into small blocks (e.g. 4KB), unused space in a block is wasted!
- Hadoop Distributed file system
  - On top of the OS and the normal file system. Blocks are large (64MB), unused space in a block will not go to waste.
  - Why does HDFS have a large block size? Large amount of data to store and expensive to store metadata.

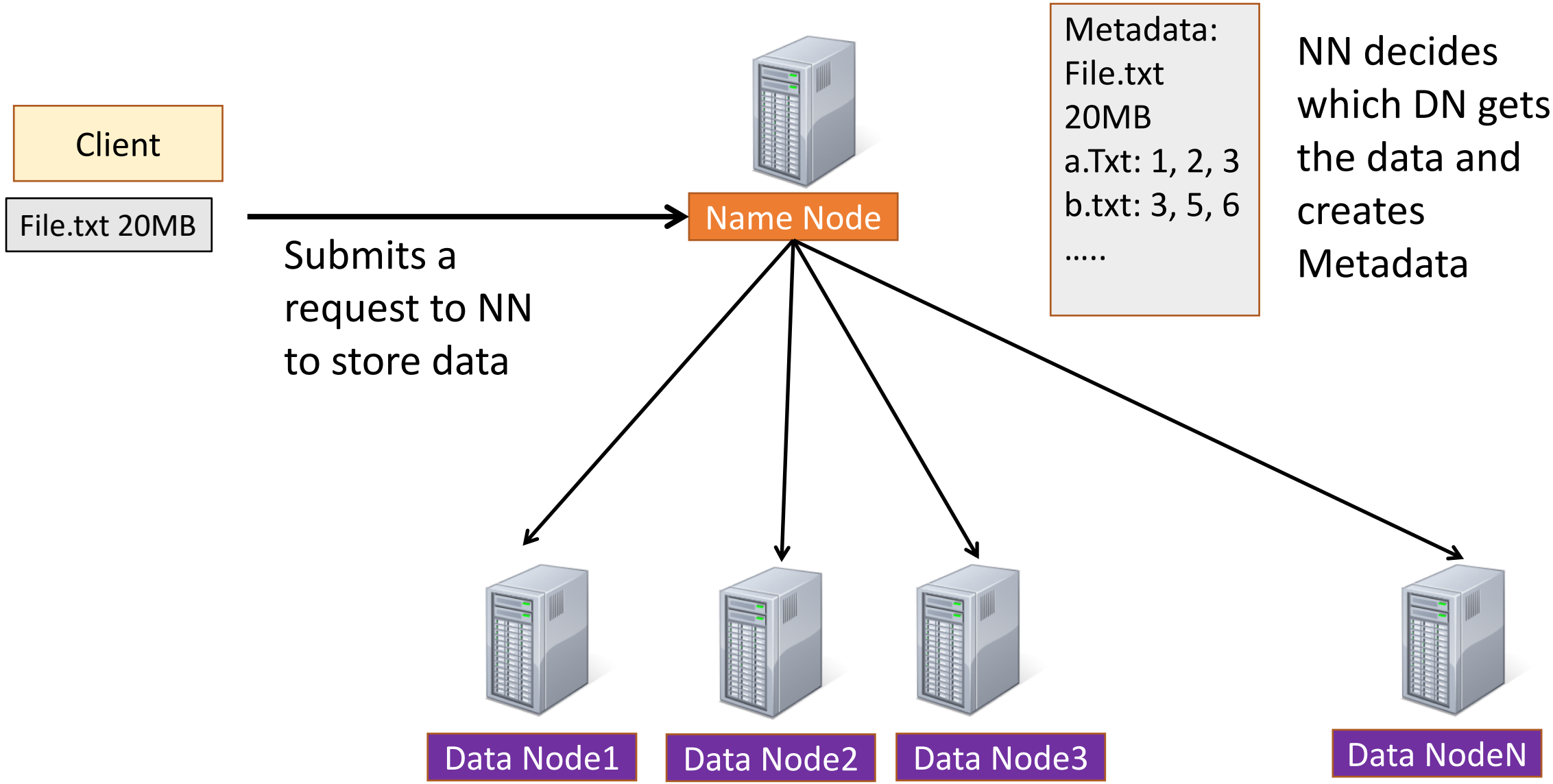
Main services on HDFS

- Master services: NameNode, Secondary NameNode, Job Tracker
- Slave services: Data Node and Task Tracker
- Master services can talk together
- Slave services can also talk together
- Job Tracker and Task Tracker can also communicate

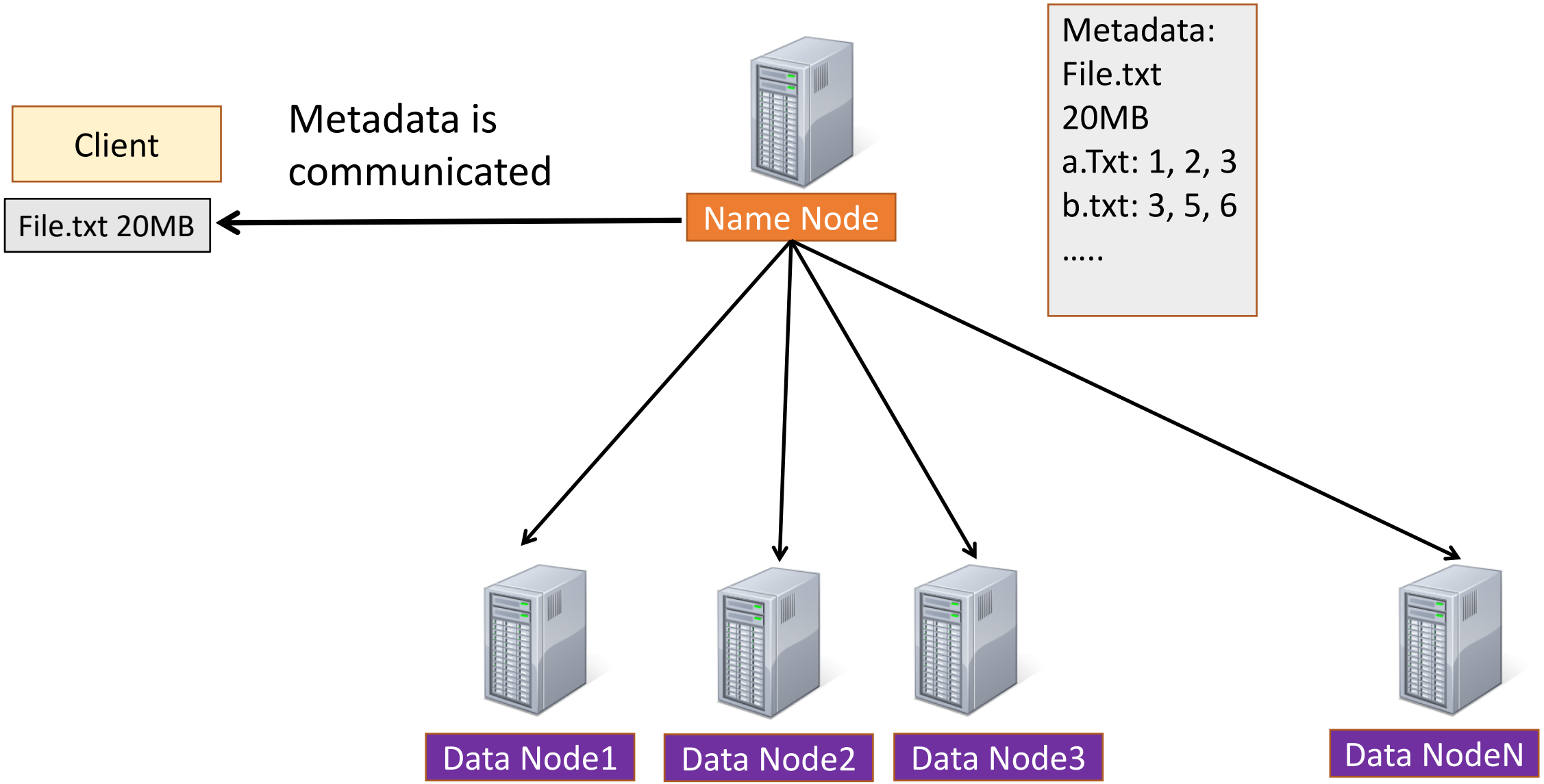
HDFS: Store Data



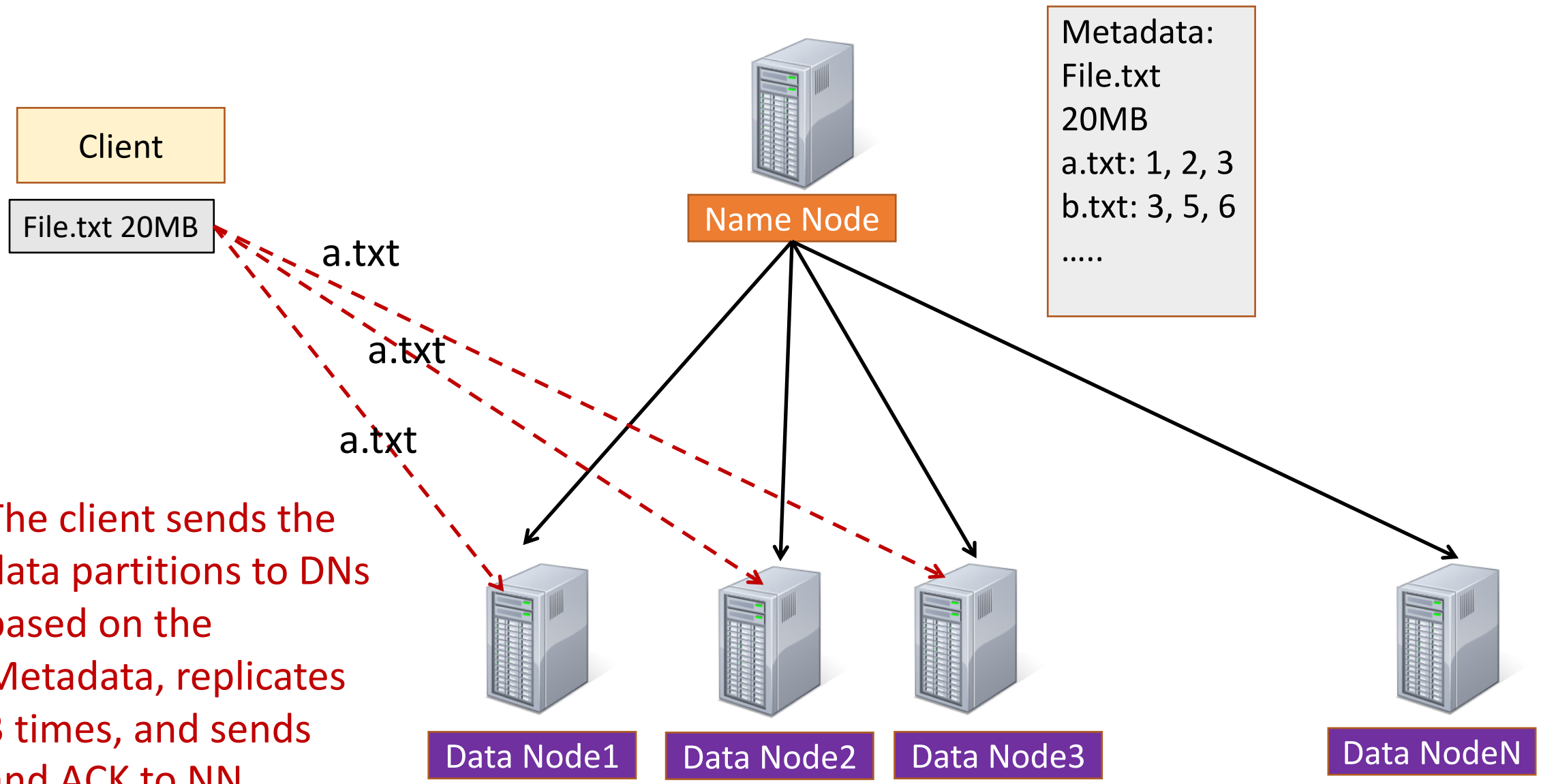
HDFS: Store Data



HDFS: Store Data



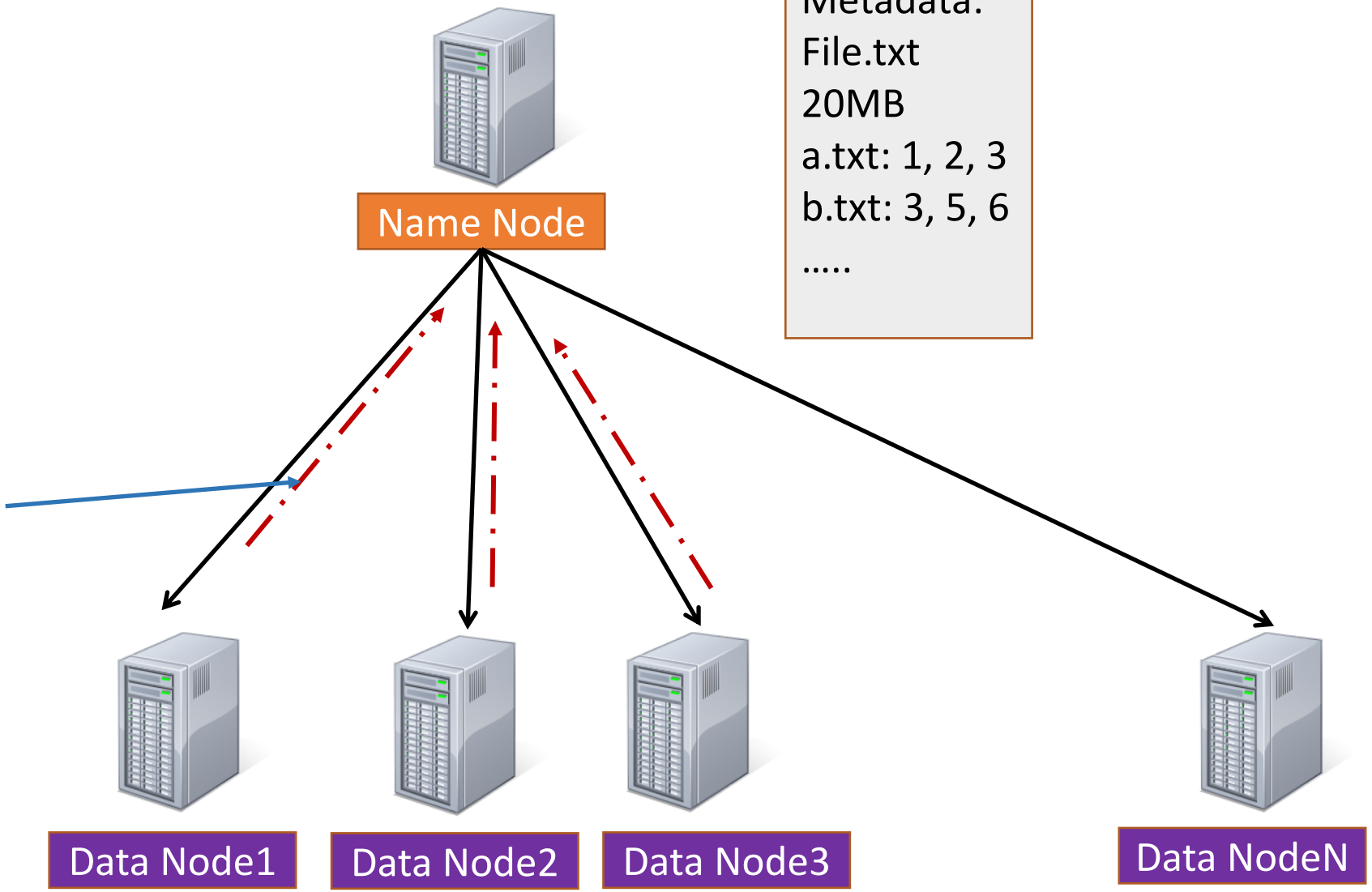
HDFS: Store Data



The client sends the data partitions to DNs based on the Metadata, replicates 3 times, and sends and ACK to NN

HDFS: Store Data

Client



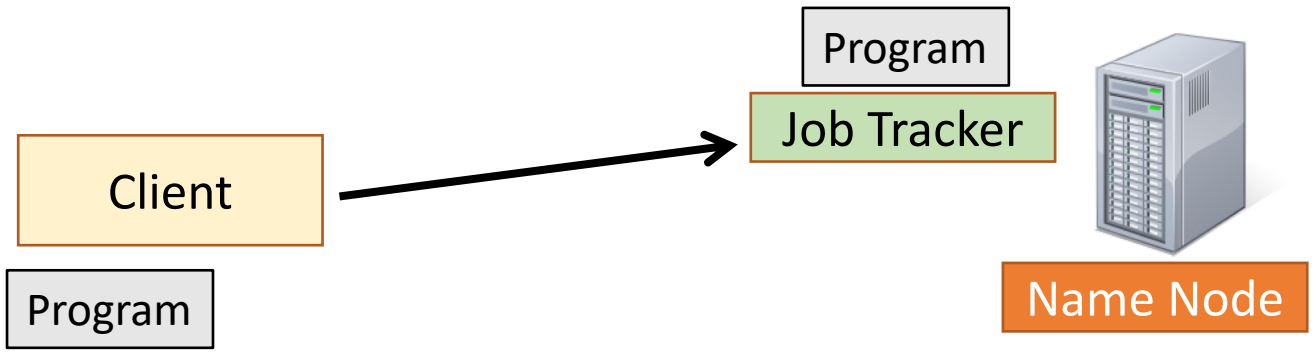
Node Failures

- If a Data Node fails it will not send heartbeats to NN. The Name Node removes related Metadata, replicates the lost data in another DN, and updates metadata.
- What if a Name Node fails?

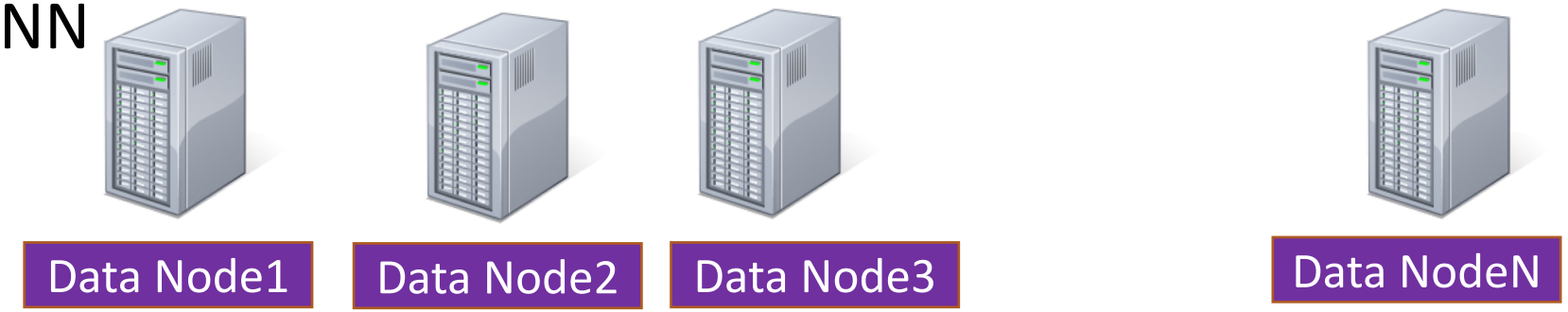
Node Failures

- If a Data Node fails it will not send heartbeats to NN. The Name Node removes related Metadata, replicates the lost data in another DN, and updates metadata.
- What if a Name Node fails? Everything is lost: Single point of failure. Have to use a reliable node for NN.

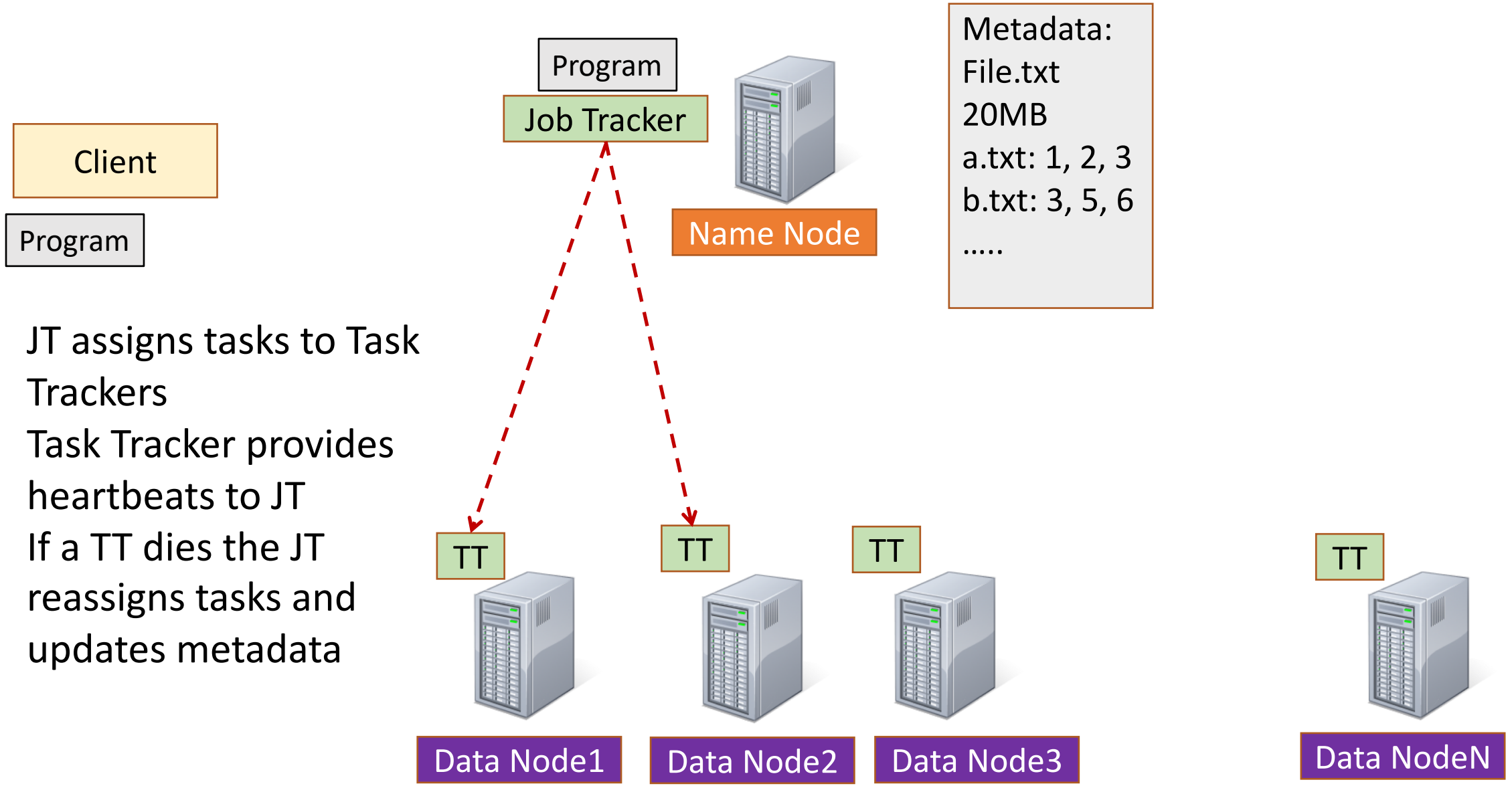
HDFS: Execute Tasks



- Client submits the program to the Job Tracker
- JT gets the metadata from NN



HDFS: Execute Tasks



- JT assigns tasks to Task Trackers
- Task Tracker provides heartbeats to JT
- If a TT dies the JT reassigns tasks and updates metadata

Performance of MapReduce (MR) with Hadoop

MR in “standard” Hadoop only provides one pass of computation

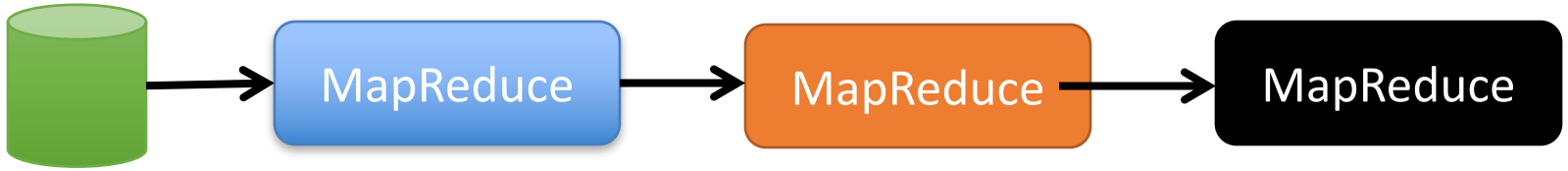
- Must write out data to file system in-between

Expensive for apps that need to *reuse* data

- Multi-step algorithms (e.g. PageRank)
- Interactive data mining

Multiple MRs

People want to compose it!



Most real applications require multiple MR steps

- Google indexing pipeline: 21 steps
- Analytics queries (e.g. sessions, top K): 2-5 steps
- Iterative algorithms (e.g. PageRank): 10’s of steps

Example of MR reuse: Page Rank

Thanks to Dan Teague and Matei Zaharia

Page Rank

- Imagine a library containing 40 billion documents but with no centralized organization and no librarians.
- In addition, anyone may add a document at any time without telling anyone.
- If one of these documents is vitally important to you, how could you find it?

Why This Order?

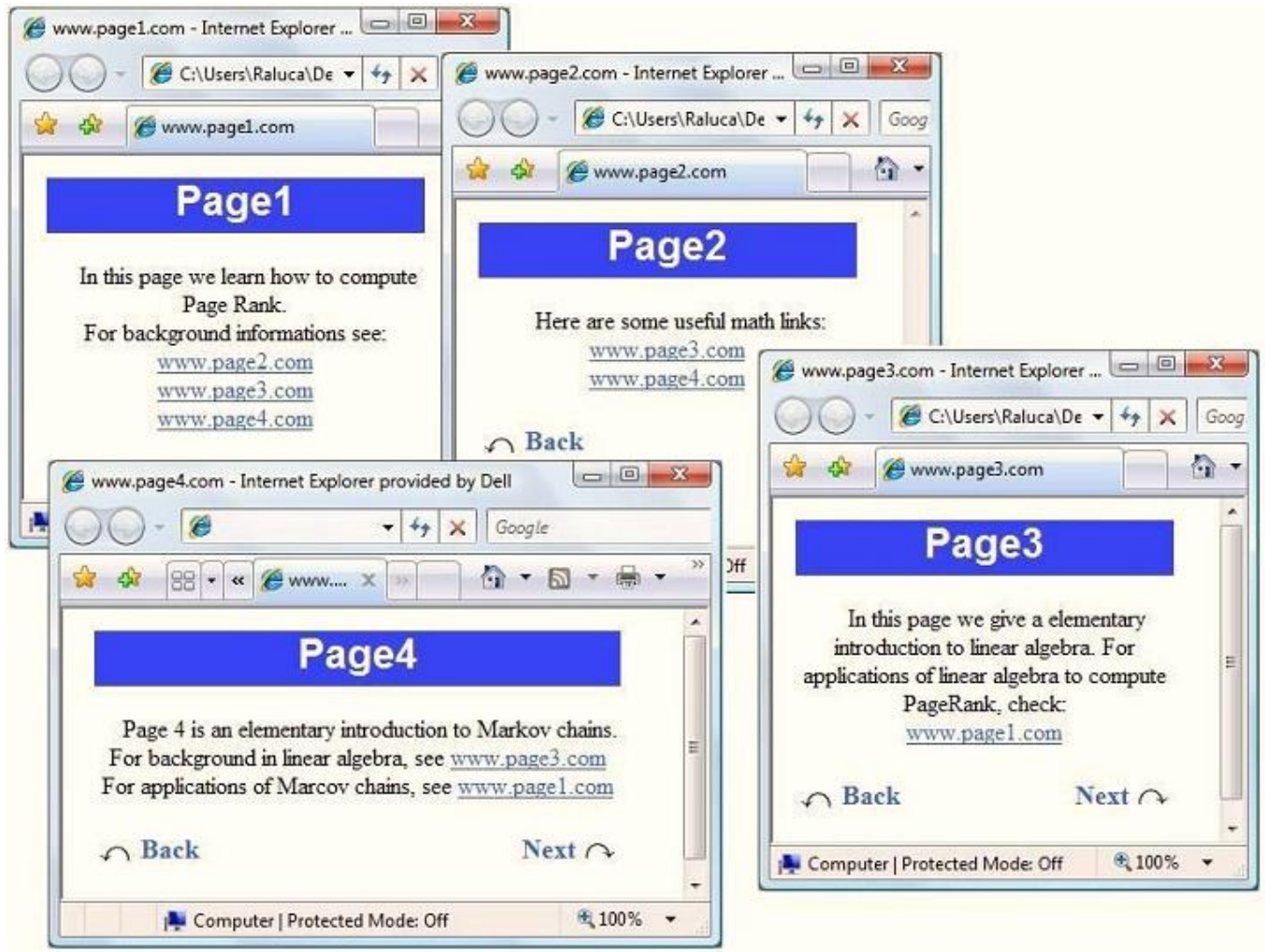
Google Pagerank System

Google was developed by Sergey Brin and Larry Page

This is the method that Larry Page developed to rank and order the pages.

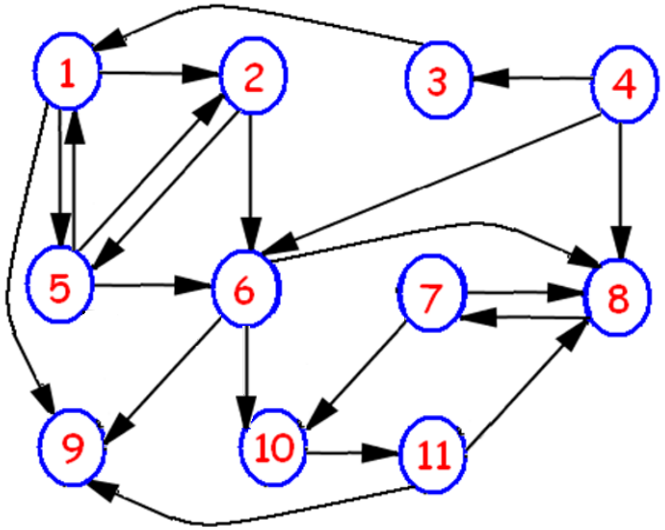
Hence, the Pagerank.

Page Rank


www.page1.com - Internet Explorer ...
C:\Users\Raluca\De
www.page1.com
Page1
In this page we learn how to compute Page Rank.
For background informations see:
www.page2.com
www.page3.com
www.page4.com
www.page2.com - Internet Explorer ...
C:\Users\Raluca\De
www.page2.com
Page2
Here are some useful math links:
www.page3.com
www.page4.com
Back
www.page3.com - Internet Explorer ...
C:\Users\Raluca\De
www.page3.com
Page3
In this page we give a elementary introduction to linear algebra. For applications of linear algebra to compute PageRank, check:
www.page1.com
Back
Next
Computer | Protected Mode: Off
100%
www.page4.com - Internet Explorer provided by Dell
Google
Page4
Page 4 is an elementary introduction to Markov chains.
For background in linear algebra, see www.page3.com
For applications of Marcov chains, see www.page1.com
Back
Next
Computer | Protected Mode: Off
100%

How would you order these site?

- Suppose each of the nodes at right have the links shown in the directed graph. Which node is most important and should appear first?



The Basic Idea

- PageRank is a numeric value that represents how important a page is on the web. Google figures that when one page links to another page, it is effectively casting a vote for the other page. The more votes that are cast for a page, the more important the page must be. Also, the importance of the page that is casting the vote determines how important the vote itself is. http://www.webworkshop.net/pagerank.html

The Basic Idea

- PageRank value for a page u is dependent on the PageRank values for each page v out of the set Bu (all pages linking to page u), divided by the number L(v) of links from page v

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

The Basic Idea

- Phase 1: Propagation
- Phase 2: Aggregation
- Input: A pool of objects, including both vertices and edges

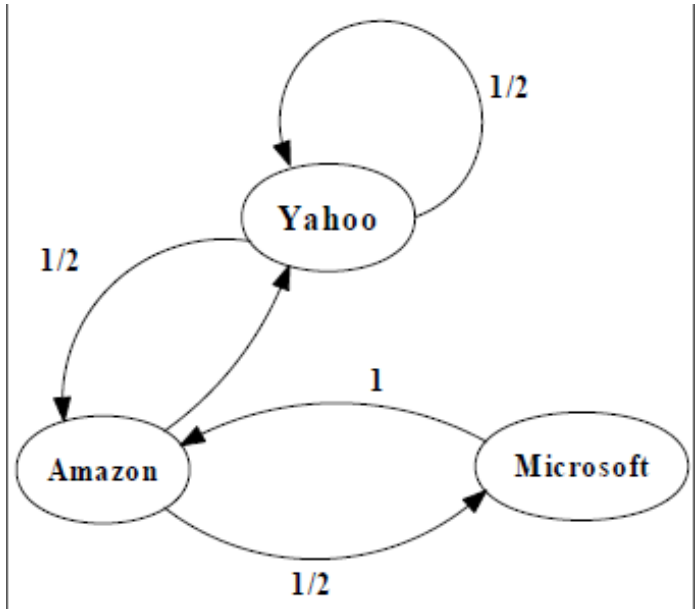
Propagation

- Map: for each object
  - If object is vertex, emit key=URL, value=object
  - If object is edge, emit key=source URL, value=object
- Reduce: (input is a web page and all the outgoing links)
  - Find the number of edge objects->outgoing links
  - Read the PageRank value from the vertex object
  - Assign PR(edges)=PR(vertex)/num_outgoing

Aggregation

- Map: for each object
  - If object is vertex, emit key=URL, value=object
  - If object is edge, emit key=destination URL, value=object
- Reduce: (input is a web page and all the incoming links)
  - Add the PR value of all incoming links
  - Assign PR(vertex)=ΣPR(incoming links)

Using matrices



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

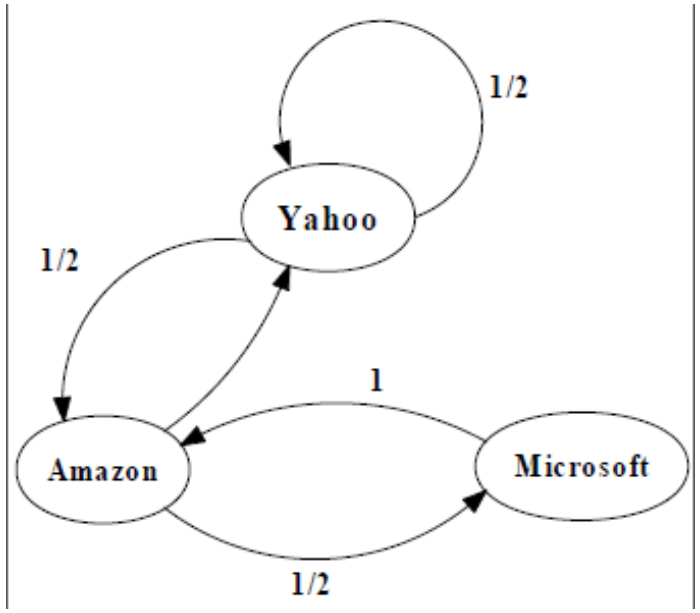
$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 1/3 \\ 1/2 \\ 1/6 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

PageRank Calculation: first iteration

Source: Fei Li

Using matrices



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

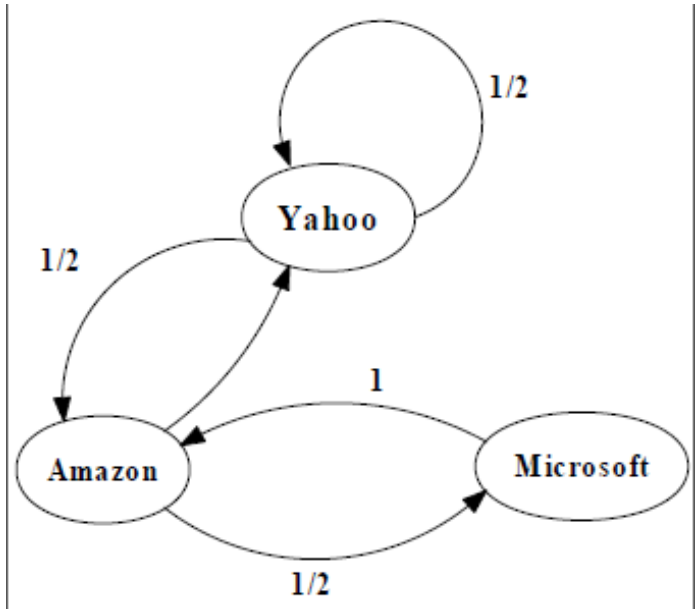
$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 5/12 \\ 1/3 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/2 \\ 1/6 \end{bmatrix}$$

PageRank Calculation: second iteration

Source: Fei Li

Using matrices



$$M = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 0 & 1 \\ 0 & 1/2 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \text{yahoo} \\ \text{Amazon} \\ \text{Microsoft} \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}$$

$$\begin{bmatrix} 3/8 \\ 11/24 \\ 1/6 \end{bmatrix} \begin{bmatrix} 5/12 \\ 17/48 \\ 11/48 \end{bmatrix} \ldots \begin{bmatrix} 2/5 \\ 2/5 \\ 1/5 \end{bmatrix}$$

Convergence after some iterations

Source: Fei Li

Page Rank as Matrix Multiplication

- Rank of each page is the probability of landing on that page for a random surfer on the web
- Probability of visiting all pages after k steps is

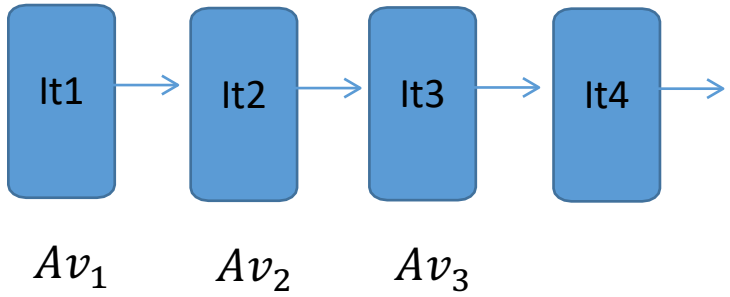
$$V_k = A^k \times V^t$$

V: the initial rank vector

A: the link structure (sparse matrix)

Iterative algorithms such as PageRank

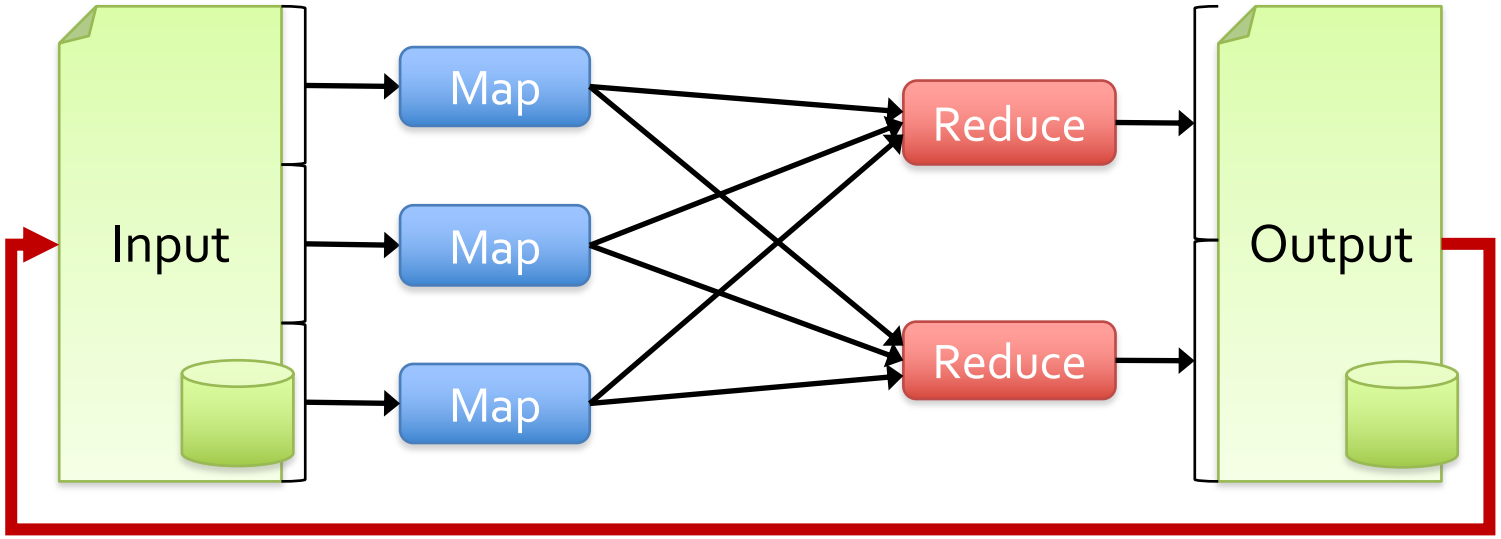
```
Iterate {
   Map: for (each) i = 1 to M  Compute();
   Reduce();
} Until converged();
```



- Repeated reads of constant (input) data in each iteration
- Run-time overheads in each iteration
- Intermediate Communication resulting from model updates

Retrofitting iterations to MapReduce

- MR does not support iteration out of the box
- The “obvious” solution: Split iteration into multiple MapReduce jobs. Write a driver program for orchestration.
- Read and writes to HDFS can take over 90 percent of the time for iterative algorithms!



Solution: Spark

Matei Zaharia
https://cs.stanford.edu/~matei/

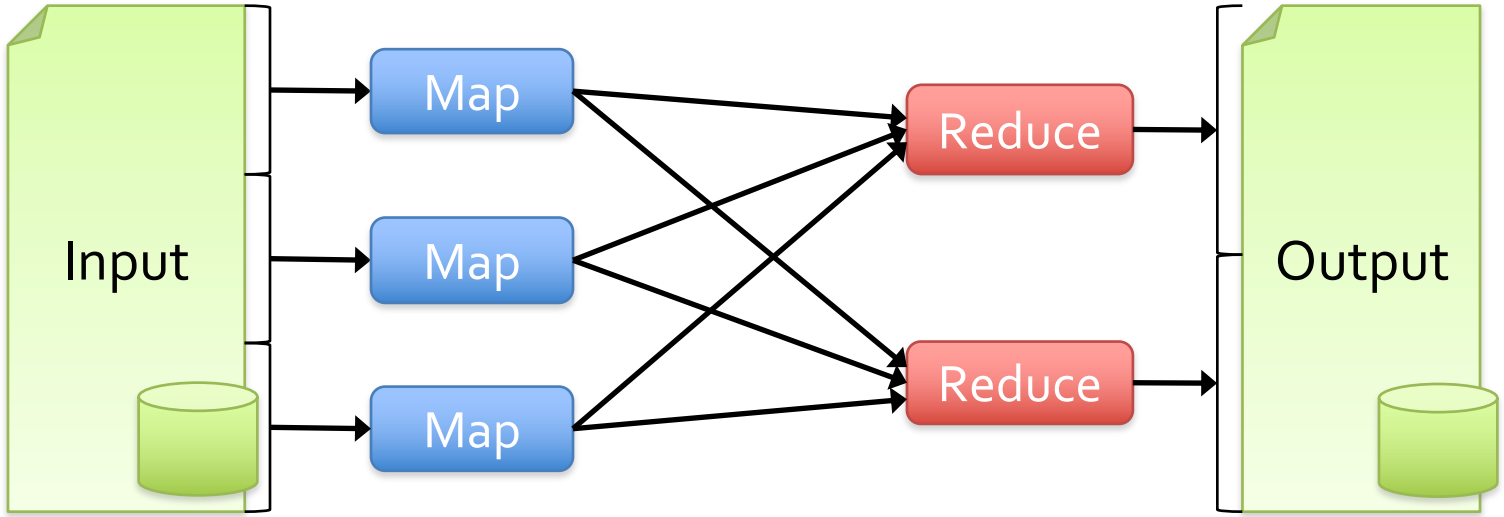
MIT CSAIL

amplab
UC Berkeley

Motivation

Most popular programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:



Motivation

Acyclic data flow is a powerful abstraction, but is not efficient for applications that repeatedly reuse a *working set* of data:

- **Iterative** algorithms (many in machine learning)
- **Interactive** data mining tools (R, Excel, Python)

Spark makes working sets a first-class concept to efficiently support these apps

What is Spark

- Fast, expressive cluster computing system compatible with Apache Hadoop
  - Works with any Hadoop-supported storage system (HDFS, S3, Avro, …)
- Improves **efficiency** through:
  - In-memory computing primitives
  - General computation graphs

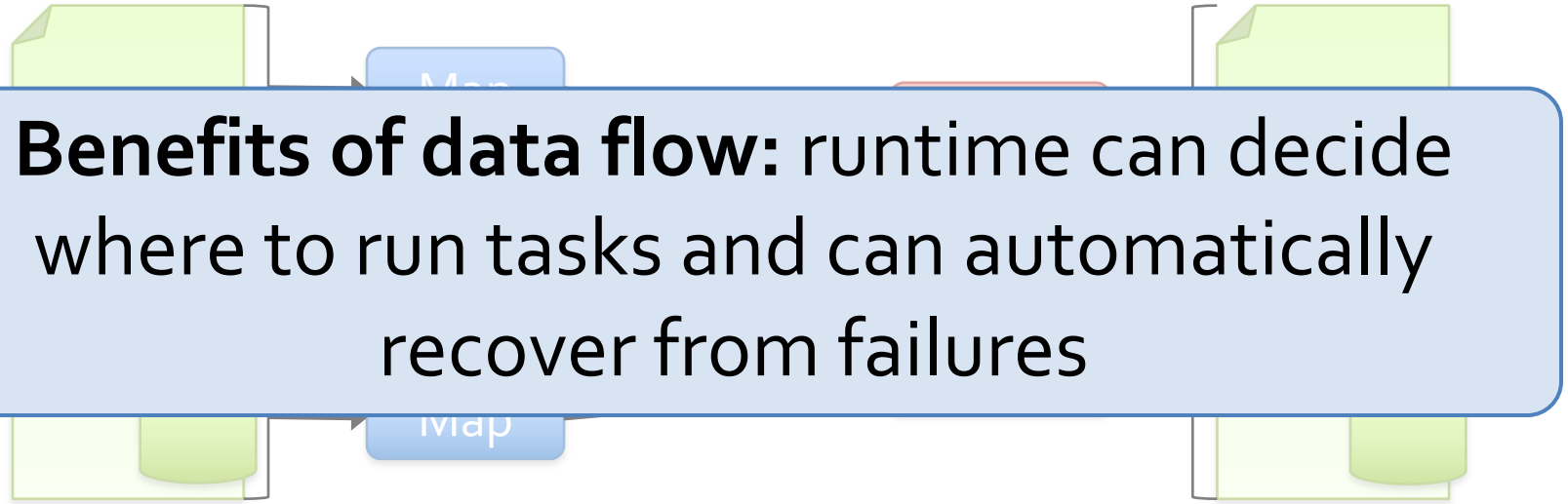
  → Up to 100× faster
- Improves **usability** through:
  - Rich APIs in Java, Scala, Python
  - Interactive shell

  → Often 2-10× less code

Motivation

Current popular programming models for clusters transform data flowing from stable storage to stable storage

E.g., MapReduce:



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

Spark Goal

Provide distributed memory abstractions for clusters to support apps with working sets, build a directed acyclic graph (DAG)

Retain the attractive properties of MapReduce:

- » Fault tolerance (for crashes & stragglers)
- » Data locality
- » Scalability

Solution: augment data flow model with "resilient distributed datasets" (RDDs)

Programming Model

Resilient distributed datasets (RDDs): now replaced by Datasets

- Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
- Can be *cached* across parallel operations
- Immutable collections partitioned across cluster that can be rebuilt if a partition is lost

Parallel operations on RDDs

- Reduce, collect, count, save, …

Restricted shared variables

- Accumulators, broadcast variables

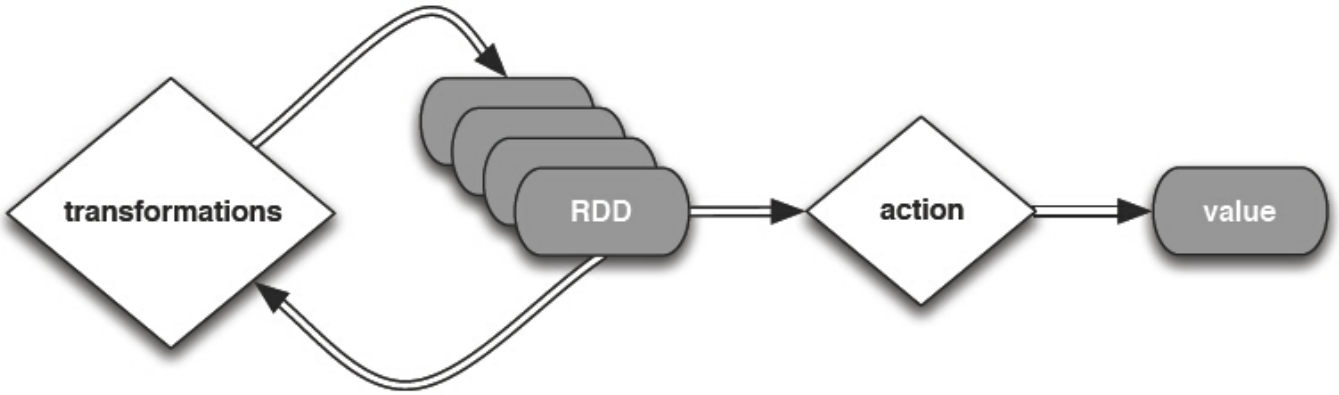
Operations

Transformations (e.g. map, filter, groupBy, join)

» Lazy operations to build RDDs from other RDDs

Actions (e.g. count, collect, save)

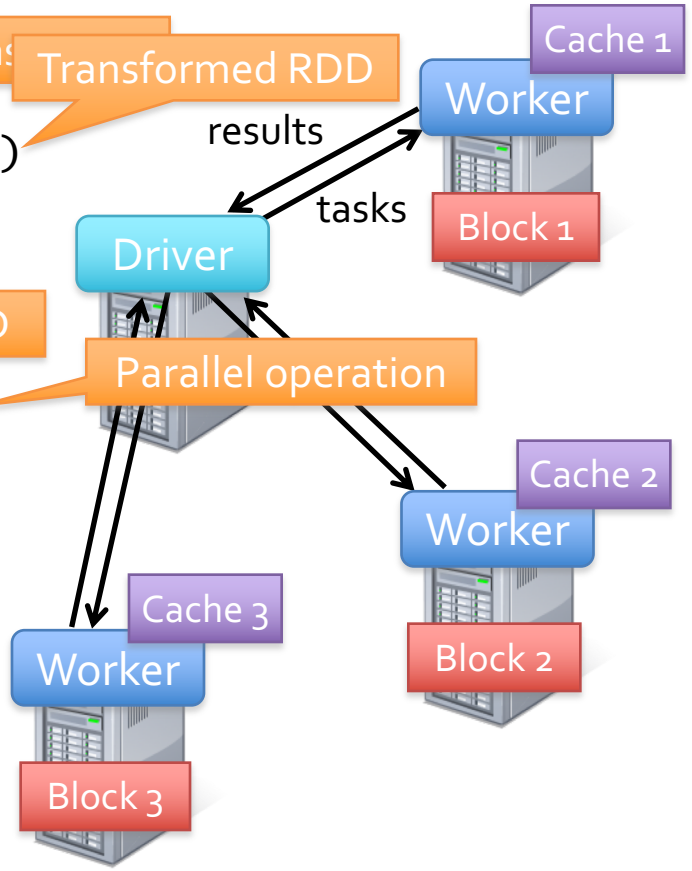
» Return a result or write it to storage



Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```



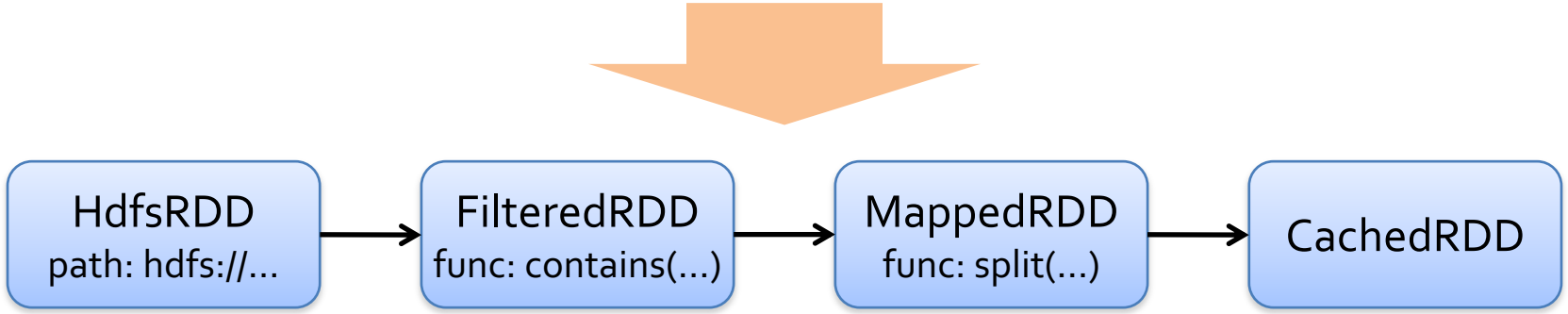
Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

RDD lineage

RDDs maintain *lineage* information

Ex:

```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```

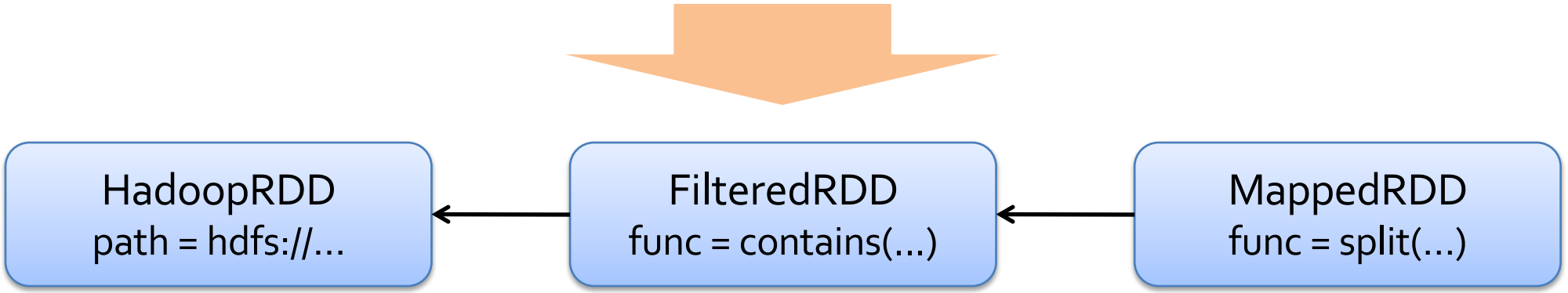


RDD Fault Tolerance

RDDs track the transformations used to build them (their *lineage*) to recompute lost data

E.g:

```
messages = textFile(...).filter(lambda s: s.contains("ERROR"))
                        .map(lambda s: s.split('\t')[2])
```



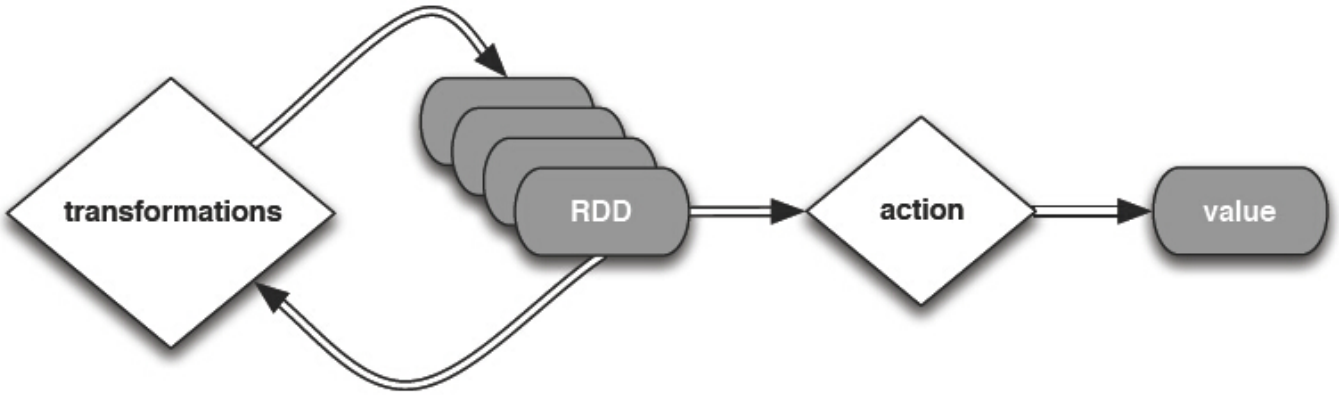
Operations

Transformations (e.g. map, filter, groupBy, join)

» Lazy operations to build RDDs from other RDDs

Actions (e.g. count, collect, save)

» Return a result or write it to storage



RDD Operations

Transformations (define a new RDD)	**Parallel operations** (return a result to driver)
map filter sample union groupByKey reduceByKey join cache …	reduce collect count save lookupKey …

Spark Essentials: *Transformations*

Transformations create a new dataset from an existing one

All transformations in Spark are *lazy*: they do not compute their results right away – instead they remember the transformations applied to some base dataset

- optimize the required calculations
- recover from lost data partitions

Spark Essentials: *Transformations*

transformation	*description*
map(*func*)	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func*)	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func*)	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

Spark Essentials: *Transformations*

transformation	*description*
groupByKey([*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
reduceByKey(*func*, [*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*]**)**	when called on a dataset of `(K, V)` pairs where `K` implements `Ordered`, returns a dataset of `(K, V)` pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset*, [*numTasks*]**)**	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, (V, W))` pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*]**)**	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith`
cartesian(*otherDataset***)**	when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements)

Spark Essentials: *Actions*

action	*description*
`reduce(`*`func`***`)`**	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
`collect()`	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
`count()`	return the number of elements in the dataset
`first()`	return the first element of the dataset – similar to *take(1)*
`take(`*`n`***`)`**	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
`takeSample(`*`withReplacement, fraction, seed`***`)`**	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Spark Essentials: *Actions*

action	*description*
`saveAsTextFile(`*`path`***`)`**	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
`saveAsSequenceFile(`*`path`***`)`**	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
`countByKey()`	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
`foreach(`*`func`***`)`**	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

Benefits of RDD Model

Consistency is easy due to immutability

Inexpensive fault tolerance (log lineage rather than replicating/checkpointing data)

Locality-aware scheduling of tasks on partitions

Despite being restricted, model seems applicable to a broad variety of applications

First Stop: SparkContext

Main entry point to Spark functionality

Created for you in Spark shells as variable `sc`

In standalone programs, you'd make your own

Spark in Java and Scala

Java API:

```
JavaRDD<String> lines = spark.textFile(…);

errors = lines.filter(
  new Function<String, Boolean>() {
    public Boolean call(String s) {
      return s.contains(“ERROR”);
    }
});

errors.count()
```

Scala API:

```
val lines = spark.textFile(…)

errors = lines.filter(s => s.contains(“ERROR”))
// can also write filter(_.contains(“ERROR”))

errors.count
```

Creating RDDs

```
# Turn a local collection into an RDD
sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
sc.textFile(“file.txt”)
sc.textFile(“directory/*.txt”)
sc.textFile(“hdfs://namenode:9000/path/file”)

# Use any existing Hadoop InputFormat
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Basic Transformations

```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)   # => {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) # => {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(0, x))  # => {0, 0, 1, 0, 1, 2}
```

Lambda Functions

- Shorthand version of def statement, useful for “inlining” functions and other situations where it's convenient to keep the code of the function close to where it's needed
- Can only contain an expression in the function definition, not a block of statements (e.g., no if statements, etc)
- A lambda returns a function; the programmer can decide whether or not to assign this function to a name

Lambda Functions

- **Simple example:**

```
>>> def sum(x,y): return x+y
>>> ...
>>> sum(1,2)
3


>>> sum2 = lambda x, y: x+y
>>> sum2(1,2)
3
```

Basic Actions

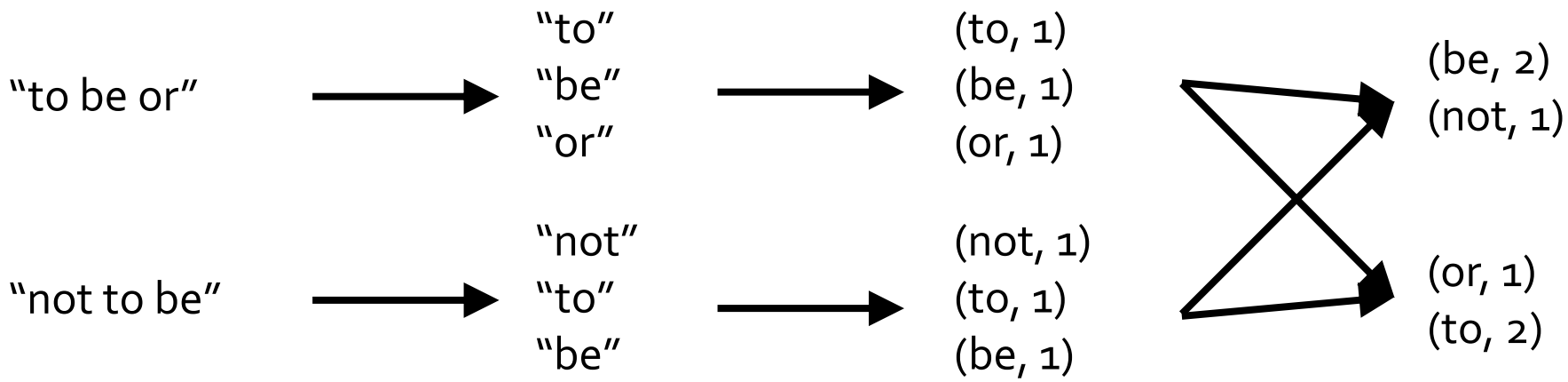
```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
nums.saveAsTextFile(“hdfs://file.txt”)
```

Some Key-Value Operations

```
pets = sc.parallelize([(“cat”, 1), (“dog”, 1), (“cat”, 2)])

pets.reduceByKey(lambda x, y: x + y)
# => {(cat, 3), (dog, 1)}

pets.groupByKey()
# => {(cat, Seq(1, 2)), (dog, Seq(1)}

pets.sortByKey()
# => {(cat, 1), (cat, 2), (dog, 1)}
```

Example: Word Count

```
lines = sc.textFile(“hamlet.txt”)

counts = lines.flatMap(lambda line: line.split(“ ”)) \
              .map(lambda word: (word, 1)) \
              .reduceByKey(lambda x, y: x + y)
```



Multiple Datasets

```
visits = sc.parallelize([(“index.html”, “1.2.3.4”),
                         (“about.html”, “3.4.5.6”),
                         (“index.html”, “1.3.3.1”)])

pageNames = sc.parallelize([(“index.html”, “Home”), (“about.html”, “About”)])

visits.join(pageNames)
# (“index.html”, (“1.2.3.4”, “Home”))
# (“index.html”, (“1.3.3.1”, “Home”))
# (“about.html”, (“3.4.5.6”, “About”))

visits.cogroup(pageNames)
# (“index.html”, (Seq(“1.2.3.4”, “1.3.3.1”), Seq(“Home”)))
# (“about.html”, (Seq(“3.4.5.6”), Seq(“About”)))
```

Controlling the Level of Parallelism

All the pair RDD operations take an optional second parameter for number of tasks

```
words.reduceByKey(lambda x, y: x + y, 5)
words.groupByKey(5)
visits.join(pageViews, 5)
```

Word Count in Spark

```
val lines = spark.textFile(“hdfs://...”)

val counts = lines.flatMap(_.split(“\\s”))
                  .reduceByKey(_ + _)

counts.save(“hdfs://...”)
```

Data Partitioning

- Spark's partitioning is available on all RDDs of key/value pairs
- Although Spark does not give explicit control of which worker node each key goes to (partly because the system is designed to work even if specific nodes fail), it lets the program ensure that a *set* of keys will appear together on *some* node.

Thanks to: *Learning Spark: Lightning-Fast Big Data Analysis!*

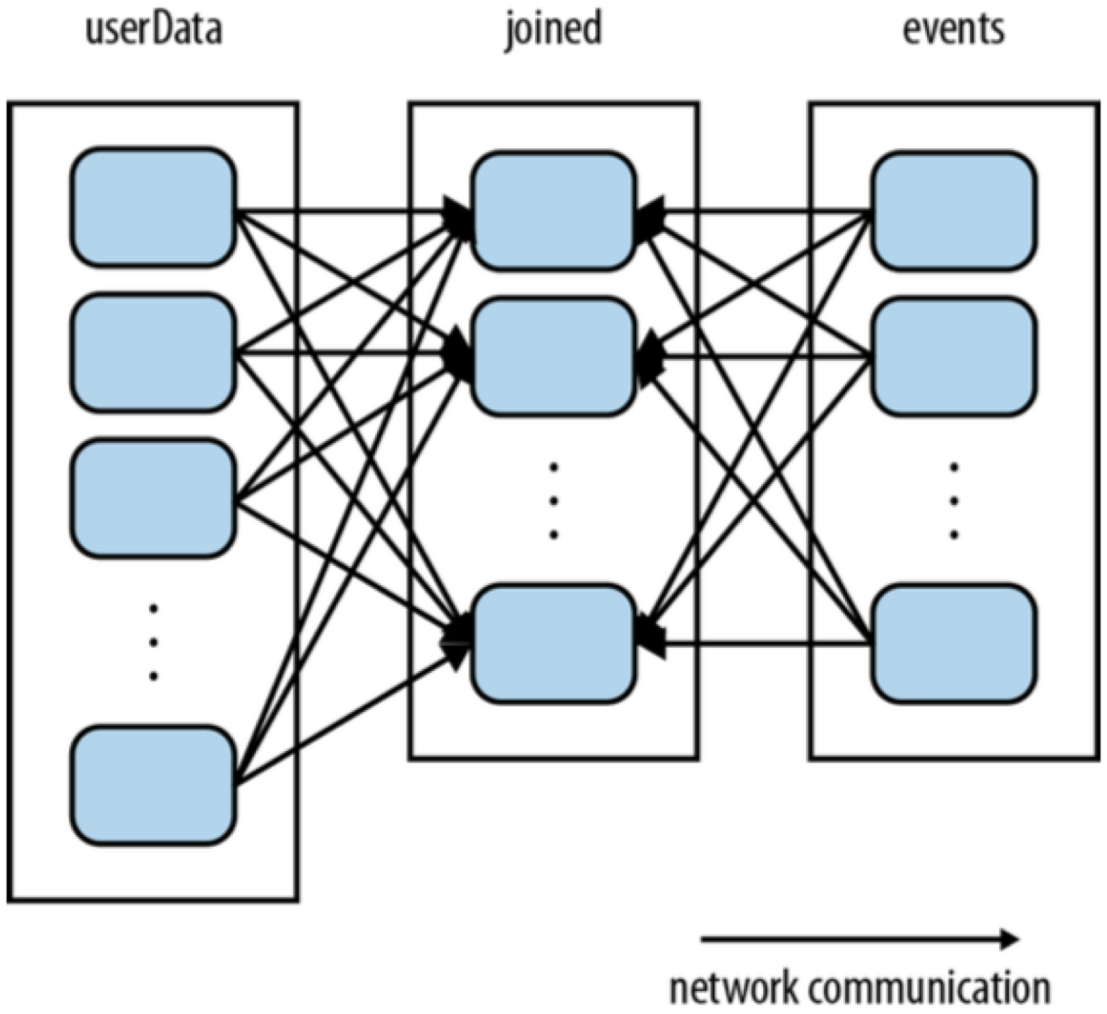
Example for Data Partitioning

- Consider an RDD of (UserID, UserInfo) pairs, where UserInfo contains a list of topics the user is subscribed to.
- A second RDD which is a table of (UserID, LinkInfo) pairs for users who have clicked a link on a website in those five minutes
- Count how many users visited a link that was *not* to one of their subscribed topics

Example for Data Partitioning

```
// Initialization code; we load the user info from a Hadoop SequenceFile on HDFS.
// This distributes elements of userData by the HDFS block where they are found,
// and doesn't provide Spark with any way of knowing in which partition a
// particular UserID is located.
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

// Function called periodically to process a logfile of events in the past 5 minutes;
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
  val joined = userData.join(events)// RDD of (UserID, (UserInfo, LinkInfo)) pairs
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Expand the tuple into its components
      !userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

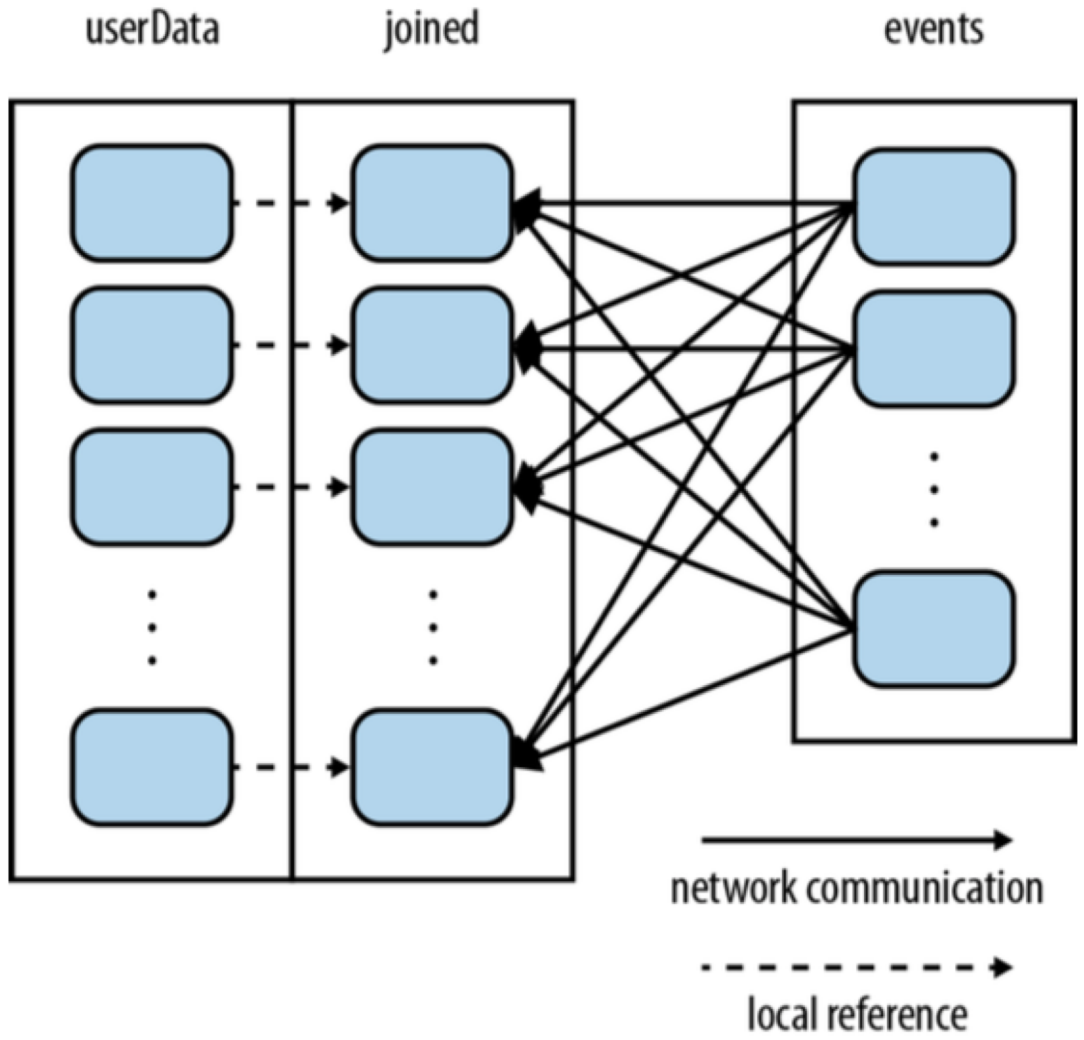
Example for Data Partitioning



Example for Data Partitioning

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
                 .partitionBy(new HashPartitioner(100))   // Create 100 partitions
                 .persist()

// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
  val joined = userData.join(events)// RDD of (UserID, (UserInfo, LinkInfo)) pairs
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Expand the tuple into its components
      !userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

Example for Data Partitioning



Example for Data Partitioning

```
scala> val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3)))
pairs: spark.RDD[(Int, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> pairs.partitioner
res0: Option[spark.Partitioner] = None

scala> val partitioned = pairs.partitionBy(new spark.HashPartitioner(2))
partitioned: spark.RDD[(Int, Int)] = ShuffledRDD[1] at partitionBy at <console>:14

scala> partitioned.partitioner
res1: Option[spark.Partitioner] = Some(spark.HashPartitioner@5147788d)
```

Operations that benefit from Partitioning

- cogroup(), groupWith(), join(), leftOuterJoin(), rightOuter Join(), groupByKey(), reduceByKey(), combineByKey(), and lookup()
- For operations working on a single RDD like reduceByKey() partitioning will enable local partitioning
- For others such as join() pre-partitioning at least one RDD keeps it partitioned. If both RDD have the same partitioning, both partitioning are preserved.

Operations that affect partitioning

- Transformations that *cannot* be guaranteed to produce a known partitioning, the output RDD will not have a partitioner set. For example, if you call map() on a hash-partitioned RDD of key/value pairs, the function passed to map() can in theory change the key of each element, so the result will not have a partitioner.
- Use mapValues() or flatMapValues() instead.

Operations with output partitioned

All the operations that result in a partitioner being set on the output RDD: cogroup(), groupWith(), join(), leftOuterJoin(), rightOuter Join(), groupByKey(), reduceByKey(), combineByKey(), partitionBy(), sort(), mapValues() (if the parent RDD has a partitioner), flatMapValues() (if parent has a partitioner), and filter() (if parent has a partitioner).

All *other* operations will produce a result with no partitioner.

Why are all transformations?

PageRank Algorithm

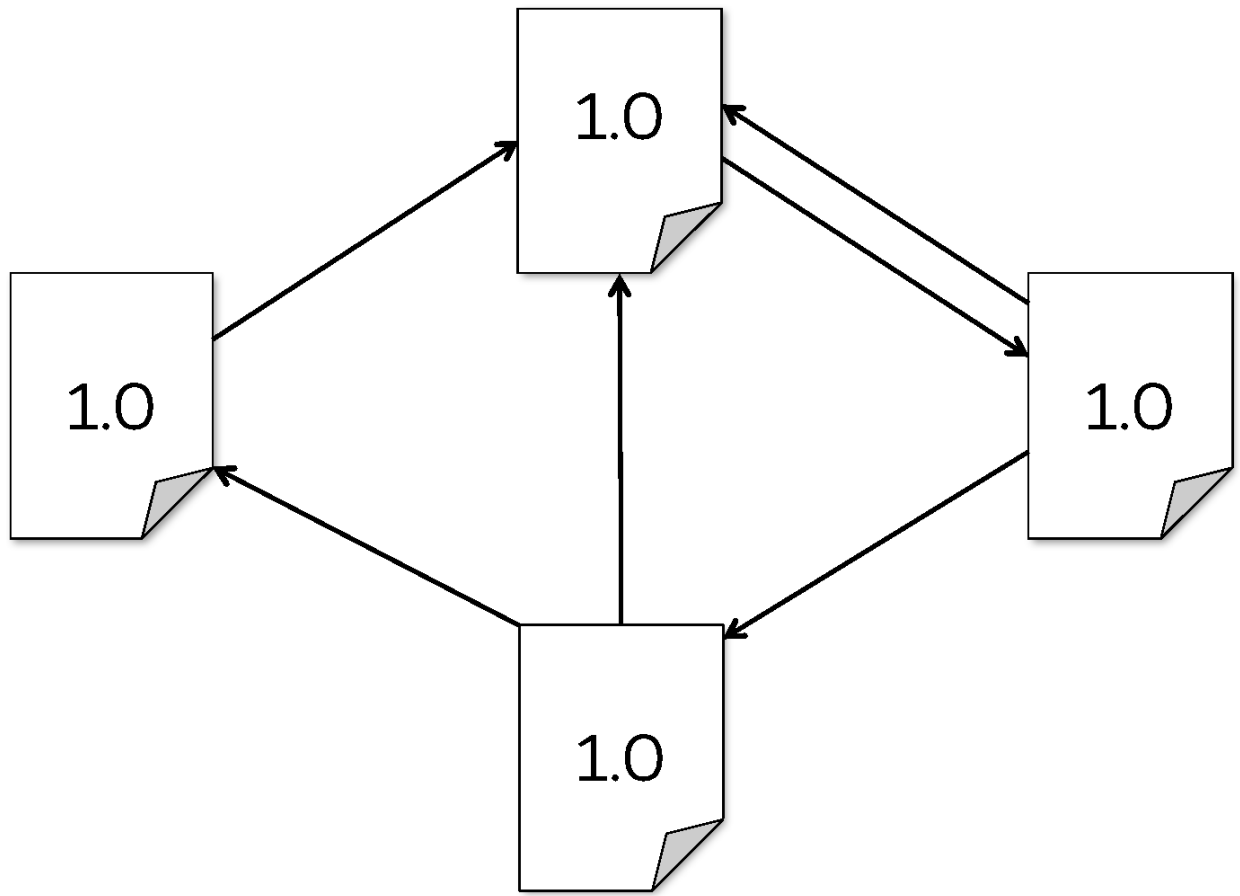
PageRank Algorithm

- Start each page with a rank of 1
- On each iteration:

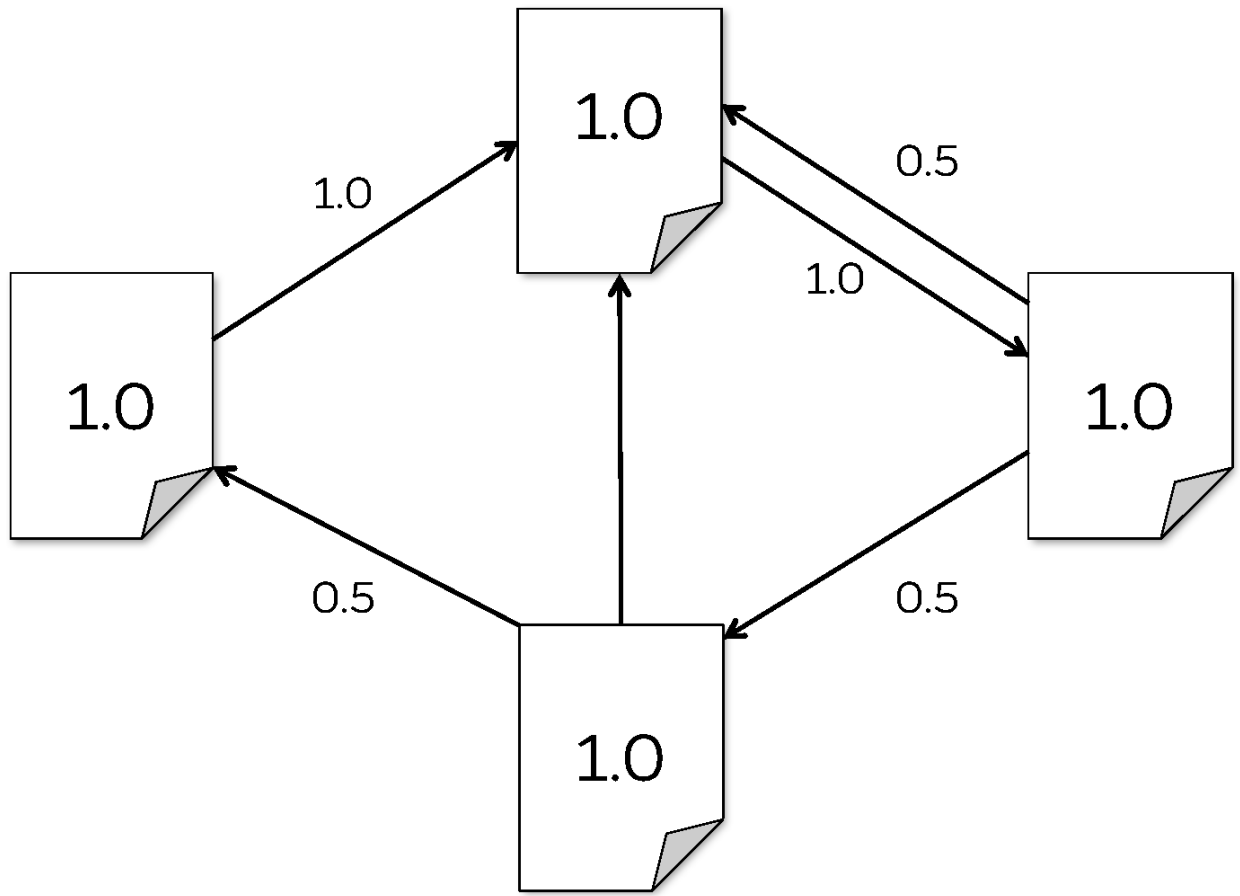
A. $$contrib = \frac{curRank}{|neighbors|}$$

B. $$curRank = 0.15 + 0.85 \sum_{neighbors} contrib_i$$

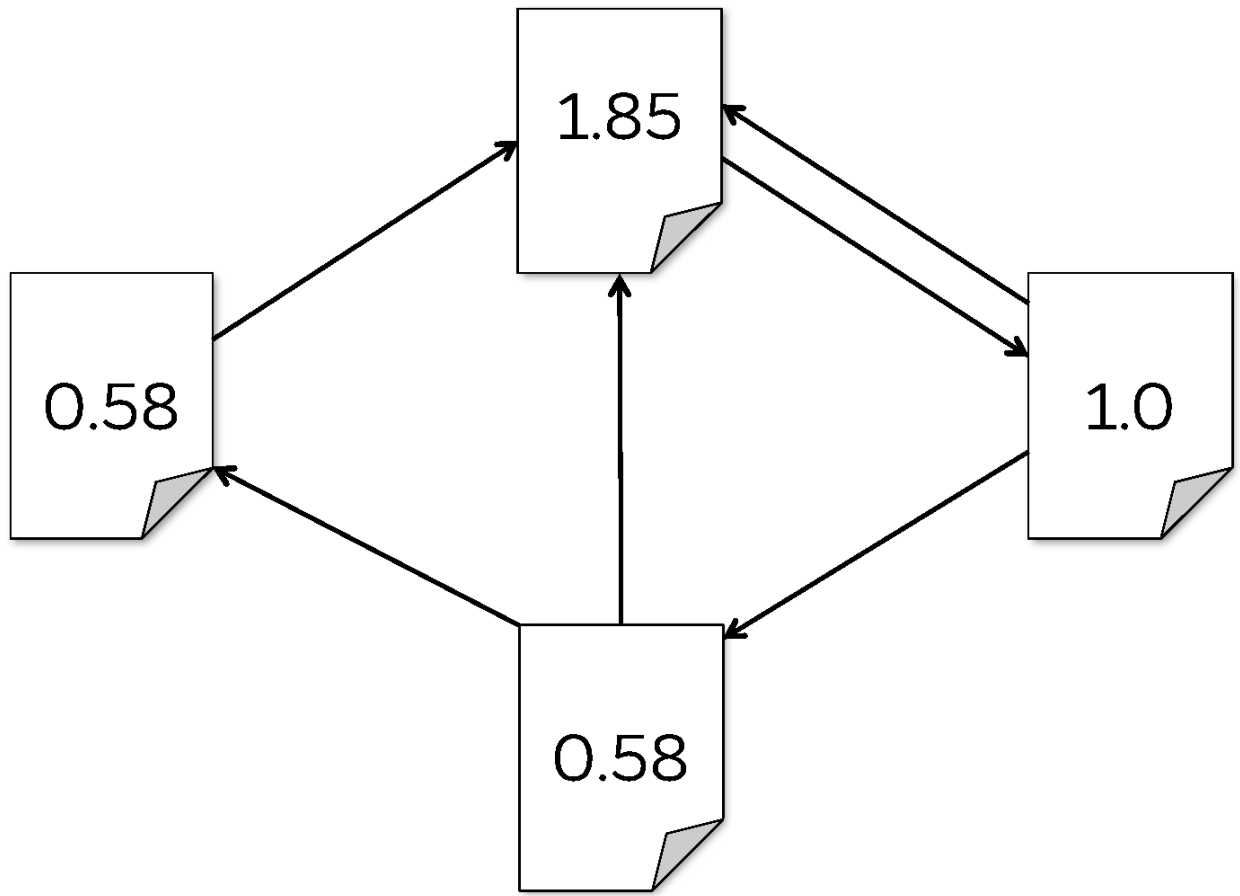
PageRank Example



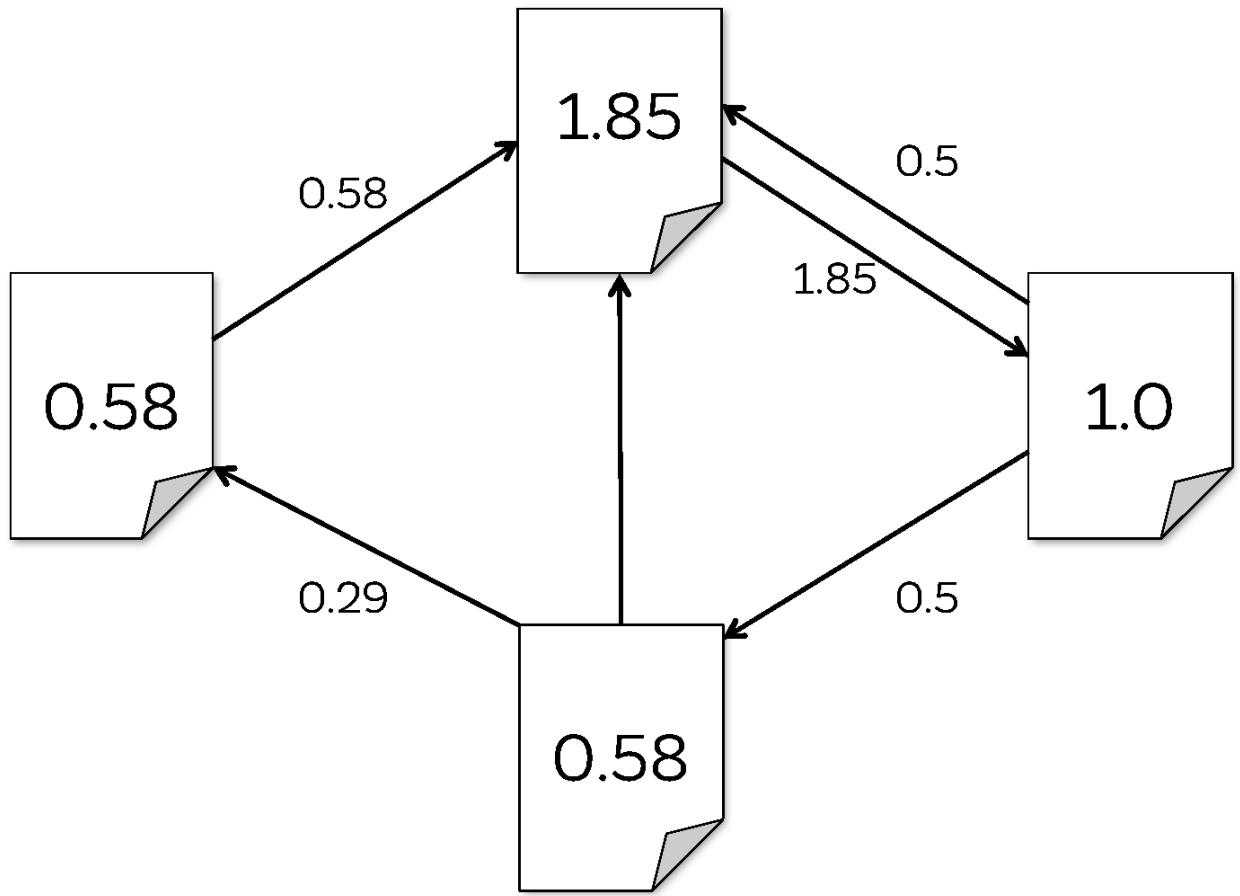
PageRank Example



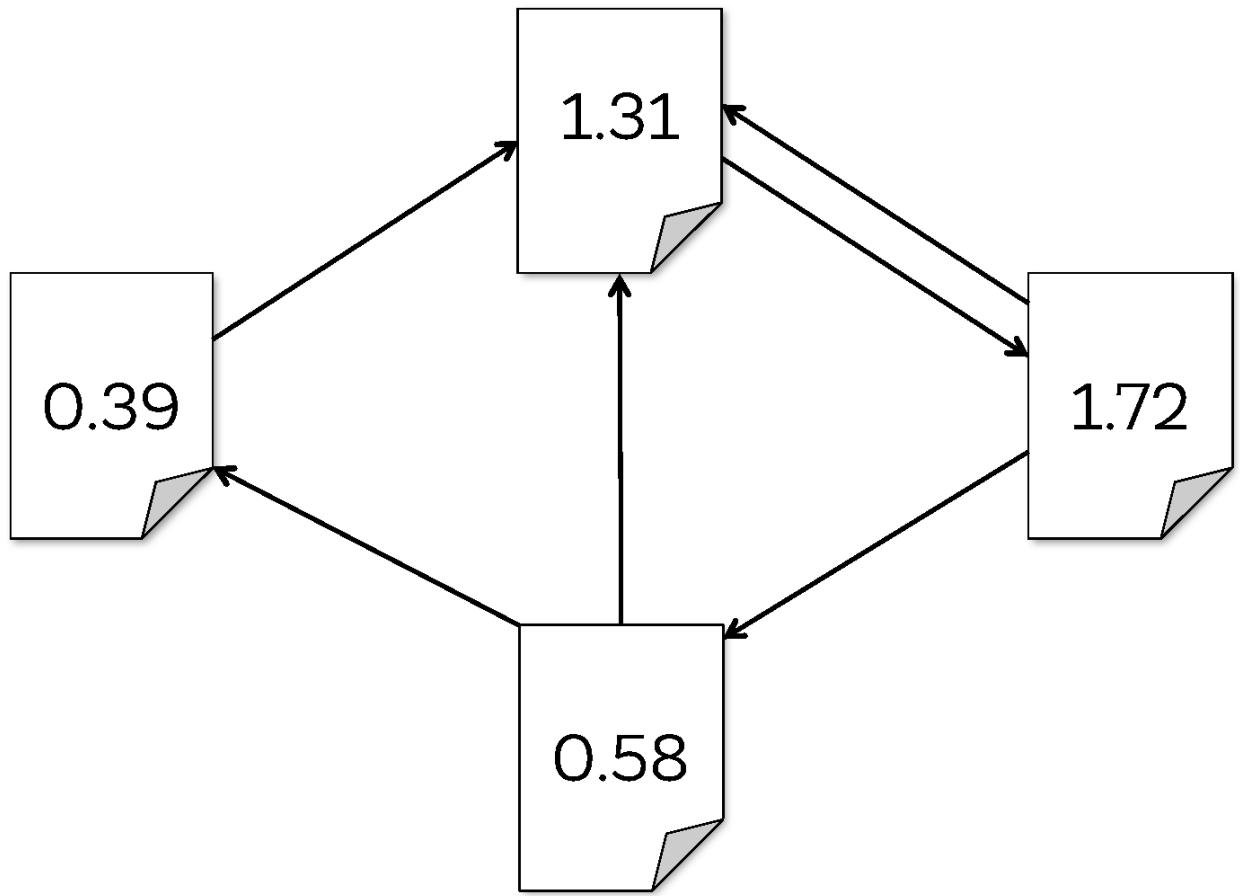
PageRank Example



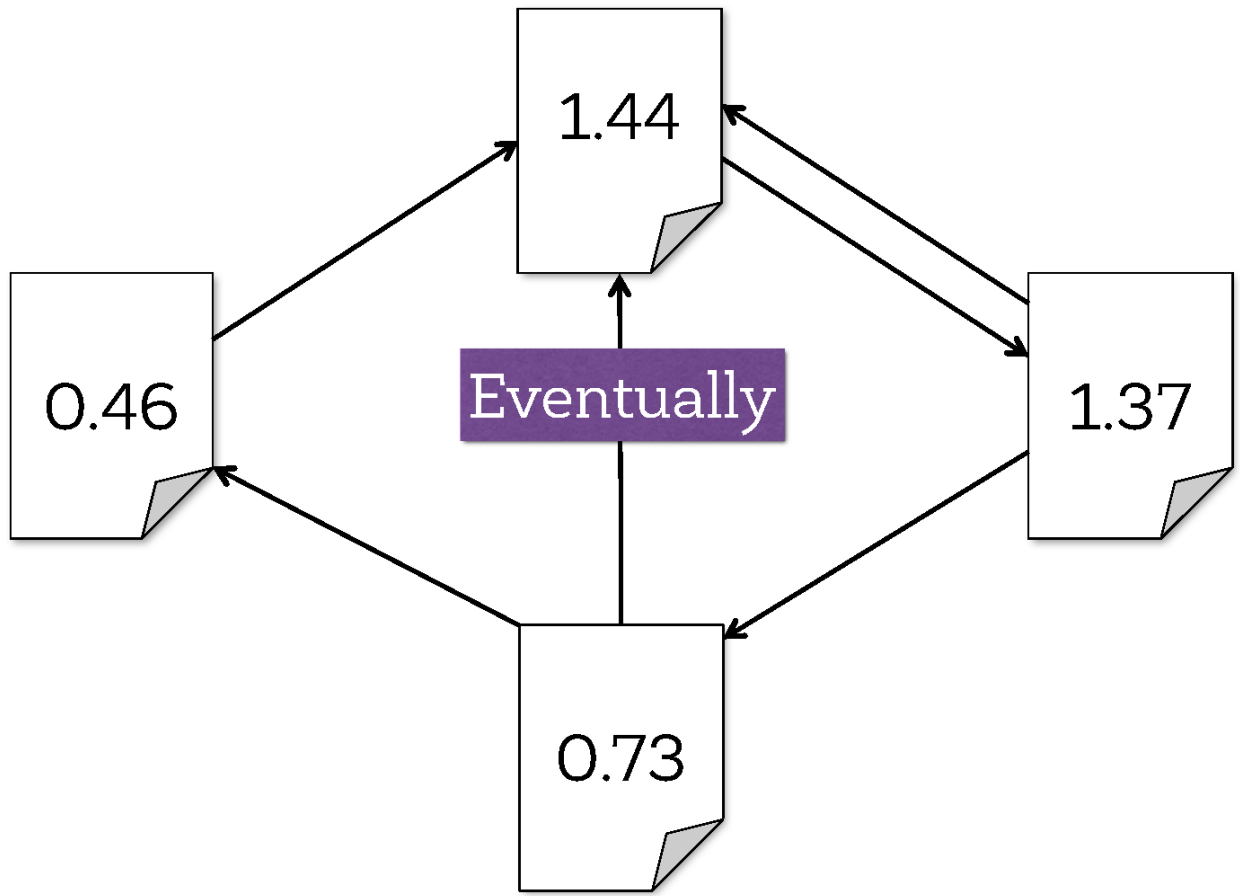
PageRank Example



PageRank Example



PageRank Example



Pagerank

- Two datasets: one of (pageID, link List) elements containing the list of neighbors of each page, and one of (pageID, rank) elements containing the current rank for each page.

1. Initialize each page's rank to 1.0.
2. On each iteration, have page p send a contribution of rank(p)/numNeighbors(p) to its neighbors (the pages it has links to).
3. Set each page's rank to 0.15 + 0.85 * contributionsReceived.

PageRank Code

```
// Assume that our neighbor list was saved as a Spark objectFile
val links = sc.objectFile[(String, Seq[String])]("links")

var ranks = links.[redacted](v => 1.0)

// Run 10 iterations of PageRank
for (i <- 0 until 10) {
  val contributions = links.join(ranks).[redacted]
    case (pageId, (links, rank)) =>
      links.map(dest => (dest, rank / links.size))
  }
  ranks = contributions.[redacted]((x, y) => x + y).[redacted](v => 0.15 + 0.85*v)
}

// Write out the final ranks
ranks.saveAsTextFile("ranks")
```

6 Keywords!

Algorithm description

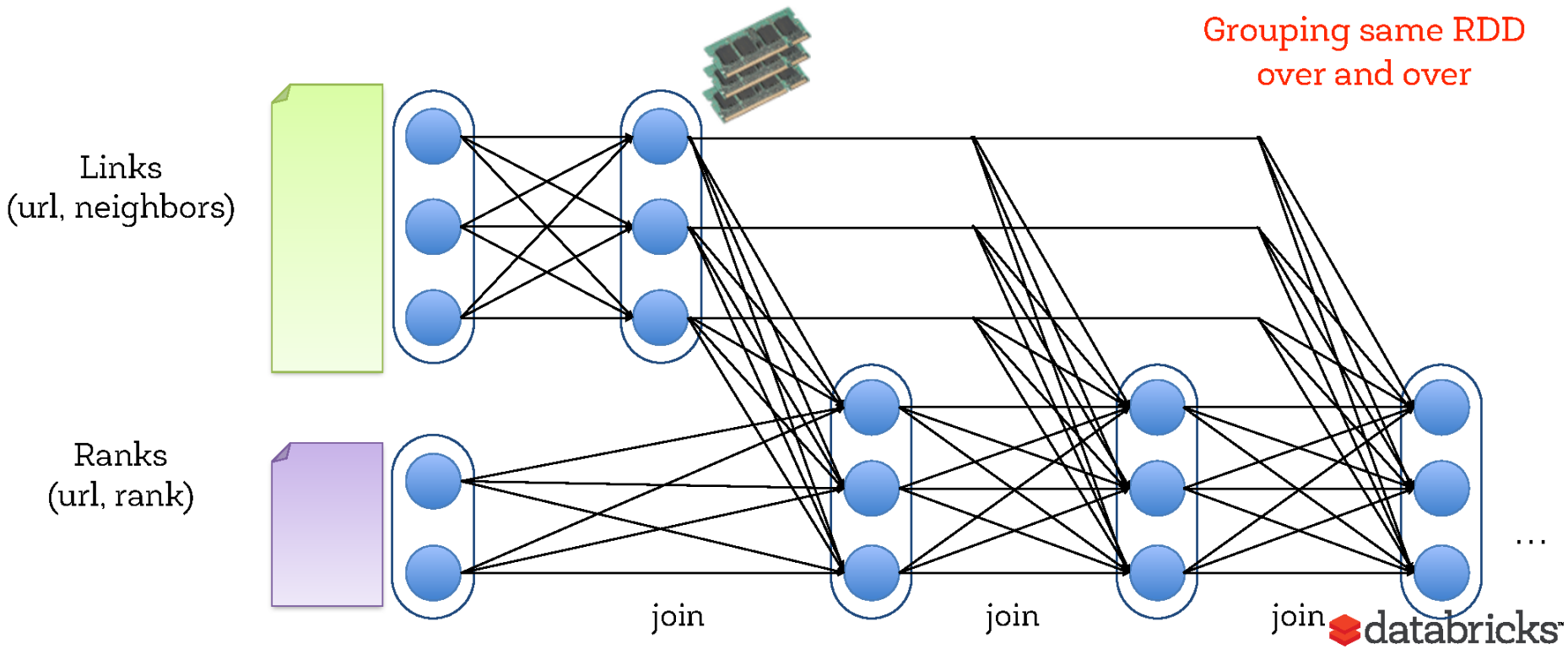
1. Starts with a ranks RDD initialized at 1.0 for each element, and keeps updating the ranks variable on each iteration
2. Do a join() between the current ranks RDD and the static links one
3. Then use this in a flatMap to create "contribution" values to send to each of the page's neighbors .
4. Then add up these values by page ID (i.e., by the page receiving the contribution) and set that page's rank to 0.15 + 0.85 * contributions Received.

To minimize communication

1. The links RDD is joined against ranks on each iteration
2. For the same reason, we call persist() on links to keep it in RAM across iterations.
3. Then add up these values by page ID (i.e., by the page receiving the contribution) and set that page's rank to 0.15 + 0.85 * contributions Received.
4. When we first create ranks, we use mapValues() instead of map() to preserve the partitioning of the parent RDD (links), so that our first join against it is cheap.
5. In the loop body, we follow our reduceByKey() with mapValues(); because the result of reduceByKey() is already hash-partitioned, this will make it more efficient to join the mapped result against links on the next iteration.

Spark can do much better

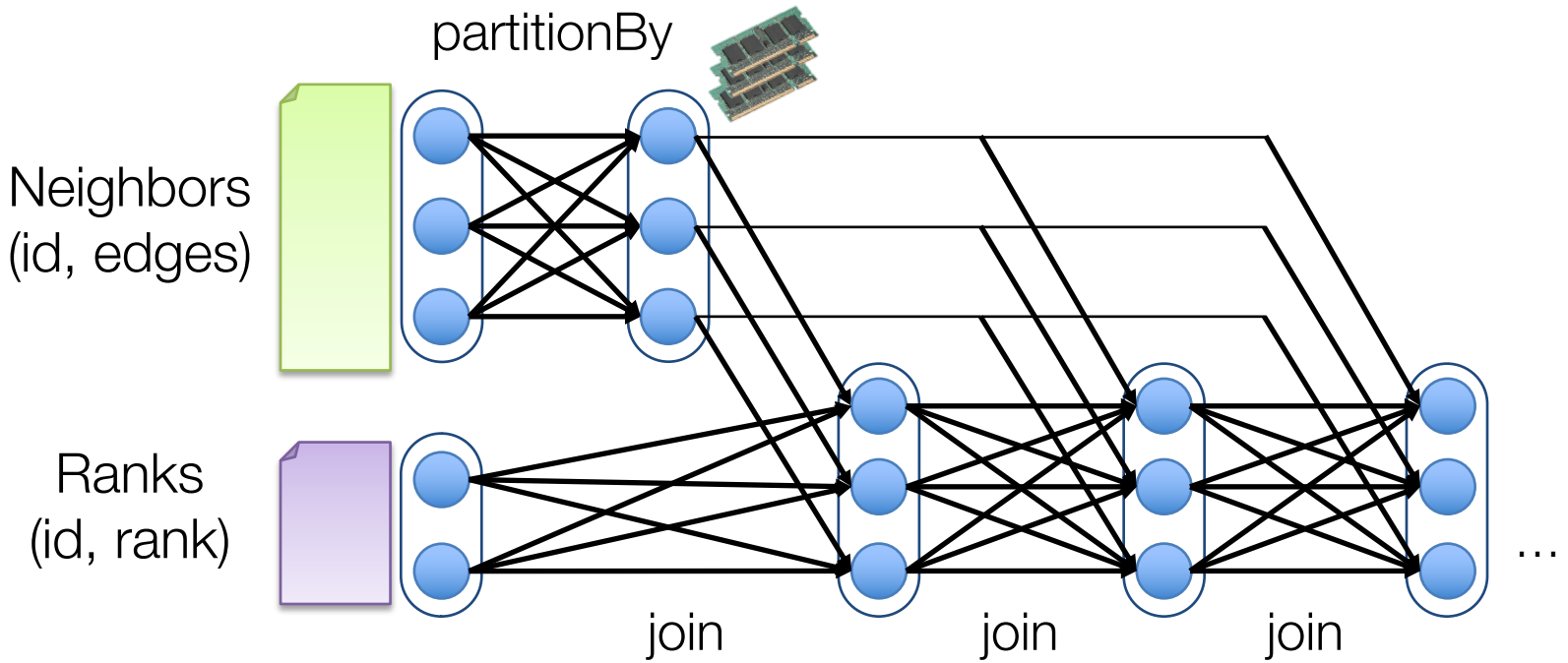
- Using cache(), keep neighbors in memory
- Do not write intermediate results on disk



PageRank

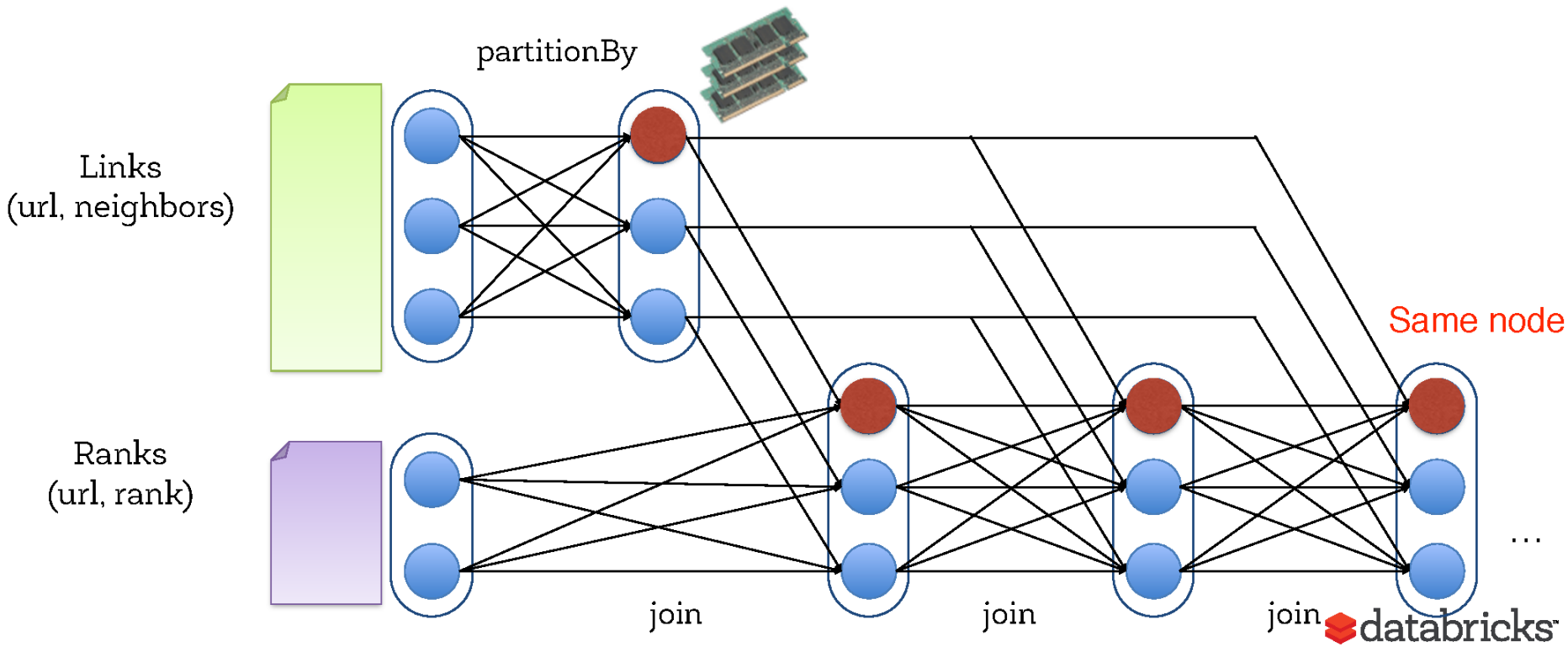
Using cache(), keep neighbors in RAM

Using partitioning, avoid repeated hashing

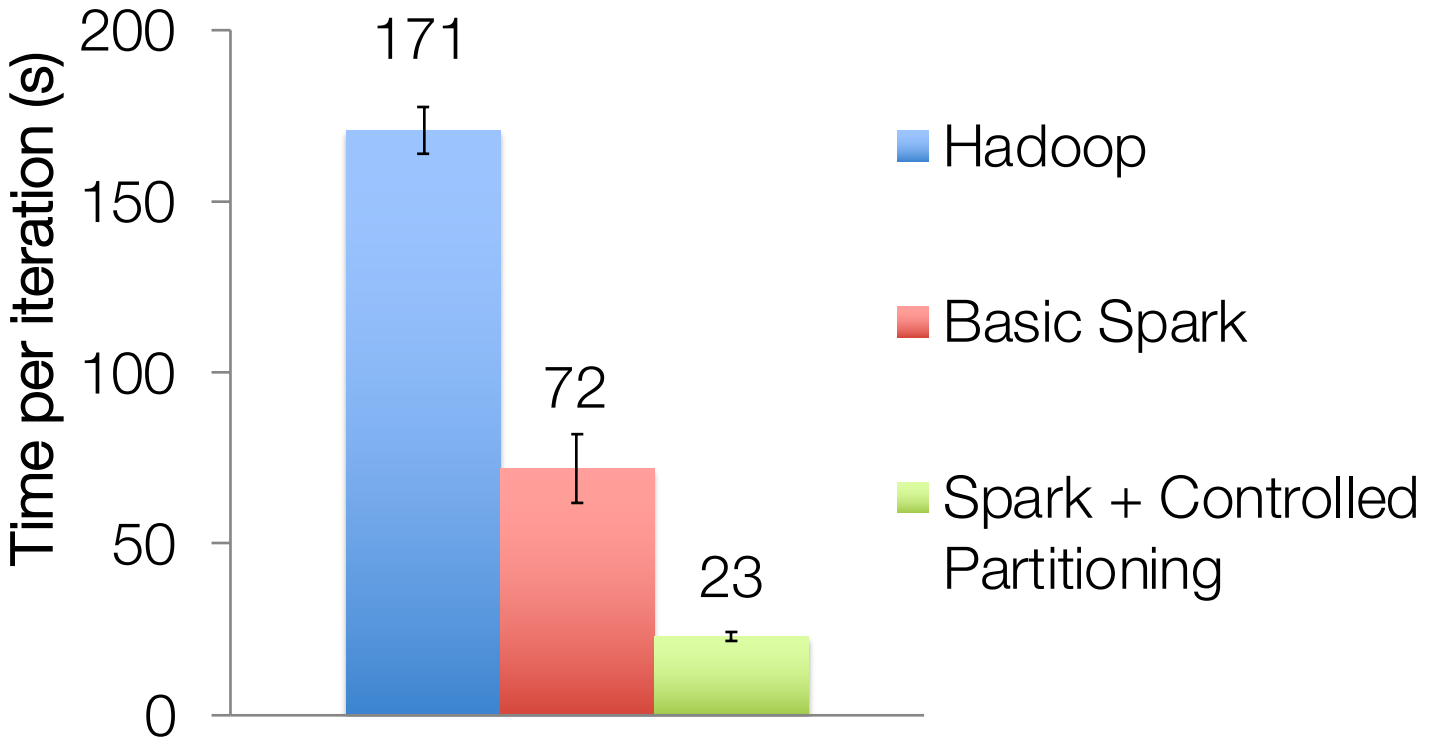


Spark can do much better

- Do not partition neighbors every time

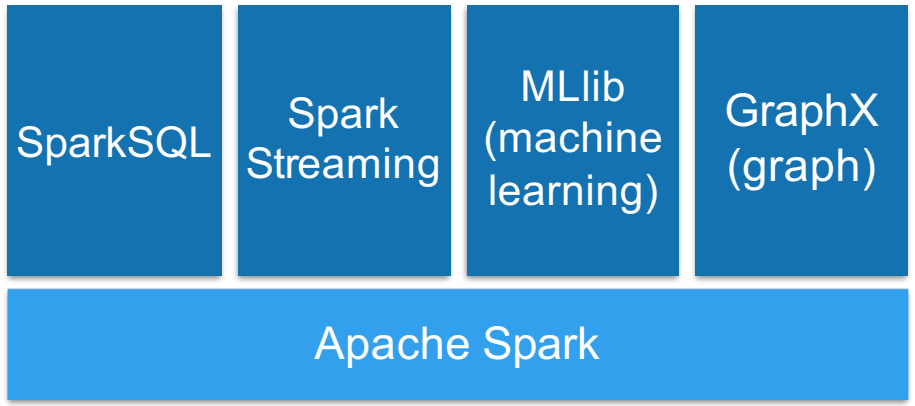


PageRank Results



MLlib

+ Simple development environment (Spark)
+ Scalable and fast
+ Part of Apache Spark Ecosystem



Link to Sparse installation with many examples:
http://spark.apache.org/docs/latest/index.html