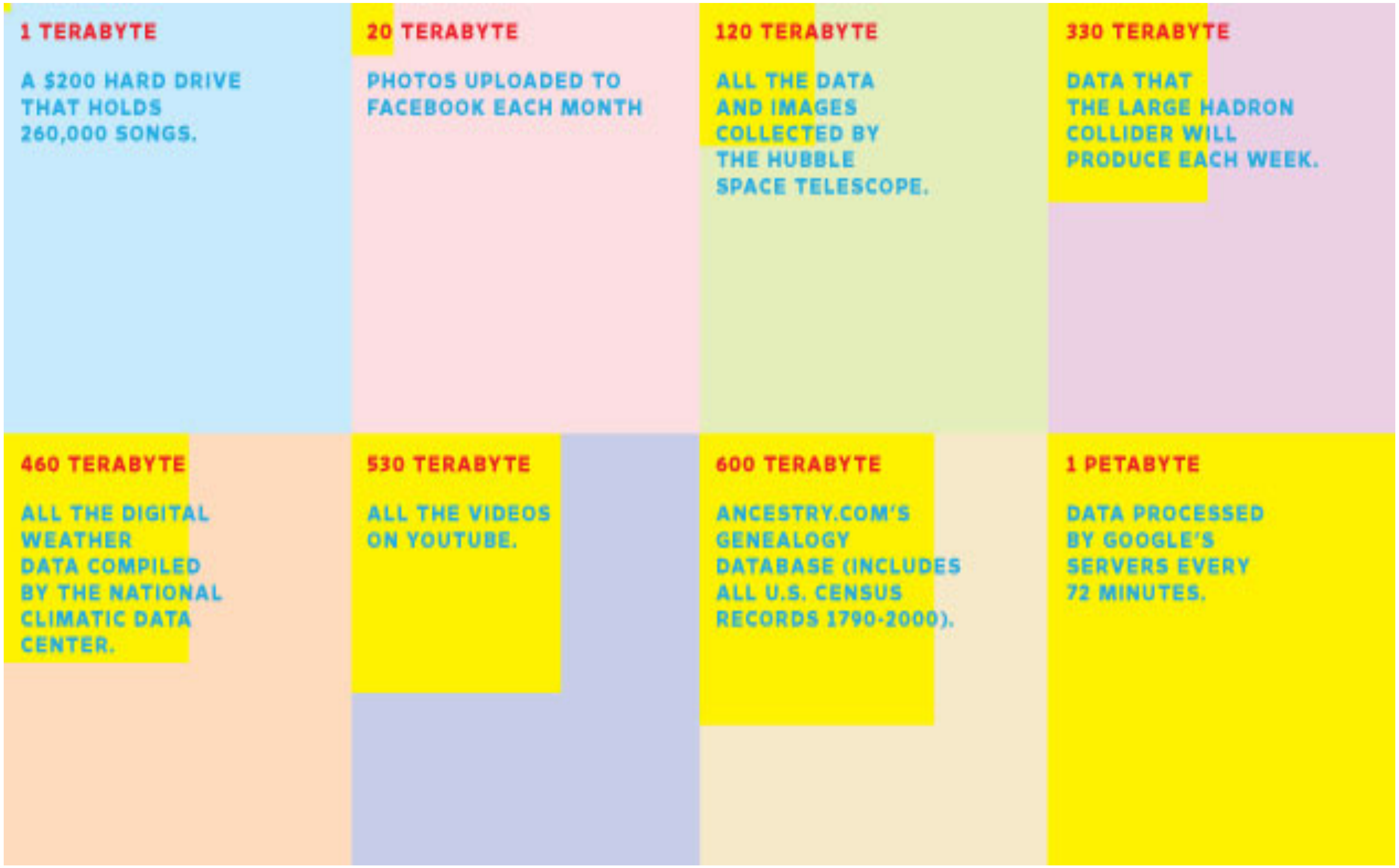
LARGE-SCALE DATA PROCESSING WITH MAPREDUCE

The Petabyte age



1 TERABYTE

A $200 HARD DRIVE THAT HOLDS 260,000 SONGS.

20 TERABYTE

PHOTOS UPLOADED TO FACEBOOK EACH MONTH

120 TERABYTE

ALL THE DATA AND IMAGES COLLECTED BY THE HUBBLE SPACE TELESCOPE.

330 TERABYTE

DATA THAT THE LARGE HADRON COLLIDER WILL PRODUCE EACH WEEK.

460 TERABYTE

ALL THE DIGITAL WEATHER DATA COMPILED BY THE NATIONAL CLIMATIC DATA CENTER.

530 TERABYTE

ALL THE VIDEOS ON YOUTUBE.

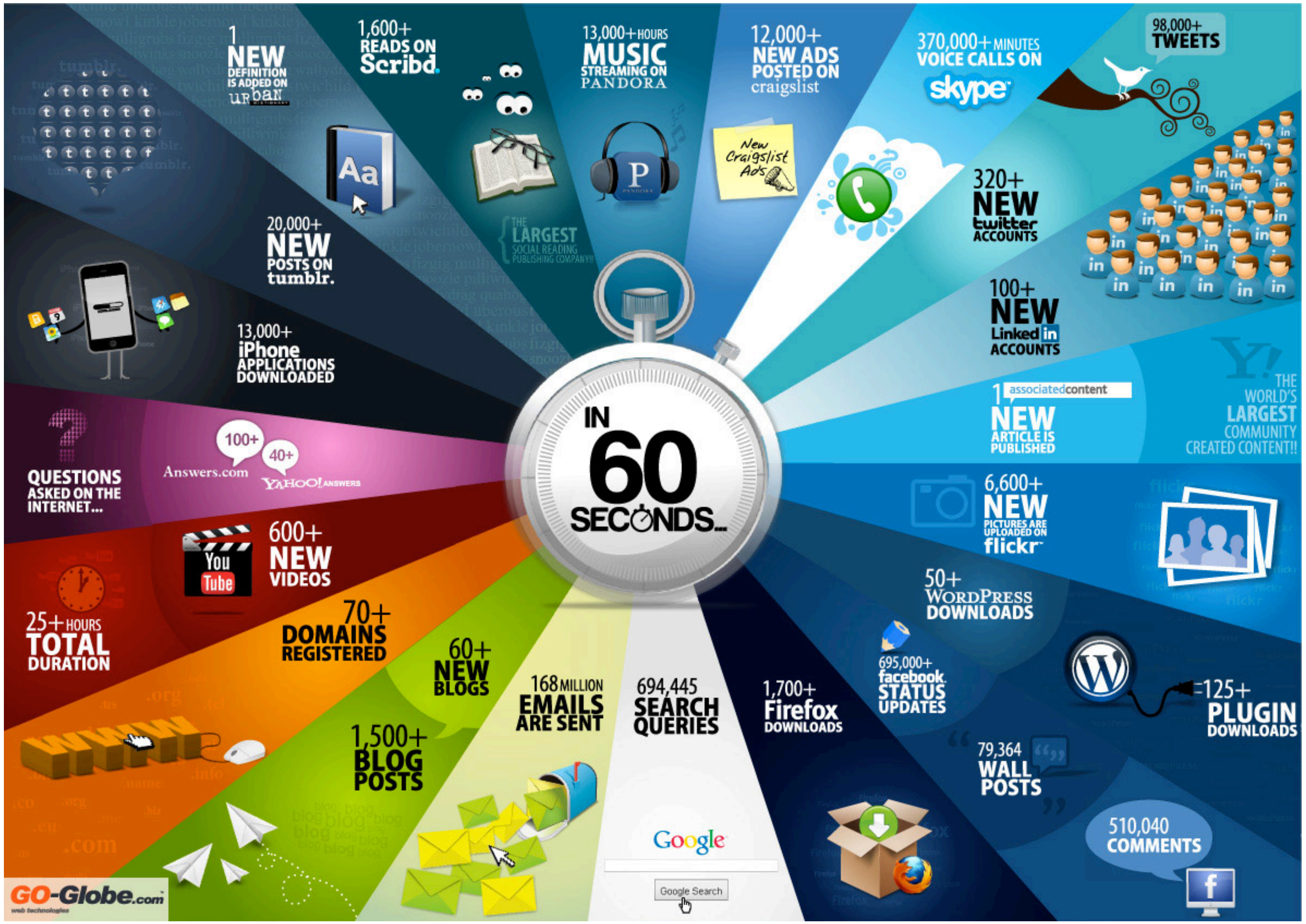
600 TERABYTE

ANCESTRY.COM'S GENEALOGY DATABASE (INCLUDES ALL U.S. CENSUS RECORDS 1790-2000).

1 PETABYTE

DATA PROCESSED BY GOOGLE'S SERVERS EVERY 72 MINUTES.

The Petabyte age


1 NEW DEFINITION IS ADDED ON urban
1,600+ READS ON Scribd.
13,000+ HOURS MUSIC STREAMING ON PANDORA
12,000+ NEW ADS POSTED ON craigslist
370,000+ MINUTES VOICE CALLS ON skype
98,000+ TWEETS
20,000+ NEW POSTS ON tumblr.
THE LARGEST SOCIAL READING PUBLISHING COMPANY!!
New Craigslist Ads
320+ NEW twitter ACCOUNTS
100+ NEW LinkedIn ACCOUNTS
13,000+ iPhone APPLICATIONS DOWNLOADED
1 associatedcontent NEW ARTICLE IS PUBLISHED
THE WORLD'S LARGEST COMMUNITY CREATED CONTENT!!
IN 60 SECONDS...
QUESTIONS ASKED ON THE INTERNET...
100+ Answers.com
40+ YAHOO! ANSWERS
600+ NEW VIDEOS
6,600+ NEW PICTURES ARE UPLOADED ON flickr
25+ HOURS TOTAL DURATION
70+ DOMAINS REGISTERED
60+ NEW BLOGS
168 MILLION EMAILS ARE SENT
694,445 SEARCH QUERIES
1,700+ Firefox DOWNLOADS
695,000+ facebook STATUS UPDATES
50+ WordPress DOWNLOADS
125+ PLUGIN DOWNLOADS
1,500+ BLOG POSTS
79,364 WALL POSTS
510,040 COMMENTS
Google
Google Search
GO-Globe.com web technologies

Plucking the diamond from the waste

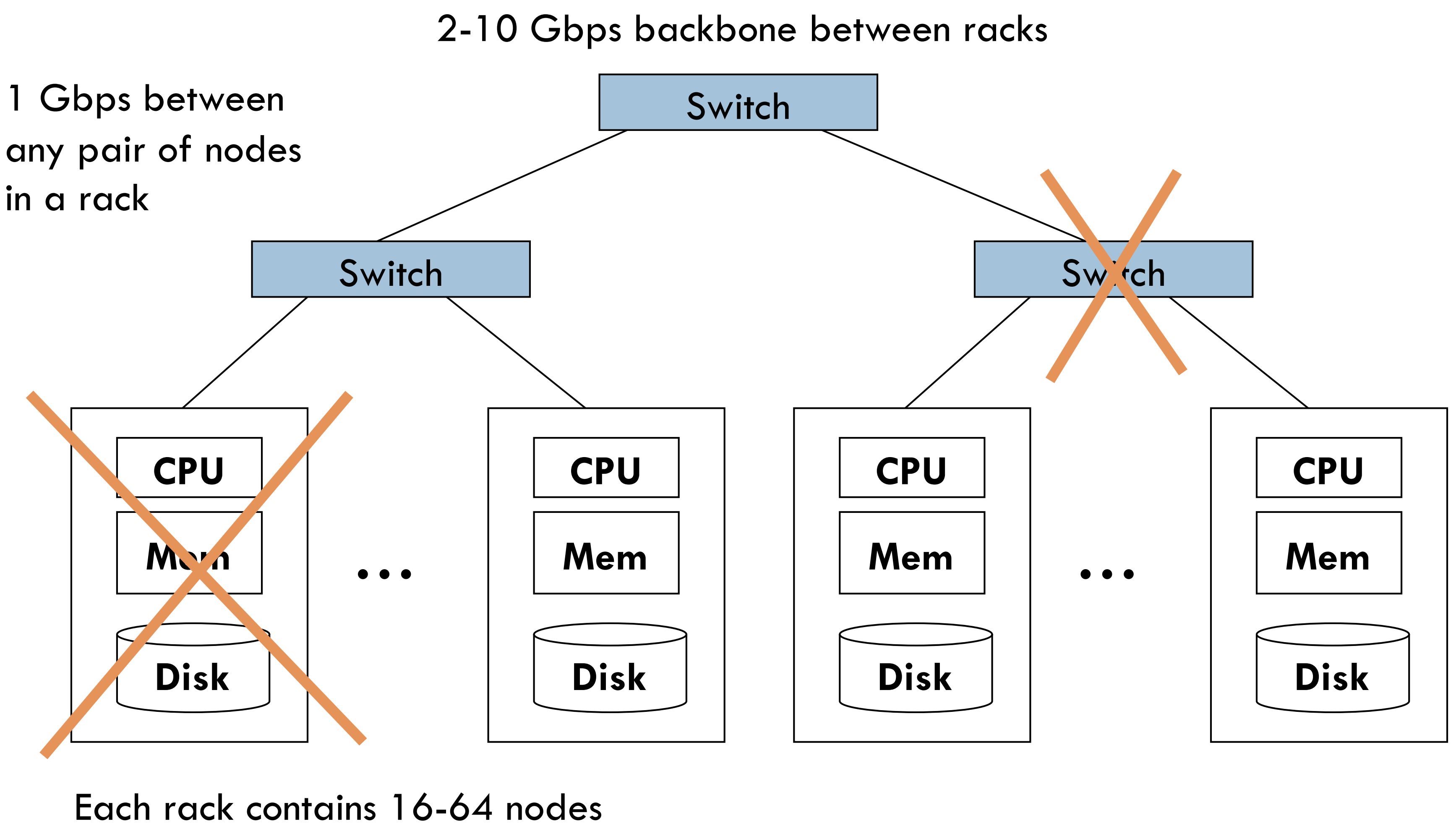
- “**Credit-card companies** monitor every purchase and can identify fraudulent ones with a high degree of accuracy, using rules derived by crunching through billions of transactions.”
- “**Mobile-phone operators**, meanwhile, analyse subscribers’ calling patterns to determine, for example, whether most of their frequent contacts are on a rival network.”
- “[…] **Cablecom**, a Swiss telecoms operator. ***It has reduced customer defections from one-fifth of subscribers a year to under 5% by crunching its numbers.***”

- “***Retailers, offline as well as online, are masters of data mining.***”

Commodity Clusters

- Web data sets can be very large
  - Tens to hundreds of terabytes
- Cannot mine on a single server
  - why? obviously …
- Standard architecture emerging:
  - Cluster of commodity Linux nodes
  - Gigabit ethernet interconnections
- How to organize computations on this architecture?
  - Mask issues such as hardware failure

Cluster Architecture



2-10 Gbps backbone between racks
1 Gbps between any pair of nodes in a rack
Switch
Switch
Switch
CPU
Mem
Disk
CPU
Mem
Disk
CPU
Mem
Disk
CPU
Mem
Disk
…
…
Each rack contains 16-64 nodes

Stable storage

- First issue:
  - if nodes can fail, how can we store data persistently?
- Answer: Distributed File System
  - Provides global file namespace
  - Google GFS; Hadoop HDFS; Kosmix KFS
- Typical usage pattern
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

Distributed File System

- Chunk Servers
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- Master node
  - a.k.a. Name Node in HDFS
  - Stores metadata
  - Might be replicated
- Client library for file access
  - Talks to master to find chunk servers
  - Connects directly to chunkservers to access data

MapReduce

- A “novel” programming paradigm.
- A different data type:
  - Everything is built on top of <key,value> pairs
    - Keys and values are user defined, they can be anything
- There are only *two user defined functions*:
  - Map
    - map(k1,v1) → list(k2,v2)
  - Reduce
    - reduce(k2, list(v2)) → list(v3)

MapReduce

- Two simple functions:

1. **map** (k1,v1) → list(k2,v2)
   - given the input data *(k1,v1)*,
     and produces some intermediate data *(v2)*
     labeled with a key *(k2)*

2. **reduce** (k2,list(v2)) → list(v3)
   - given every data *(list(v2))* associated with a key *(k2)*,
     produces the output of the algorithm *list(v3)*

MapReduce

- All in parallel:
  - we have a set of ***mappers*** and a set of ***reducers***

1. **map** (k1,v1) → list(k2,v2)
   - a *mapper* processes only a *split* of the input, which may be distributed across several machines

2. **shuffle**
   - a middle phase transfers the data associated with a given key from the mappers to the proper reducer.
   - reducers will receive data sorted by key

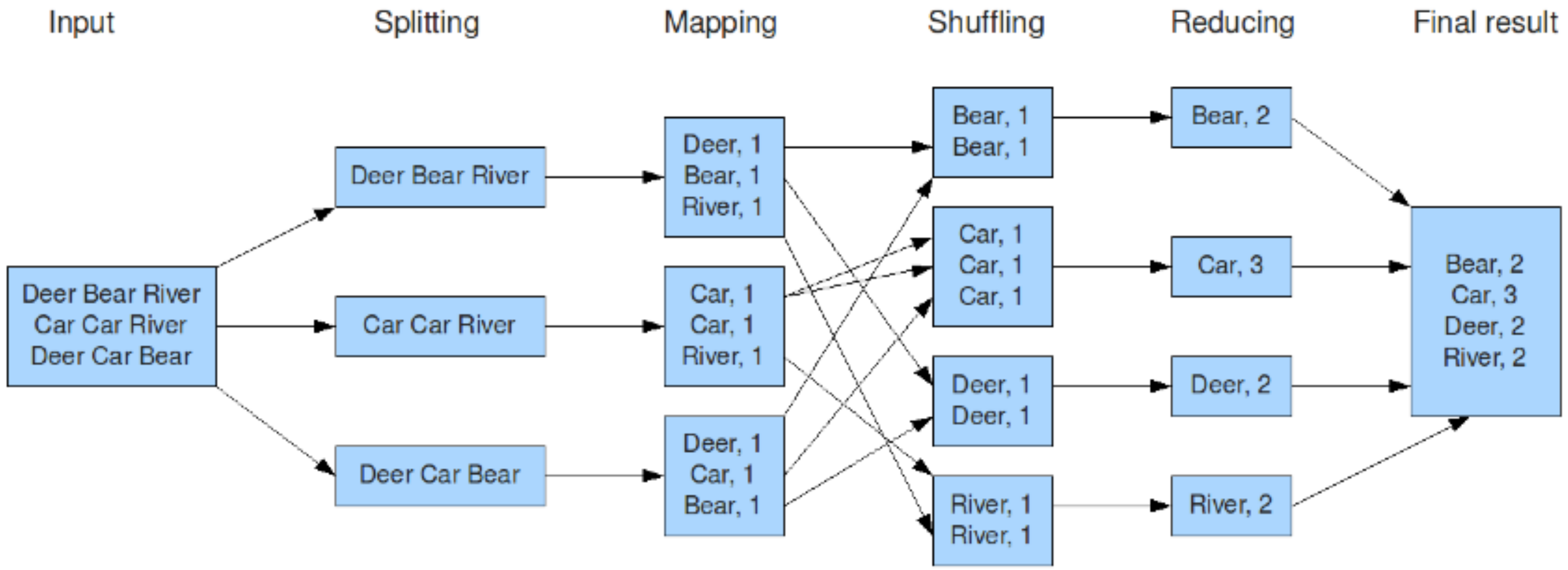
3. **reduce** (k2,list(v2)) → list(v3)
   - a *reducer* produces only a portion of the output associated with a given set of keys

Word Count using MapReduce

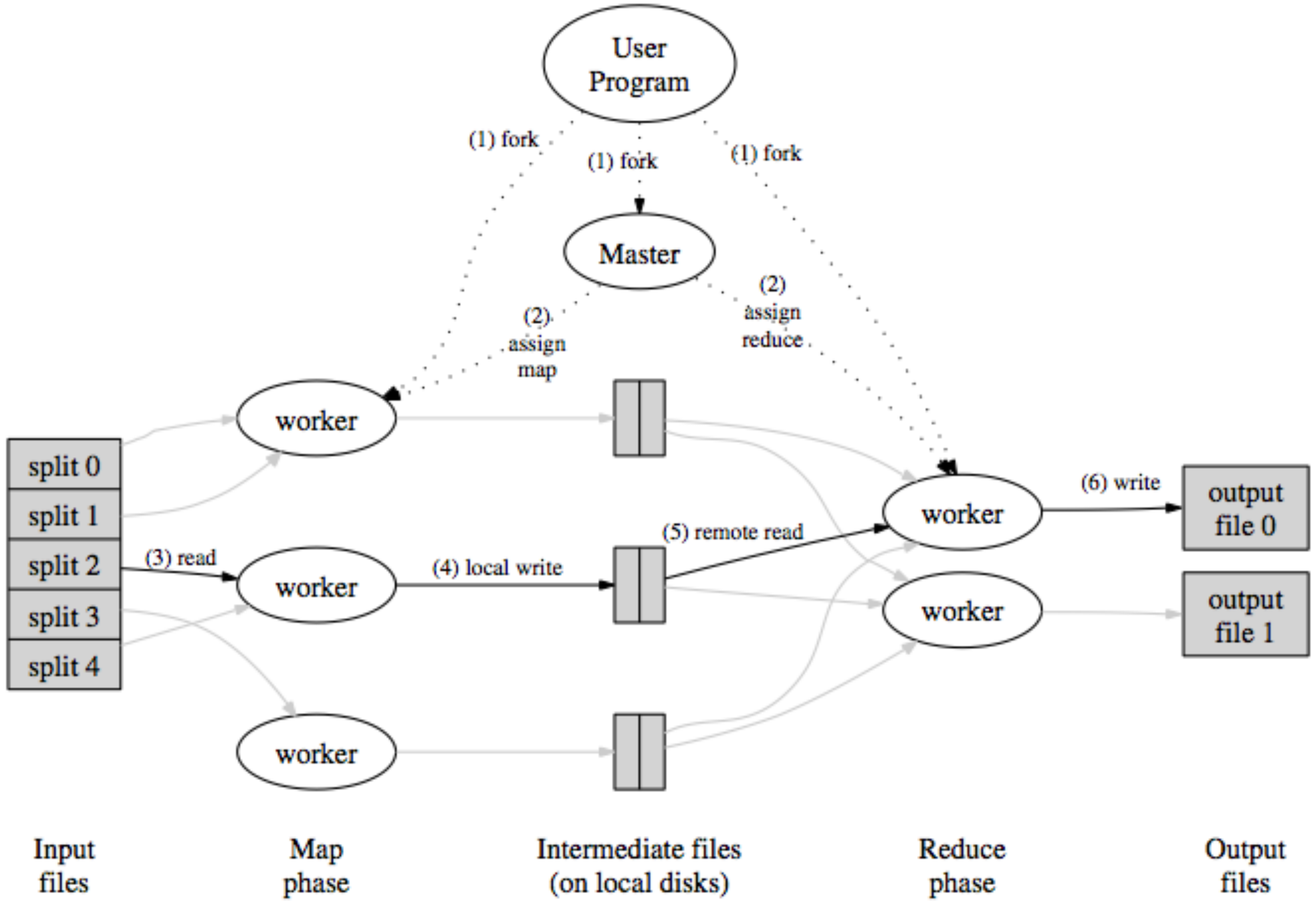
```
map(key, value):
// key: document name; value: text of document
    for each word w in value:
            emit(w, 1)


reduce(key, values):
// key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(result)
```

Word Count example

The overall MapReduce word count process



MapReduce

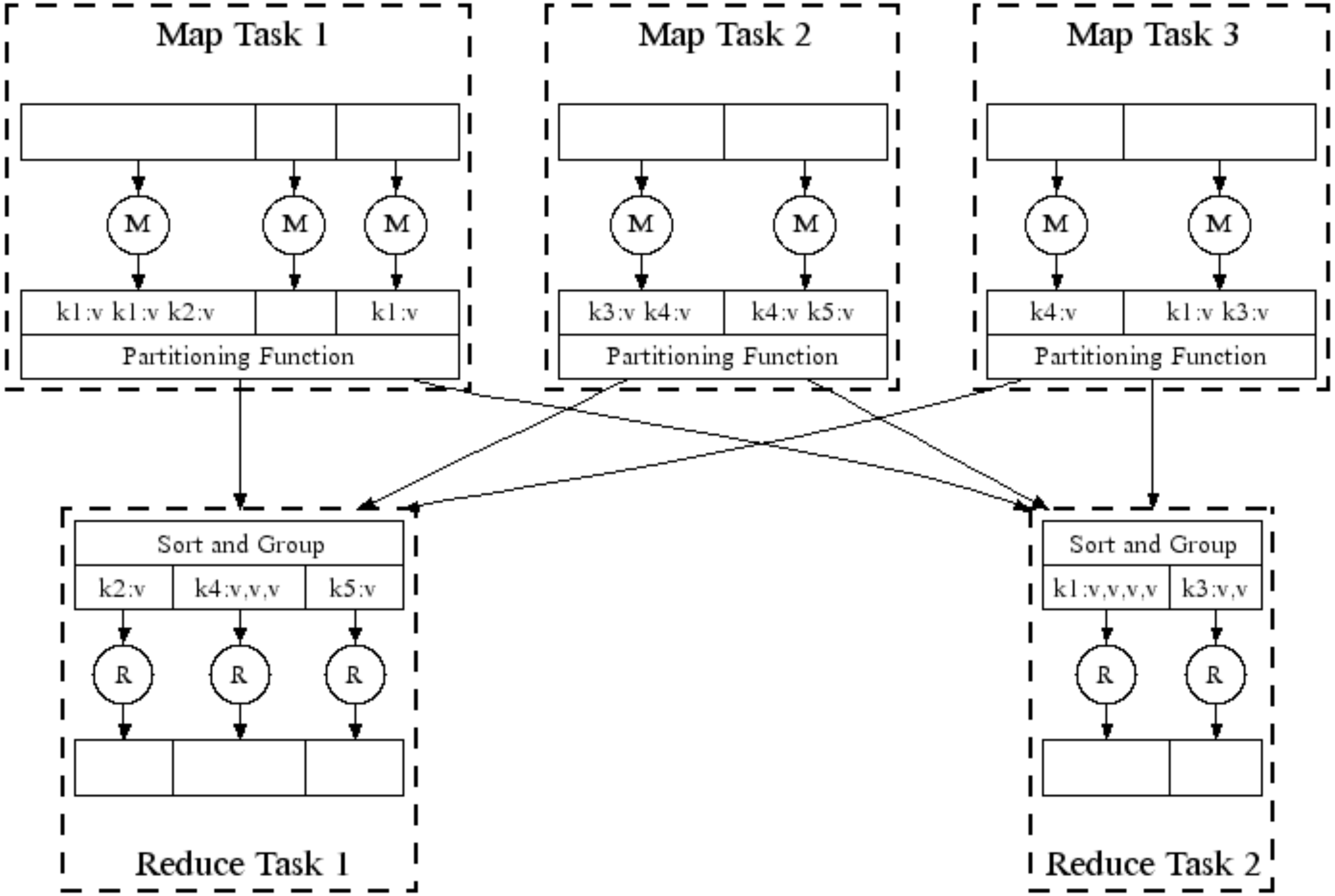


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

Coordination

- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

Process mapping



Failures

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
  - A different reducer may take the idle task over
- Master failure
  - MapReduce task is aborted and client is notified

How many Map and Reduce jobs?

- *M* map tasks, *R* reduce tasks
- Rule of thumb:
  - Make *M* and *R* larger than the number of nodes in cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds recovery from worker failure
- Usually *R* is smaller than *M*
  - Having several reducers increases load balancing, but generates multiple “waves” somehow delaying the shuffling
  - output is spread across *R* files

Combiners

- Often a map task will produce many pairs of the form (k,v1), (k,v2), … for the same key k
  - E.g., popular words in Word Count
- Can save network time by pre-aggregating at mapper
  - *Combine (k1, list(v1)) → (k1,list(v2))*
  - Usually same as reduce function
  - If reduce function is commutative and associative

Partition Function

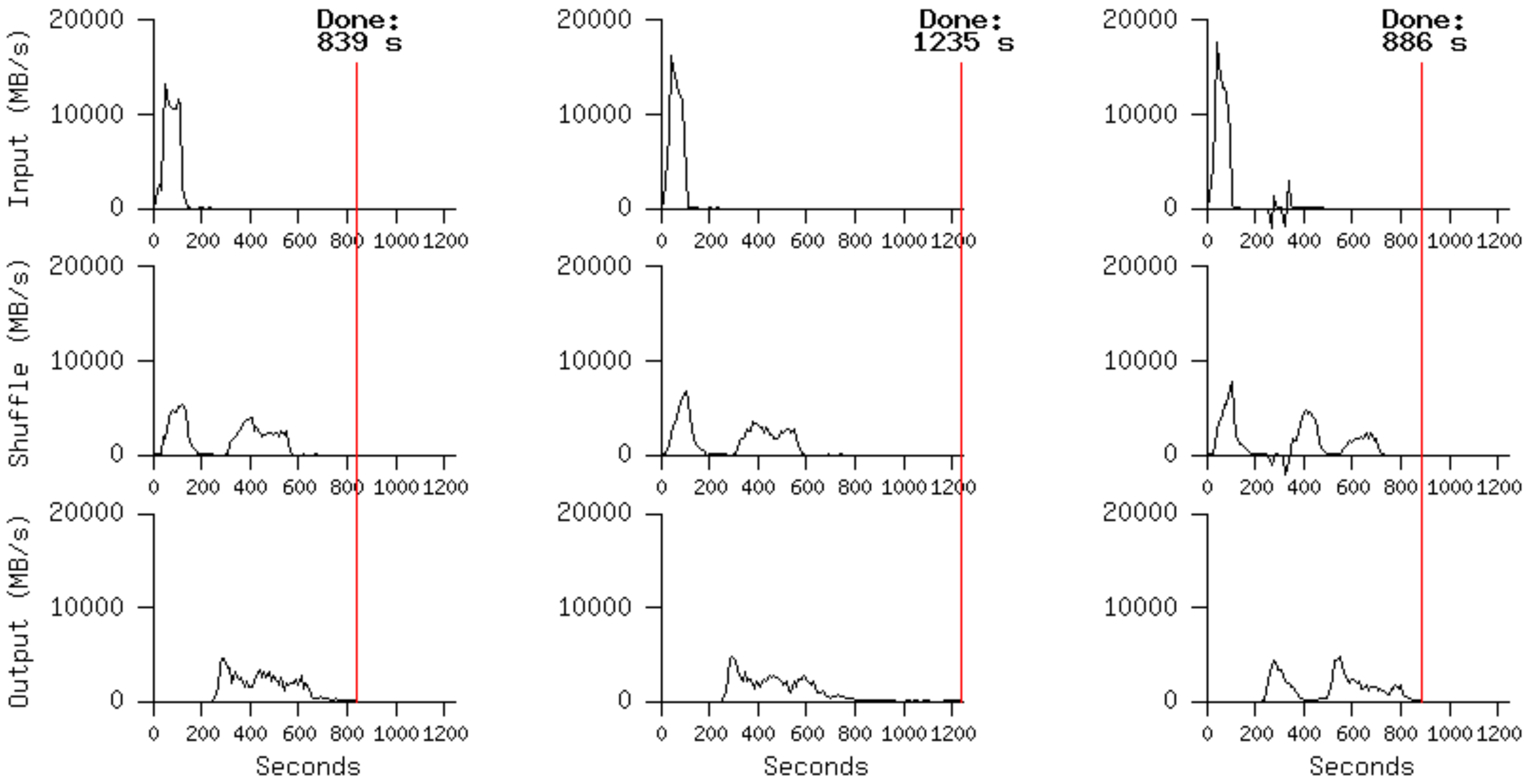
- Inputs to map tasks are created by contiguous splits of input file
- For reducer, we need to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function
  e.g., hash(key) mod R

Back-up Tasks

- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks transfer data very slowly
  - Weird things: processor caches disabled (!!)
- Solution: near end of phase,
  spawn backup copies of tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time

MR-Sort (1800 nodes)

Normal — No back-up tasks — 200 processes killed



Is MapReduce so good ?

- MapReduce is not for performance !!
  - Every mapper writes to disk which creates huge overhead
  - The shuffle step is usually the bottleneck
- Little coding time
  - You just need to override two functions
  - (sometimes you need to implement other stuff such as combiner, partitioner, sort)
  - But not everything can be implemented in terms of MR
- Fault tolerance:
  - Drives avg lifetime is 3 years, if you have 3600 drives then 3 will go broken *per day*.
  - How would you implement fault tolerance in MPI ??
- Data-parallel programming model helps

Hard to be implemented in MR

- All Pairs Similarity Search Problem:
  - Given a collection of documents find any pairs of documents with similarity greater than τ
  - With N documents, we have N^2 candidates
  - Some of experiments show that for 60MB worth of documents, you generate 40GB of intermediate data to be shuffled.
- Graph mining problems:
  - Find communities, find leaders, PageRank
  - In many cases you need to share the adjacency matrix, but how to broadcast ?

Hadoop

- Hadoop is
  - an Apache project
  - providing a Java implementation of MapReduce
    - Just need to copy java libs to each machine of your cluster
  - HDFS Hadoop distributed file system
    - Efficient and reliable
- Check on-line javadoc and “Hadoop the Definitive Guide”

Uses of MapReduce/Hadoop

- "the New York Times a few years ago used cloud computing and Hadoop to convert over 400,000 scanned images from its archives, from 1851 to 1922. By harnessing the power of hundreds of computers, it was able to do the job in 36 hours."

- "Visa, a credit-card company, in a recent trial with Hadoop crunched two years of test records, or 73 billion transactions, amounting to 36 terabytes of data. The processing time fell from one month with traditional methods to a mere 13 minutes."

Dryad & DryadLINQ

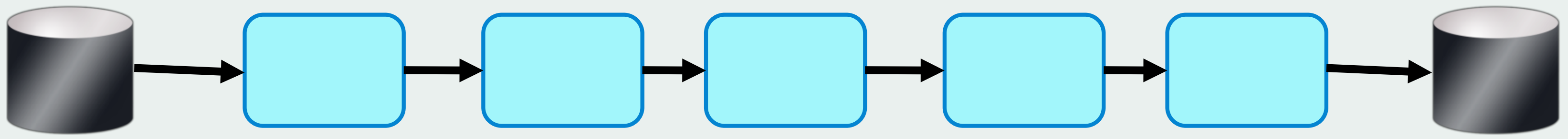
- Written at Microsoft Research, Silicon Valley

- Deployed since 2006
- Running 24/7 on >> 10^4 machines
- Sifting through > 10Pb data daily
- Clusters > 3000 machines
- Jobs with > 10^5 processes each
- Platform for rich software ecosystem

2-D Piping

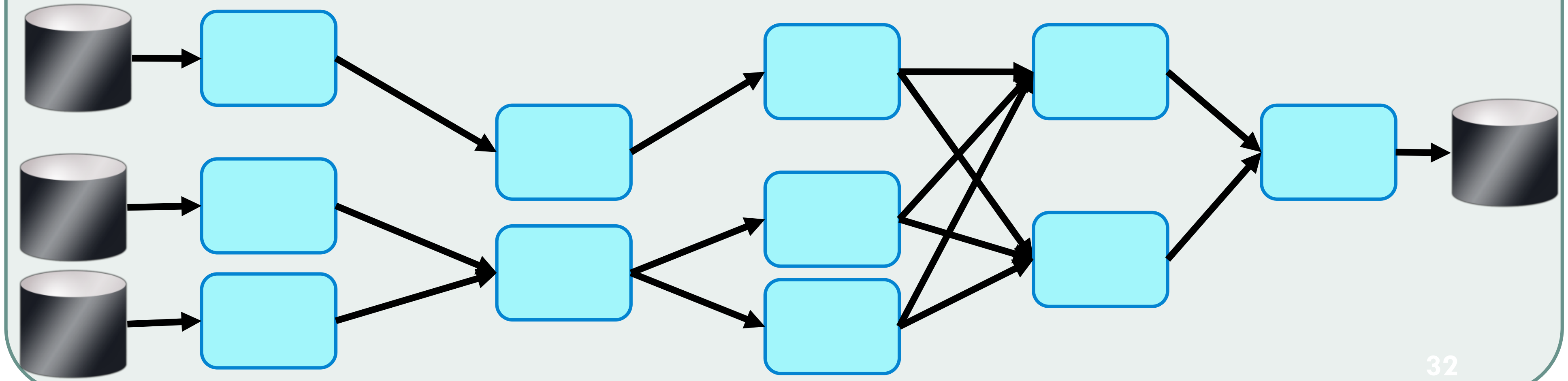
- MapReduce Pipe: 1-D

grep | sed | sort | awk | perl

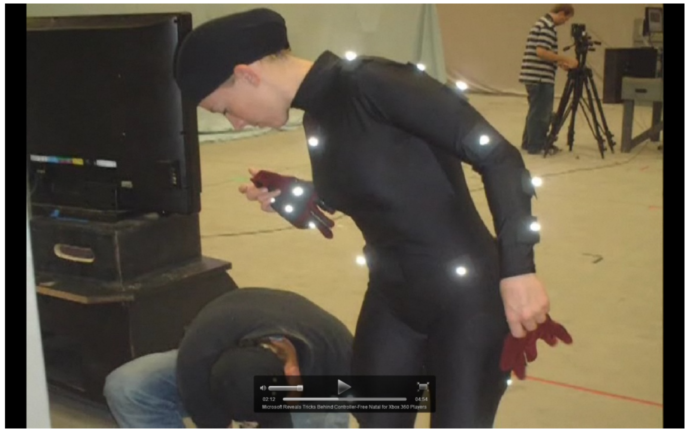


- Dryad: 2-D

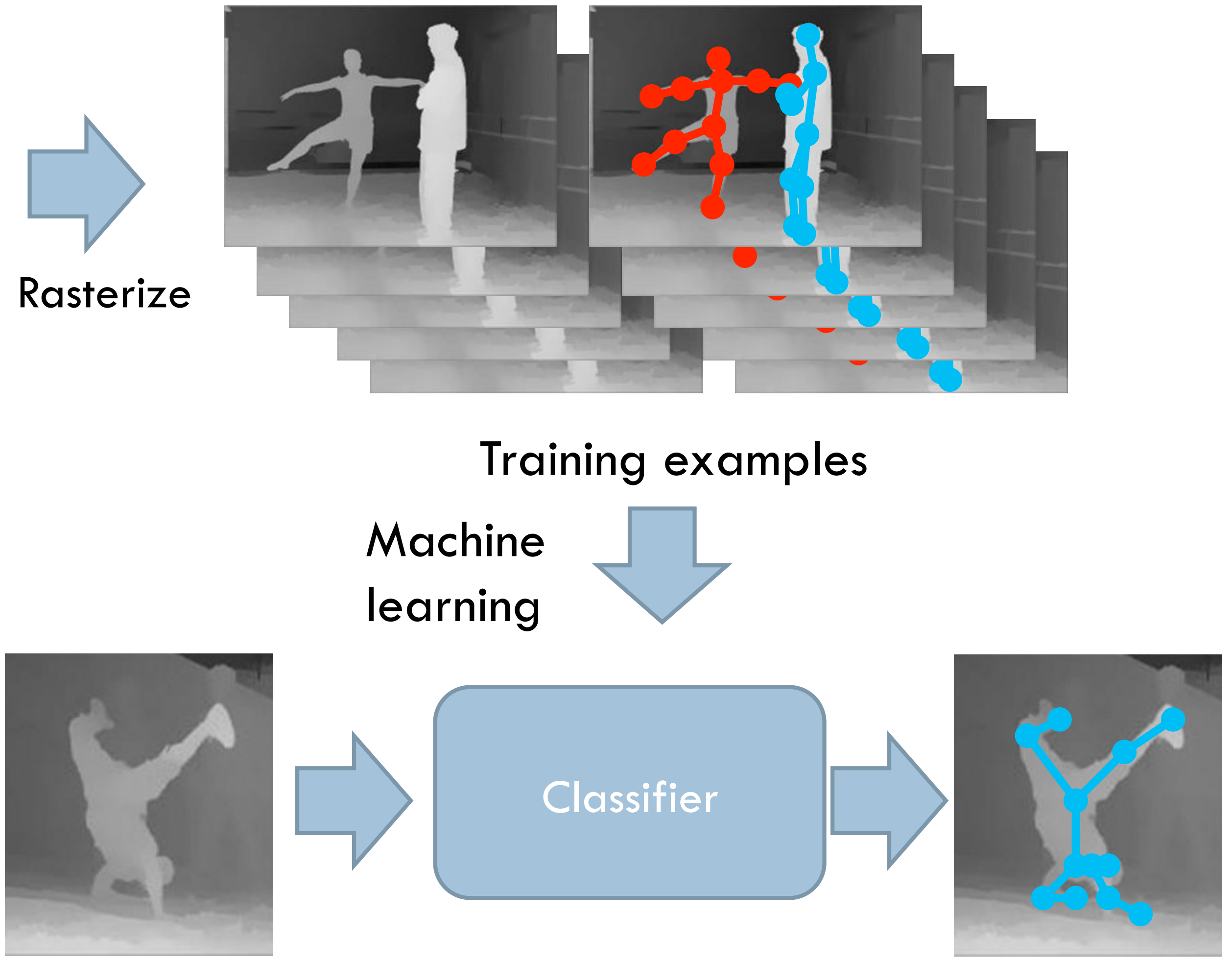
grep^{1000} | sed^{500} | sort^{1000} | awk^{500} | perl^{50}



Real Example



Motion Capture
(ground truth)



Large-Scale Machine Learning

- $> 10^{22}$ objects
- Sparse, multi-dimensional data structures
- Complex datatypes
  (images, video, matrices, etc.)
- Complex application logic and dataflow
  - >35000 lines of .Net
  - 140 CPU days
  - $> 10^5$ processes
  - 30 TB data analyzed
  - 140 avg parallelism (235 machines)
  - 300% CPU utilization (4 cores/machine)

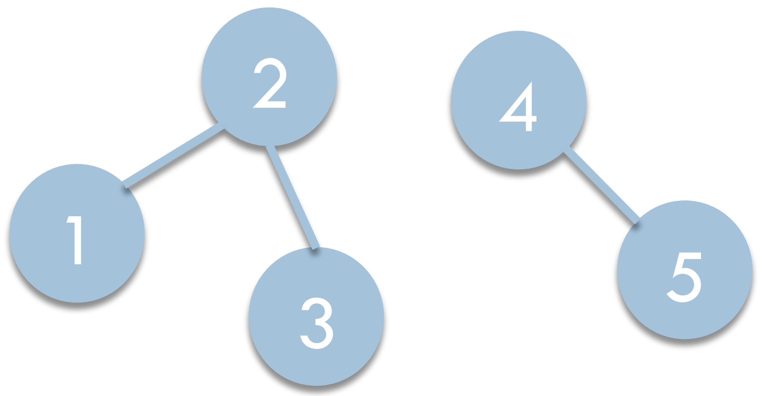
Result: XBOX Kinect

Connected components

- Input format:
  - *X Y* (meaning that node *X* links to node *Y*)
- Iterative Algorithm:
  1. Initially node *X* belongs to component with id *X*
  2. Node *X* sends to its neighbours its own component id
  3. Node *X* receives a list of component ids and keeps the minimum
  4. Repeat until convergence
- Output:
  - *X* C (meaning that *X* belongs to connected component C)
- *Note: complexity is* O*(d), where d is the diameter*

[U. Kang, Charalampos E. Tsourakakis, and C. Faloutsos. 2009. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In Proc. of the 2009 Ninth IEEE Int. Conf. on Data Mining (ICDM '09).]

Connected components



	Node 1	Node 2	Node 3	Node 4	Node 5
Iter 0	1	2	3	4	5
Iter 1	(1) 2 1	(2) 1 3 1	(3) 2 2	(4) 5 4	(5) 4 4
Iter 2	(1) 1 1	(1) 1 2 1	(2) 1 1	(4) 4 4	(4) 4 4
Iter 3	(1) 1 **1**	(1) 1 1 **1**	(1) 1 **1**	(4) 4 **4**	(4) 4 **4**

Connected Components: Hadoop Implementation

```
public static class Pegasus_Mapper_first
   extends Mapper<LongWritable, Text,
                  LongWritable, LongArrayWritable> {

  // Extend the Class Mapper
  // The four generic type are resp.
  //   - the input key type
  //   - the input value type
  //   - the output key type
  //   - the output value type
  // Any key should implement WritableComparable
  // Any value should implement Writable
```

Connected Components: Hadoop Implementation

```
public static class Pegasus_Mapper_first
   extends Mapper<LongWritable, Text,
                  LongWritable, LongArrayWritable> {
 @Override
 protected void map(LongWritable key, Text value,
                  Context context)
            throws IOException, InterruptedException {


 // Override the map method
 //   - by default it implements identity
 // Context provides the emit function
 //   - and some other useful stuff
```

Connected Components: Hadoop Implementation

```
… … …
String [] values = value.toString().split(" ");
LongWritable node  = new LongWritable( Long.parseLong(values[0]) );
LongWritable neigh = new LongWritable( Long.parseLong(values[1]) );

// Read the pair of nodes from the input
```

Connected Components: Hadoop Implementation

```
… … …
String [] values = value.toString().split(" ");
LongWritable node  = new LongWritable( Long.parseLong(values[0]) );
LongWritable neigh = new LongWritable( Long.parseLong(values[1]) );

LongWritable[] singlet = new LongWritable[1];
singlet[0] = neigh;
context.write(node, new LongArrayWritable(singlet) );

// Emit the pair <node, neighbor>
// i.e. tell to node who is its neighbor
// otherwise it will not know its neighbors in the following
   iterations
```

Connected Components: Hadoop Implementation

```
… … …
String [] values = value.toString().split(" ");
LongWritable node  = new LongWritable( Long.parseLong(values[0]) );
LongWritable neigh = new LongWritable( Long.parseLong(values[1]) );

LongWritable[] singlet = new LongWritable[1];
singlet[0] = neigh;
context.write(node, new LongArrayWritable(singlet) );

singlet[0] = new LongWritable(-1-node.get());
context.write(node, new LongArrayWritable(singlet) );
context.write(neigh, new LongArrayWritable(singlet) );

// Tell to the neighbor and to the node itself
//      what is the currently know smallest component id
// The component ids are made negative (-1)
```

Connected Components: Hadoop Implementation

```
public static class Pegasus_Reducer
  extends Reducer<LongWritable, LongArrayWritable,
                   LongWritable, LongArrayWritable> {
 @Override
 protected void reduce(LongWritable key,
                   Iterable<LongArrayWritable> values,
                    Context context)
     throws IOException, InterruptedException {

 // Extend the Class Reducer
 // Override the reduce method
 //  ( similarly to what we did for the mapper)

 // Note: reducer receives a list (Iterable) of values
```

Connected Components: Hadoop Implementation

```
… … …
LongWritable min = null;
Writable[] neighbors = null;
for(LongArrayWritable cluster : values) {
   Writable[] nodes = cluster.get();
   LongWritable first = (LongWritable) nodes[0];
   if (first.get()<0) { // This is a min !
        if (min==null) min = first;
        else if (min.compareTo(first)<0) {
                min = first;
        }
   } else { … … … …

// Scan the list of values received.
// Each value is an array of ids.
// If the first element is negative,
//    this message contains a component id
//    Keep the smallest! (absolute value)
```

Connected Components: Hadoop Implementation

```
… … …
} else { // This is the actual graph
   if (neighbors == null) neighbors = nodes;
   else {
        Writable[] aux = new Writable[neighbors.length +
                                                nodes.length];
        System.arraycopy(neighbors, 0, aux, 0, neighbors.length);
        System.arraycopy(nodes, 0, aux, neighbors.length,
                                                              nodes.length);
        neighbors = aux;
   }
}
// If we received graph information
// (potentially from many nodes).
// store it a new array
```

Connected Components: Hadoop Implementation

```
… … …
int num_neigh = neighbors==null ? 0 : neighbors.length;
LongWritable[] min_nodes = new LongWritable[num_neigh + 1];
min_nodes[0] = min;
for (int i=0; i<num_neigh; i++)
   min_nodes[i+1] = (LongWritable) neighbors[i];

// send current min + graph
context.write(key, new LongArrayWritable(min_nodes));

// Create a vector where
// The first position is the current component id
// The rest is the list of neighbors
// “send” this information to the node
```

Connected Components: Hadoop Implementation

```
public static class Pegasus_Mapper
     extends Mapper< LongWritable, LongArrayWritable,
                        LongWritable, LongArrayWritable> {
 @Override
     protected void map(LongWritable key,
 LongArrayWritable value,
                                   Context context)

 // In subsequent iterations:
 // the mapper receives the vector with the component id
 // and the set of neighbors.
 //  - it propagates the id to the neighbors
 //  - send graph information to the node
```

Connected Components: Hadoop Implementation

```
Path iter_input_path = null;
Path iter_output_path = null;
int max_iteration = 30;

// Pegasus Algorithm invocation
Configuration configuration = new Configuration();
FileSystem fs = FileSystem.get(configuration);
for (int iteration = 0; iteration<=max_iteration; iteration++ ) {
   Job jobStep = new Job(configuration, “PEGASUS iteration " +
                                                    iteration);
   jobStep.setJarByClass(MRConnectedComponents.class);


// A MapReduce Job is defined by a Job object
// In each iteration we must update the “fields” of such object
```

Connected Components: Hadoop Implementation

```
// common settings
iter_input_path = new Path(intermediate_results, "iter"+iteration);
iter_output_path = new Path(intermediate_results, "iter"+
                                                    (iteration+1));

Class mapper_class        = Pegasus_Mapper.class;
Class reducer_class       = Pegasus_Reducer.class;
Class output_class        = LongArrayWritable.class;
Class output_format_class = SequenceFileOutputFormat.class;
Class input_format_class  = SequenceFileInputFormat.class;


// Some parameters are the same at each iteration
```

Connected Components: Hadoop Implementation

```
// per iteration settings
if (iteration==0) {
   iter_input_path     = input_dir;
   mapper_class        = Pegasus_Mapper_first.class;
   input_format_class  = TextInputFormat.class;
} else if (iteration == max_iteration) {
   mapper_class        = Pegasus_Outputter.class;
   reducer_class       = null;
   iter_output_path    = output_dir;
   output_format_class = TextOutputFormat.class;
   output_class        = LongWritable.class;
}

// The first and the last iterations are different
```

Connected Components: Hadoop Implementation

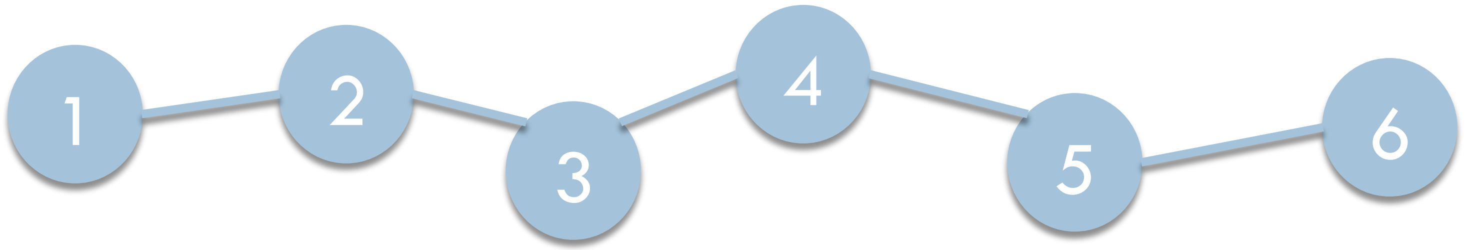
```
jobStep.setMapperClass(mapper_class);
if (reducer_class!=null) { jobStep.setReducerClass(reducer_class); }

jobStep.setOutputKeyClass(LongWritable.class);
jobStep.setOutputValueClass(output_class);

jobStep.setInputFormatClass(input_format_class);
jobStep.setOutputFormatClass(output_format_class);

FileInputFormat.setInputPaths(jobStep, iter_input_path);
FileOutputFormat.setOutputPath(jobStep, iter_output_path);

boolean success = jobStep.waitForCompletion(true);

// Set all parameters of the job and launch
```

Connected components: Hash-To-Min

- Iterative Algorithm:
  1. Initially node *X* “knows” cluster C=X *plus its neighbors*
  2. Node X sends C to the smallest node in C
  3. Node X sends the smallest node of C to any other node in C
  4. Node X receives creates a new C by merging all the received messages
  5. Repeat until convergence

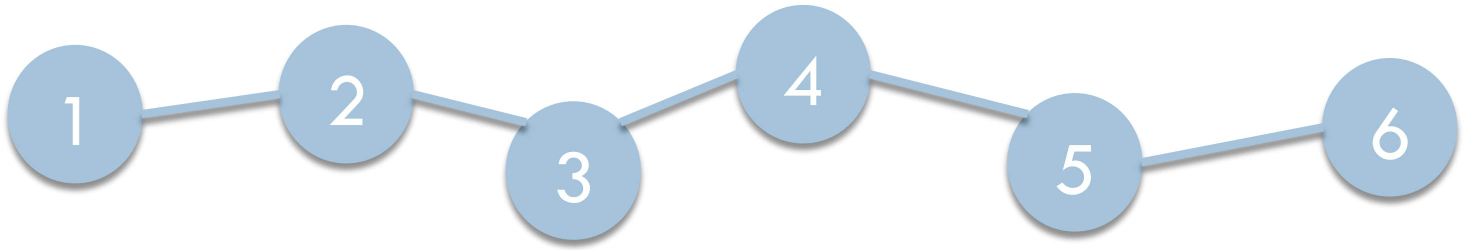
[Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, Anish Das Sarma: Finding Connected Components on Map-reduce in Logarithmic Rounds. CoRR abs/1203.5387 (2012).]

Connected components: Hash-To-Min



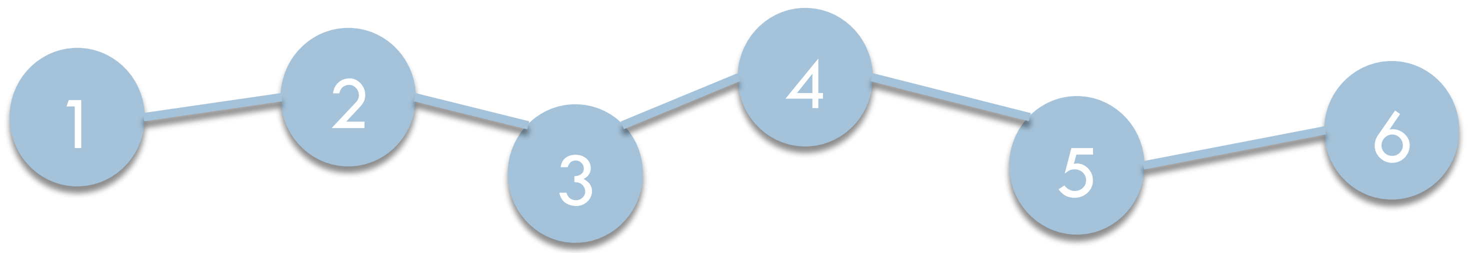
	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
Iter 0	1,2	1,2,3	2,3,4	3,4,5	4,5,6	5,6
Iter 1	(1,2)	1				
	(1,2,3)	1	1			
		(2,3,4)	2	2		
			(3,4,5)	3	3	
				(4,5,6)	4	4
					5,6	5
	1,2,3	1,2,3,4	1,2,3,4,5	2,3,4,5,6	3,4,5,6	4,5

Connected components: Hash-To-Min



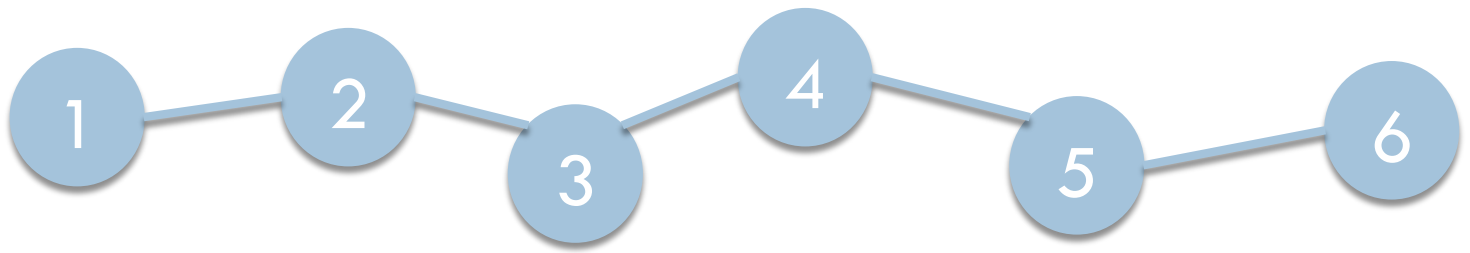
	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
Iter 1	1,2,3	1,2,3,4	1,2,3,4,5	2,3,4,5,6	3,4,5,6	4,5
Iter 2	(1,2,3)	1	1			
	(1,2,3,4)	1	1	1		
	(1,2,3,4,5)	1	1	1	1	
		(2,3,4,5,6)	2	2	2	2
			(3,4,5,6)	3	3	3
				(4,5)	4	
	1,2,3,4,5	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5	1,2,3,4	2,3

Connected components: Hash-To-Min



	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
Iter 2	1,2,3,4,5	1,2,3,4,5,6	1,2,3,4,5,6	1,2,3,4,5	1,2,3,4	2,3
Iter 3	(1,2,3,4,5)	1	1	1	1	
	(1,2,3,4,5,6)	1	1	1	1	1
	(1,2,3,4,5,6)	1	1	1	1	1
	(1,2,3,4,5)	1	1	1	1	
	(1,2,3,4)	1	1	1		
		(2,3)	2			
	1,2,3,4,5,6	1,2,3	1,2	1	1	1

Connected components: Hash-To-Min



	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6
Iter 3	1,2,3,4,5,6	1,2,3	1,2	1	1	1
Iter 4	(1,2,3,4,5,6)	1	1	1	1	1
	(1,2,3)	1	1			
	(1,2)	1				
	1					
	1					
	1					
	1,2,3,4,5,6	**1**	**1**	**1**	**1**	**1**

- *Note: complexity is O(log d), where d is the diameter*

… The end

If you are interested in the following topics:

- large-scale data processing
- data mining
- web search and mining

feel free to contact me/fabrizio at
claudio.lucchese@isti.cnr.it
fabrizio.silvestri@isti.cnr.it