Cloud Computing

MapReduce and Hadoop

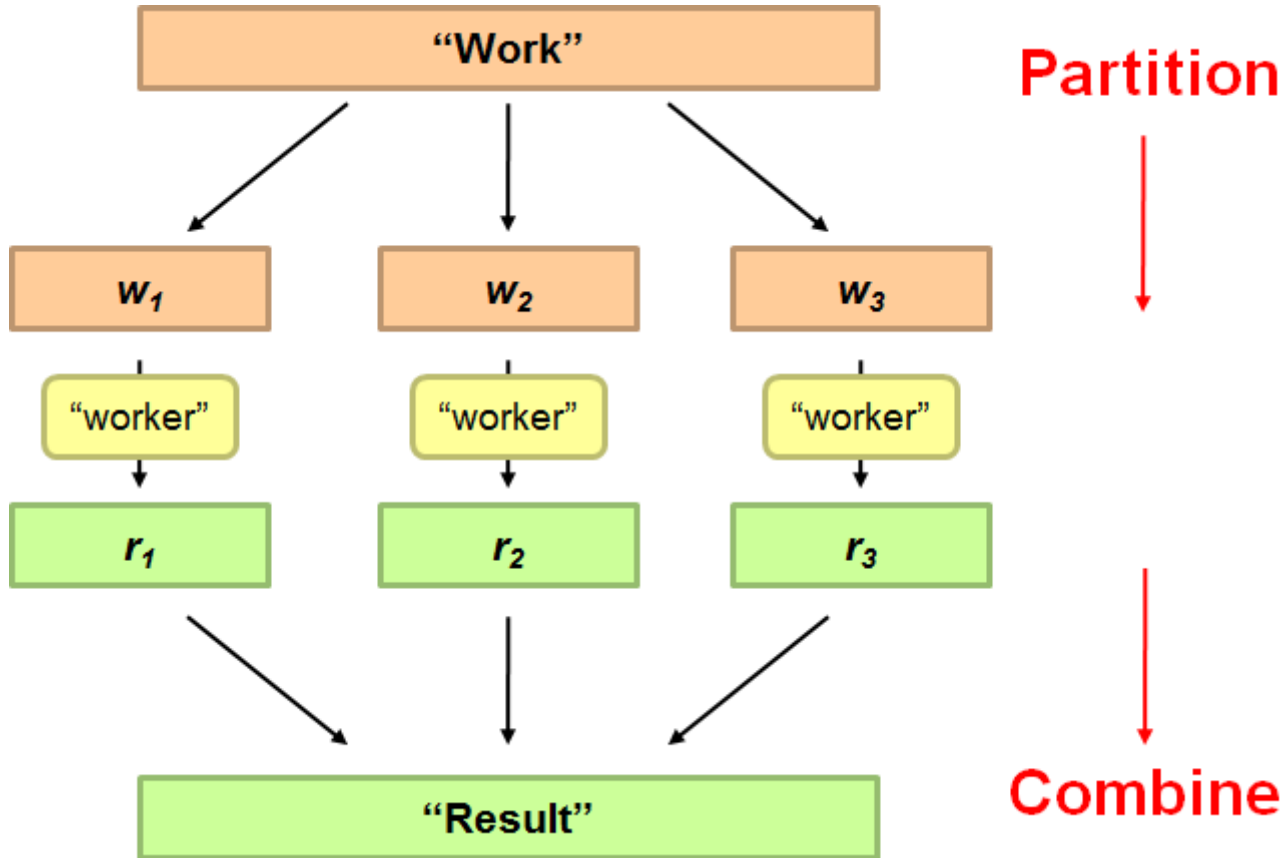## Dell Zhang

Birkbeck, University of London

2018/19

MapReduce Basics

Map

Reduce

Divide and Conquer

Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

Common Theme?

- Parallelization problems arise from:
    - Communication between workers (e.g., to exchange state)
    - Access to shared resources (e.g., data)
- Thus, we need a **synchronization** mechanism

Source: Ricardo Guimarães Herrmann

Managing Multiple Workers

- Difficult because
  - We don't know the order in which workers run
  - We don't know when workers interrupt each other
 - We don't know the order in which workers access shared data

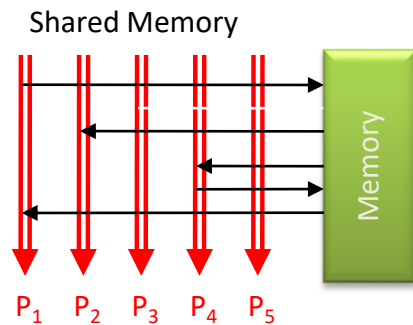# Managing Multiple Workers

- Thus, we need:
  - semaphores (lock, unlock)
  - conditional variables (wait, notify, broadcast)
  - barriers
- Still, lots of problems:
  - deadlock, livelock, race conditions…
  - dining philosophers, sleeping barbers, cigarette smokers…
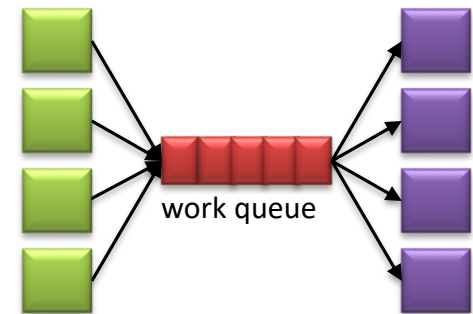- Moral of the story: be careful!
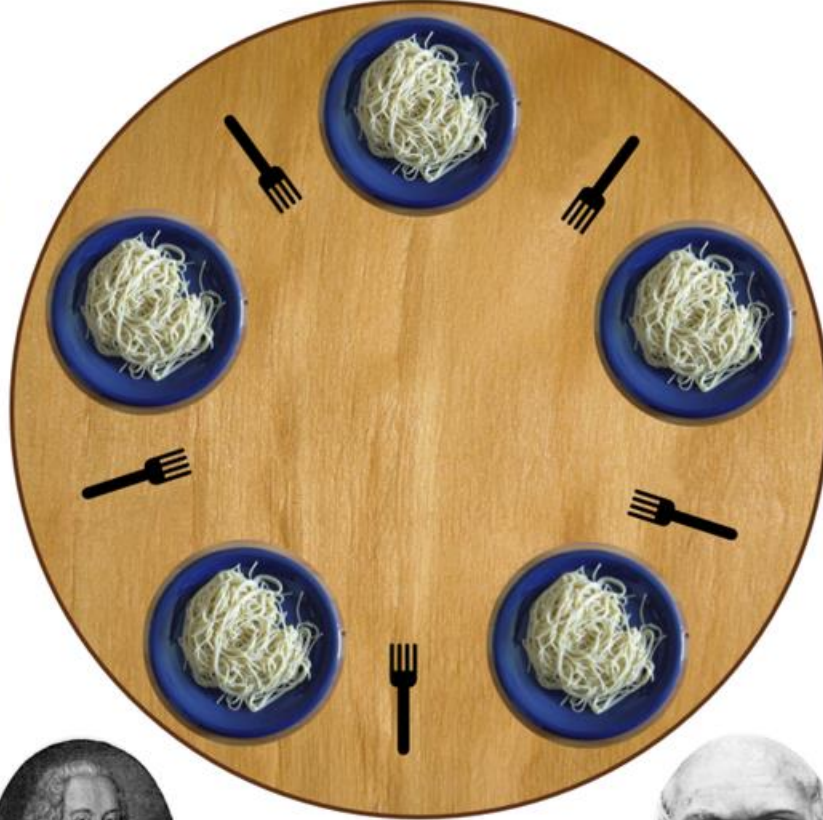
Current Tools

- Programming models
  - Shared Memory (pthreads)
  - Message Passing (MPI)
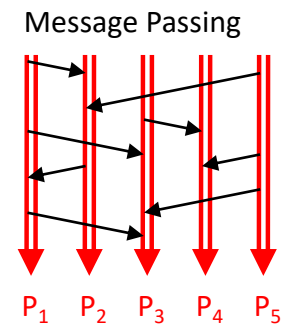
Current Tools

- Design Patterns
  - Master-Slaves
  - Producer-Consumer Flows
  - Shared Work Queues

master

slaves

producer   consumer

producer   consumer

work queue
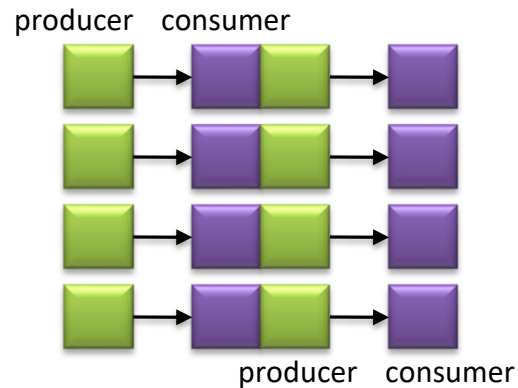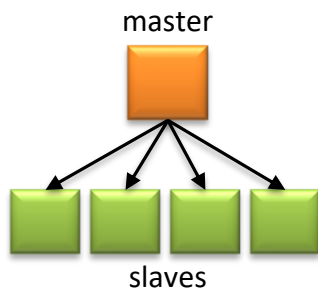
Where the rubber meets the road

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters (even across datacenters)
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…

Where the rubber meets the road

- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

Fallacies of Distributed Computing

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

What's the point?

- It's all about the right level of abstraction
  - The von Neumann architecture has served us well, but is no longer appropriate for the multi-core or cluster environment.
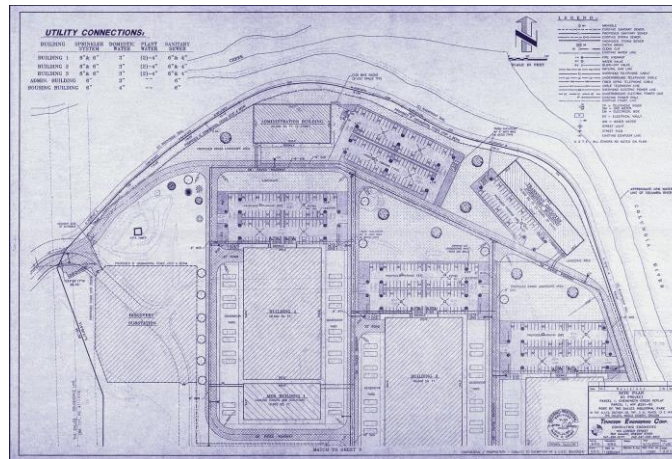
The datacenter **is** the computer!

What's the point?

- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the *what* from *how*
  - Developer specifies the computation that needs to be performed
  - Execution framework ("runtime") handles actual execution

"Big Ideas"

- *Scale "out", not "up"*
  - Limits of SMP and large shared-memory machines
- *Move processing to the data*
  - Cluster have limited bandwidth
- *Process data sequentially, avoid random access*
  - Seeks are expensive, disk throughput is reasonable
- *Seamless scalability*
  - From mythical man-month to tradable machine-hour

Warm-Up

- The task:
  - We have a huge text document
  - Count the number of times each distinct word appears in the file

- Sample application:
  - Analyse web server logs to find popular URLs

Warm-Up

- In UNIX, it can be easily done:

  `words(doc.txt) | sort | uniq -c`

  - Here `words` is a script that takes a file and outputs the words in it, one per a line.

  - The file `doc.txt` may be too large for memory, but all <word, count> pairs fit in memory

  - The great thing is that it is naturally parallelizable

  - This captures the essence of MapReduce

Typical Big-Data Problem

- Iterate over a large number of records
- Extract something of interest from each

    Map

- Shuffle and sort intermediate results
- Aggregate intermediate results

    Reduce

- Generate final output

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

MapReduce overview

- Read a lot of data, sequentially

- **Map**:

  – Extract something you care about

- Shuffle and Sort

  – Group by key

- **Reduce**:

  – Aggregate, summarize, filter or transform

- Write the result

Outline stays the same, **map** and **reduce** change to fit the problem.

The **Map** Step
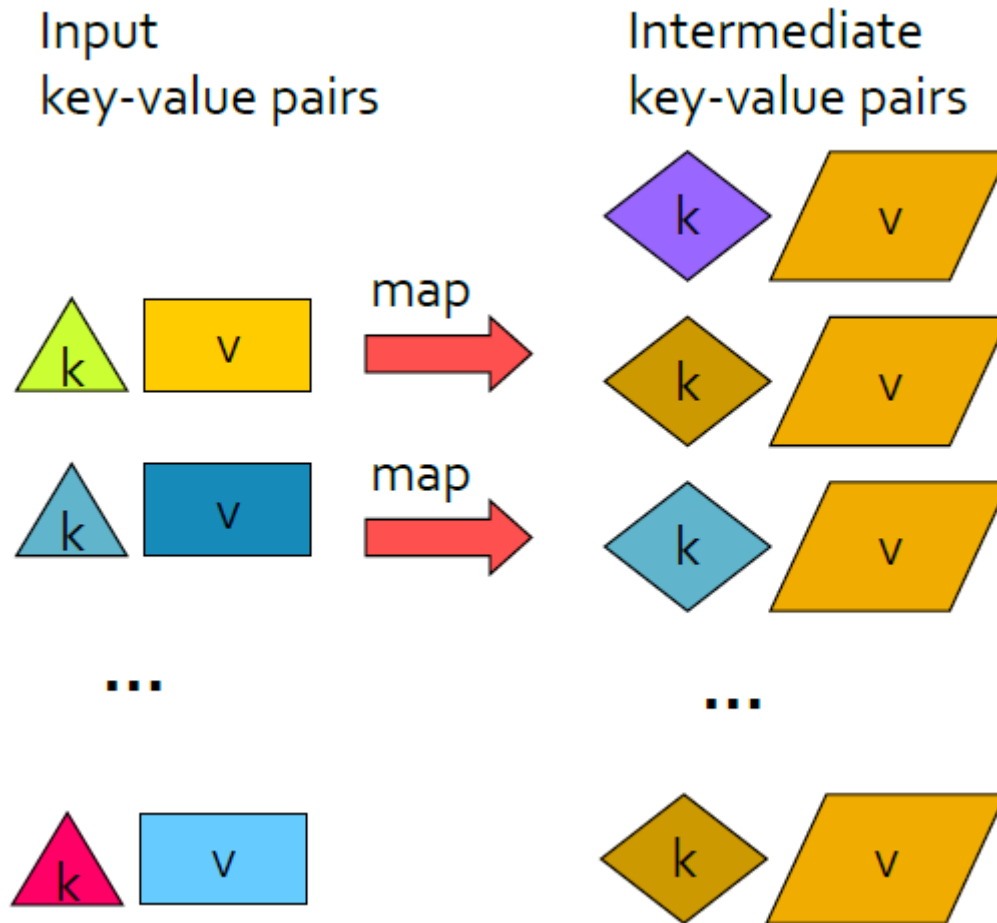
The **Reduce** Step
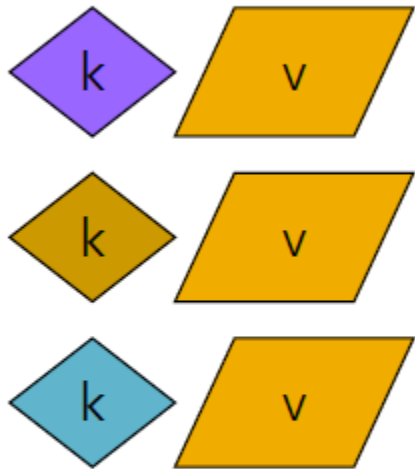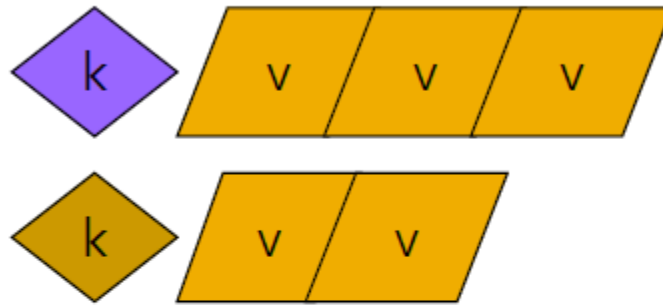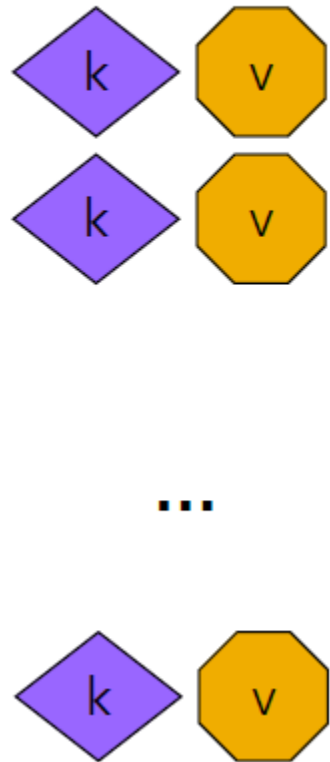
Roots in *Functional* Programming
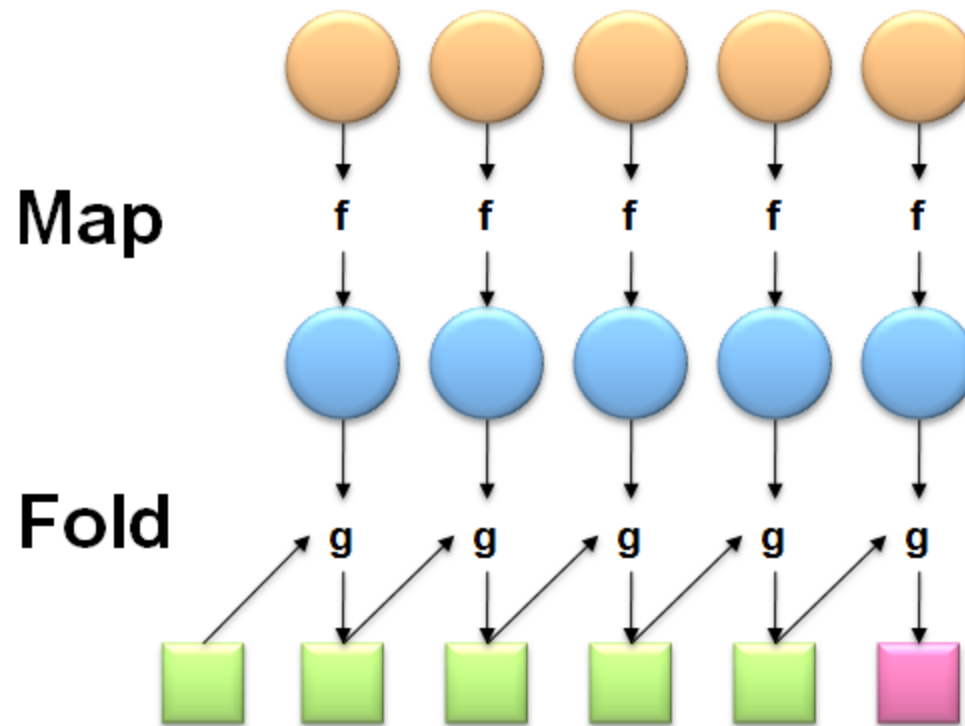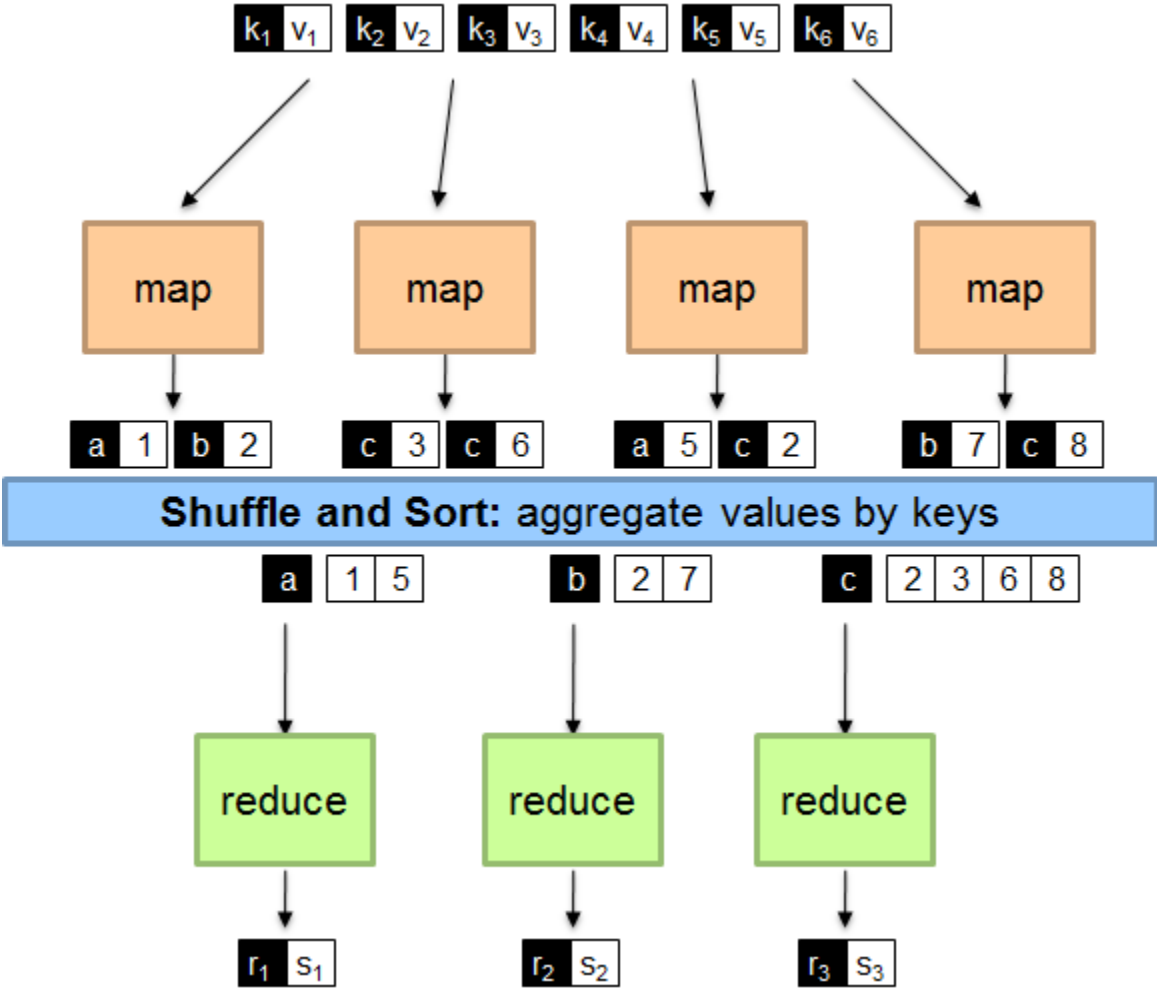
MapReduce Programming Model

- Programmers must specify two functions:

  map (k, v) → <k', v'>*
  - Takes a key value pair and outputs a set of key value pairs, e.g., key = filename, value = the file content
  - There is one Map call for every (k,v) pair

  reduce (k', <v'>*) → <k', v''>*
  - All values v' with same key k' are reduced together and processed in v' order
  - There is one Reduce function call per unique key k'

- The execution framework handles everything else

	Input	Big document						

MAP: reads input and produces a set of key value pairs

M M M M M M M

Intermediate: k1:v k1:v k2:v | k1:v | k3:v k4:v | k4:v k5:v | k4:v | k1:v k3:v

Group by key: Collect all pairs with same key

Group by Key

Grouped: k1:v,v,v,v | k2:v | k3:v,v | k4:v,v,v | k5:v

Reduce: Collect all values belonging to the key and output

R R R R R
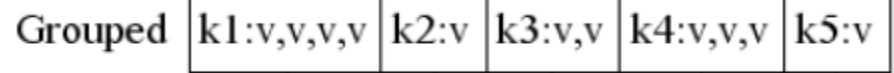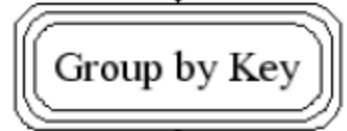
Output

Execution in Parallel
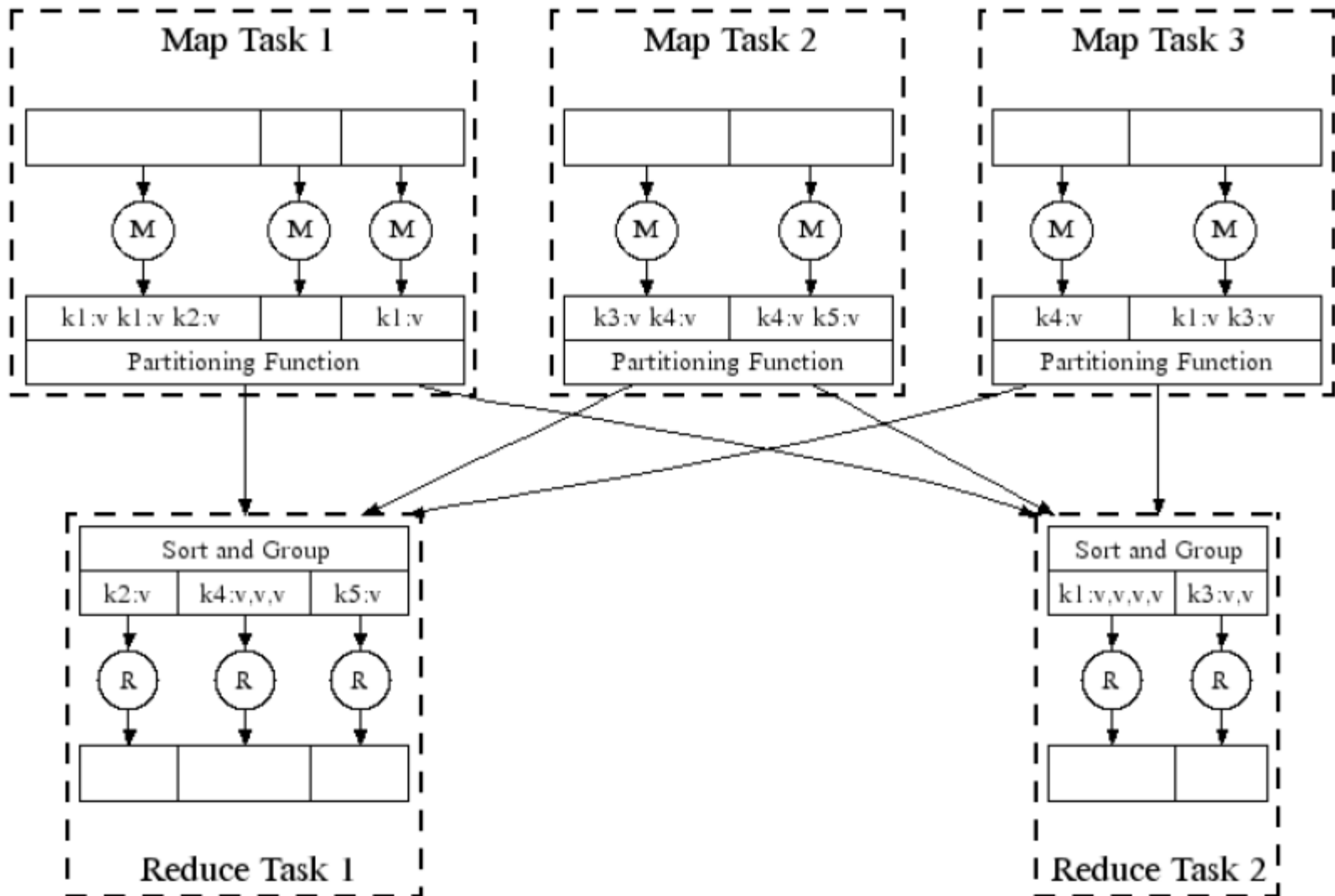
Two More Details…

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

"Hello World": Word Count

Provided by the programmer

MAP:
reads input and produces a set of key value pairs

Provided by the programmer

Group by key:
Collect all pairs with same key

Reduce:
Collect all values belonging to the key and output

The crew of the space shuttle Endeavor recently returned to Earth as ambassadors, harbingers of a new era of space exploration. Scientists at NASA are saying that the recent assembly of the Dextre bot is the first step in a long-term space-based man/machine partnership. "'The work we're doing now -- the robotics we're doing -- is what we're going to need to do to build any work station or habitat structure on the moon or Mars," said Allard Beutel.

Big document

(the, 1)
(crew, 1)
(of, 1)
(the, 1)
(space, 1)
(shuttle, 1)
(Endeavor, 1)
(recently, 1)
....

(key, value)

(crew, 1)
(crew, 1)
(space, 1)
(the, 1)
(the, 1)
(the, 1)
(shuttle, 1)
(recently, 1)
...

(key, value)

(crew, 2)
(space, 1)
(the, 3)
(shuttle, 1)
(recently, 1)
...

(key, value)

Only sequential reads

"Hello World": Word Count

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


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

"Hello World": Word Count

Input	Map	Shuffle & Sort	Reduce	Output

the quick brown fox → Map

the, 1
brown, 1
fox, 1

the fox ate the mouse → Map

the, 1
fox, 1
the, 1

quick, 1

how now brown cow → Map

how, 1
now, 1
brown, 1

ate, 1
mouse, 1

cow, 1

Reduce →

brown, 2
fox, 2
how, 1
now, 1
the, 3

Reduce →

ate, 1
cow, 1
mouse, 1
quick, 1
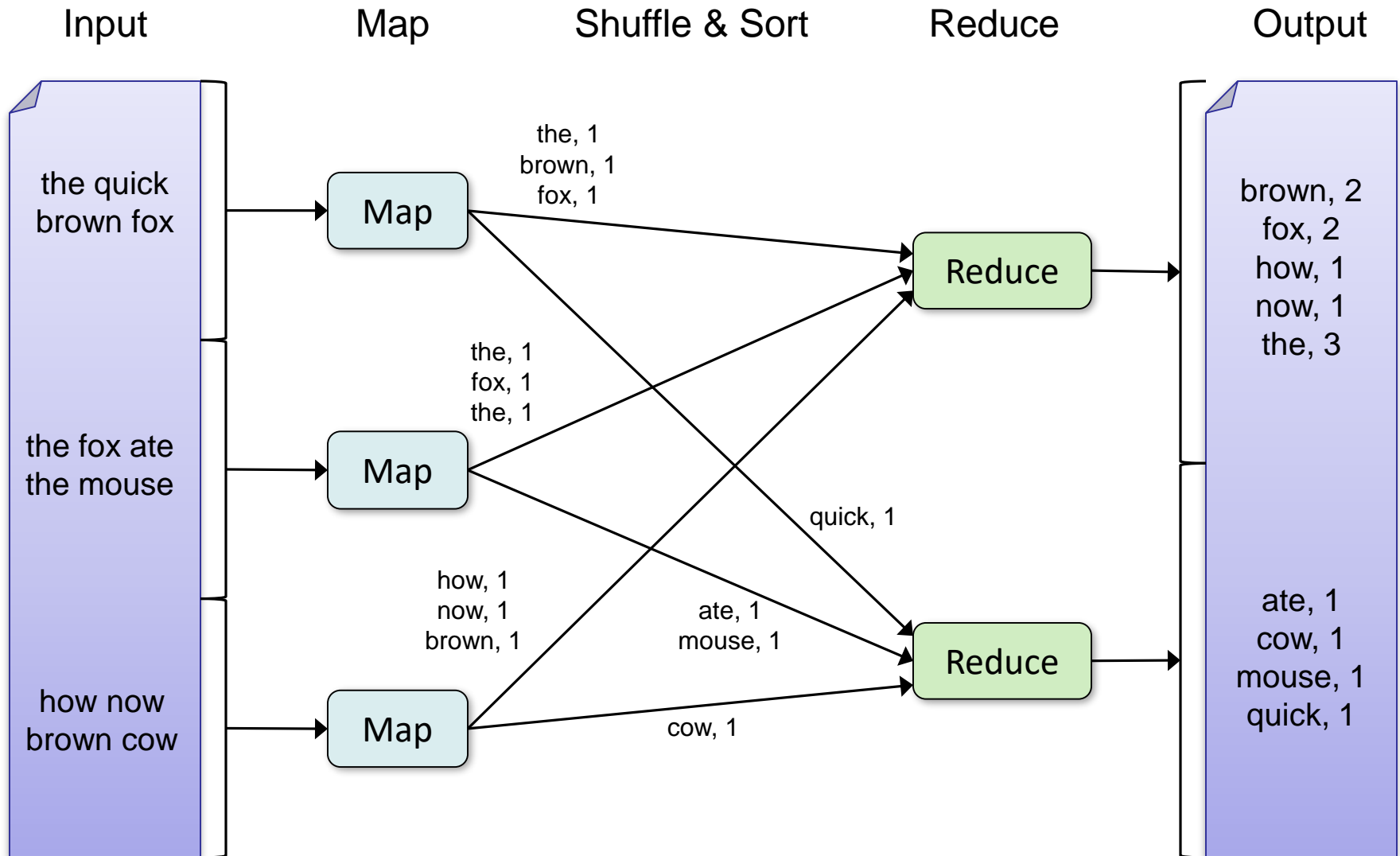
Exercise

- Information Retrieval (TFxIDF):

  – Given a large document collection, for every word, we need to count the number of documents containing it (i.e., document frequency)

- Statistical Machine Translation (5-gram):

  – Given a large document collection, for every 5-word sequence, we need to count the number of times it occurs in the whole collection

What's "everything else"?

- MapReduce "runtime" environment handles
  - **scheduling**: assigns workers to map and reduce tasks
  - **data distribution**: partitions the input data and moves processes to data
  - **synchronization**: manages required inter-machine communication to gather, sort, and shuffle intermediate data
  - **errors and faults**: detects worker failures and restarts

    Everything happens on top of a DFS

Data Flow

- The input and final output are stored on a distributed file system

  - Scheduler tries to schedule map tasks "close" to physical storage location of input data

- The intermediate results are stored on the local file system of map and reduce workers

- The output is often the input to another MapReduce task

Coordination

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

Dealing with 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
- Master failure
  - MapReduce task is aborted and client is notified

How many Map and Reduce tasks?

- M map tasks, R reduce tasks
- Rule of a thumb:
  - Make M and R much 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
  - because output is spread across R files

Task Granularity

- Fine granularity tasks: map tasks >> machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution
  - Better dynamic load balancing

Process	Time ---------------------->						
User Program	MapReduce() ... wait ...						
Master	Assign tasks to worker machines...						
Worker 1		Map 1	Map 3				
Worker 2		Map 2					
Worker 3		Read 1.1	Read 1.3	Read 1.2			Reduce 1
Worker 4		Read 2.1			Read 2.2	Read 2.3	Reduce 2
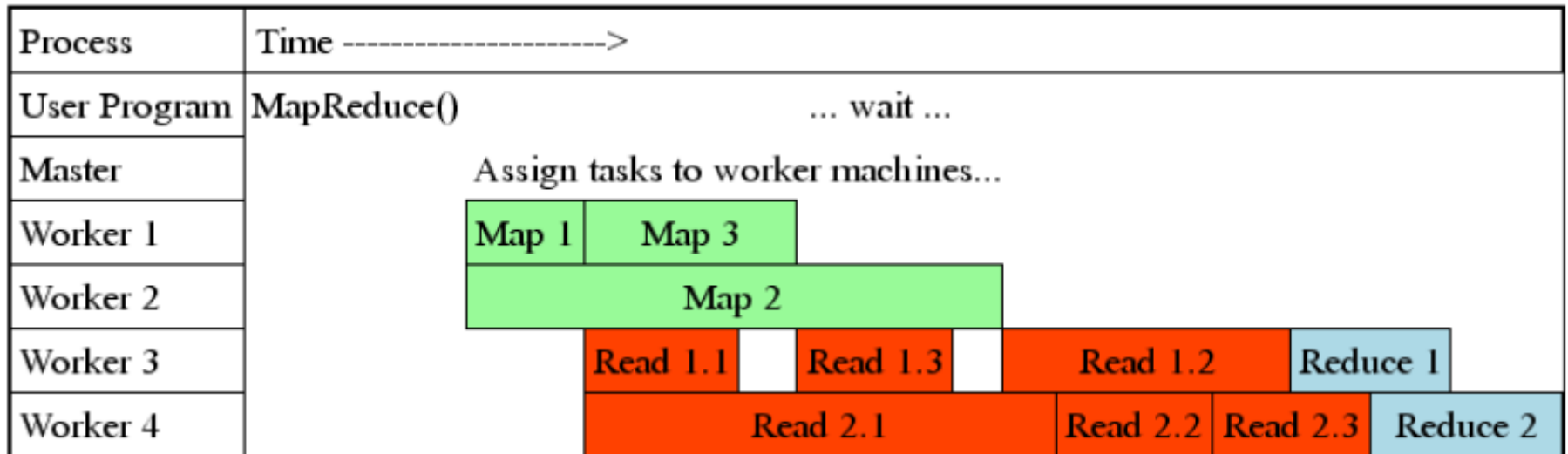
Refinement: Combine & Partition

- Really, "everything else"?
- Not quite... often, programmers also specify:
  combine (k', v') → <k', v'>*
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
  partition (k', #partitions) → partition for k'
    - Often a simple hash of the key, e.g., hash(k') mod n
    - Divides up key space for parallel reduce operations

Combine Function

- A map task can produce many pairs with the same key: (k,v1), (k,v2), …
  - e.g., popular words in the WordCount example
  - They need to be sent over the network to the reducer: costly
- It is often possible to pre-aggregate these pairs into a single key-value pair at the mapper
  - Decreases the size of intermediate data and thus save network time

Combine Function

- The combiner is executed to combine the values for a given key
  - e.g., (jaguar,1), (jaguar, 1), (jaguar, 1), (jaguar, 2) $\rightarrow$ (jaguar, 5)
  - The combiner can be called several times, or not at all
- Easy case: the combine function is the same as the reduce function
  - Works only if the reduce function is commutative and associative.

Word Count with *Combine*

Input	Map & Combine	Shuffle & Sort	Reduce	Output

the quick brown fox

the fox ate the mouse

how now brown cow

Map

Map

Map

the, 1
brown, 1
fox, 1

the, 2
fox, 1

how, 1
now, 1
brown, 1

quick, 1

ate, 1
mouse, 1

cow, 1

Reduce

Reduce

brown, 2
fox, 2
how, 1
now, 1
the, 3

ate, 1
cow, 1
mouse, 1
quick, 1
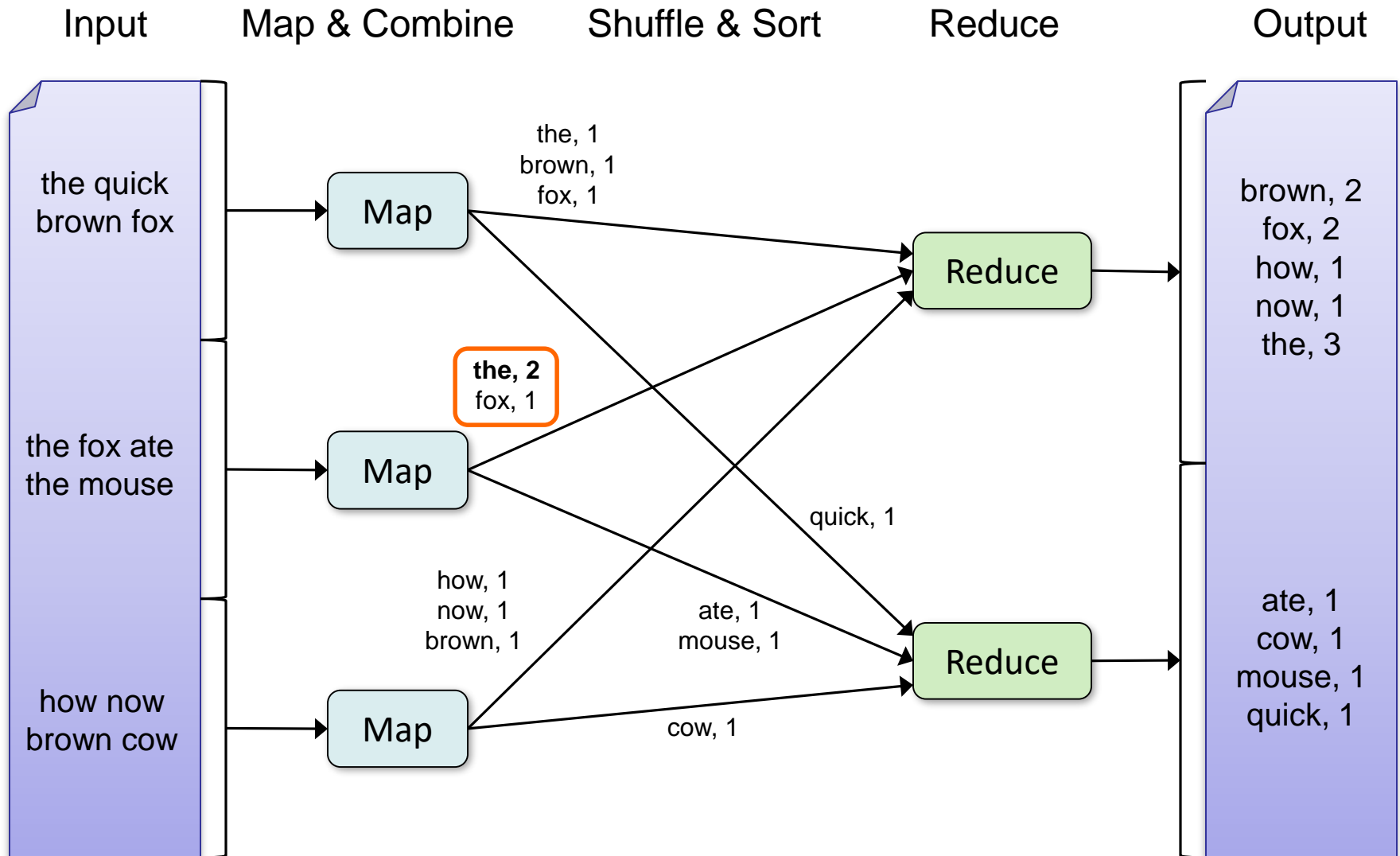
Partition Function

- Inputs to map tasks are simply created by contiguous splits of input file, but reduce tasks need to ensure that records with the same intermediate key end up at the same worker
  - System uses a default partition function: hash(key) mod *R*
  - Sometimes useful to override: e.g., hash(hostname(URL)) mod *R* ensures URLs from one host end up in the same output file

| k_1 | v_1 | k_2 | v_2 | k_3 | v_3 | k_4 | v_4 | k_5 | v_5 | k_6 | v_6 |

map   map   map   map

| a | 1 | b | 2 | | c | 3 | c | 6 | | a | 5 | c | 2 | | b | 7 | c | 8 |

combine   combine   combine   combine

| a | 1 | b | 2 | | c | 9 | | a | 5 | c | 2 | | b | 7 | c | 8 |

partition   partition   partition   partition

Shuffle and Sort: aggregate values by keys

| a | 1 | 5 | | b | 2 | 7 | | c | 2 | 9 | 6 | 8 |

reduce   reduce   reduce

| r_1 | s_1 | | r_2 | s_2 | | r_3 | s_3 |
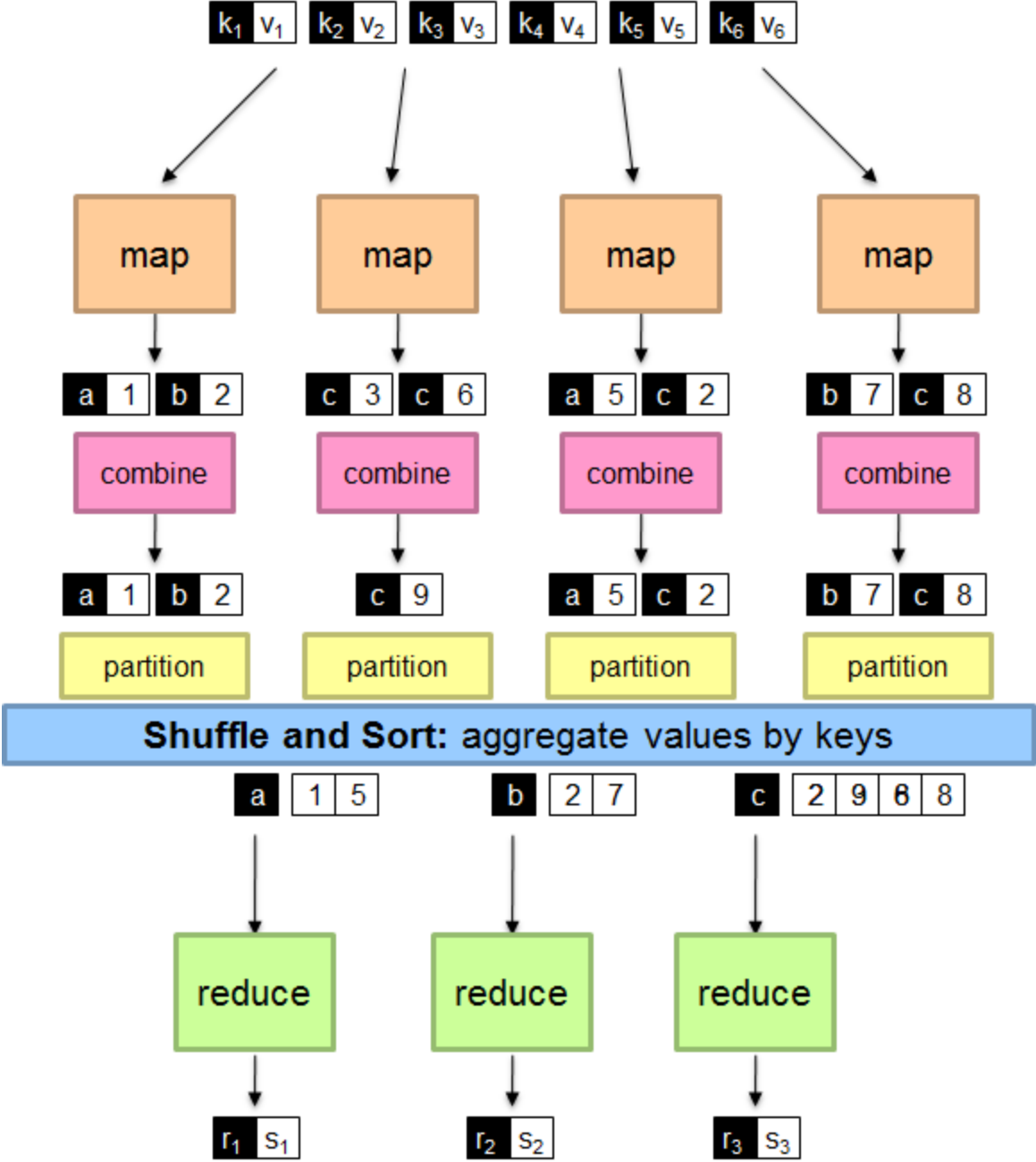
Optimisation: Compression

- Data transfers over the network:
  - From datanodes to mapper nodes (usually reduced using data locality)
  - From mappers to reducers
  - From reducers to datanodes to store the final output
- Each of these can benefit from compression
  - Trade-off between the volume of data transfer and the (de)compression time.
  - Usually, compressing map outputs using a fast compressor increases efficiency

Optimisation: Shuffle and Sort

- Sorting of pairs on each reducer to compute the groups is a costly operation
- It would be much more efficient in memory than on disk
  - Increasing the amount of memory available for shuffle operations can greatly increase the performance
  - . . . at the downside of less memory available for map and reduce tasks (but usually not much needed)

Optimisation: Speculative Execution

- The MapReduce jobtracker tries detecting tasks that take longer than usual
  - Problem: Such slow workers (e.g., due to hardware problems or heavy workload of the machine …) would significantly lengthen the job completion time.
  - Solution: When detected, *backup copies of the task* would be spawn and ***speculatively*** executed, without discarding the existing task. Eventually, whichever one finishes first "wins" and the others will be killed.
  - Effect: The job completion time could be dramatically shortened.

MapReduce can refer to…

- The programming model

- The execution framework (aka "runtime")

- The specific implementation

Usage is usually clear from the context!

MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, used in production
  - Now an Apache project
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.

User
Program

(1) submit

Master

(2) schedule map   (2) schedule reduce

worker

split 0
split 1
split 2   (3) read   worker   (4) local write
split 3
split 4   (5) remote read   worker   (6) write   output
                                                                                                                                        file 0

                                   worker   worker   output
                                                                                                                                        file 1

Input   **Map**   **Intermediate files**   **Reduce**   **Output**
files   **phase**   **(on local disk)**   **phase**   **files**

Adapted from (Dean and Ghemawat, OSDI 2004)
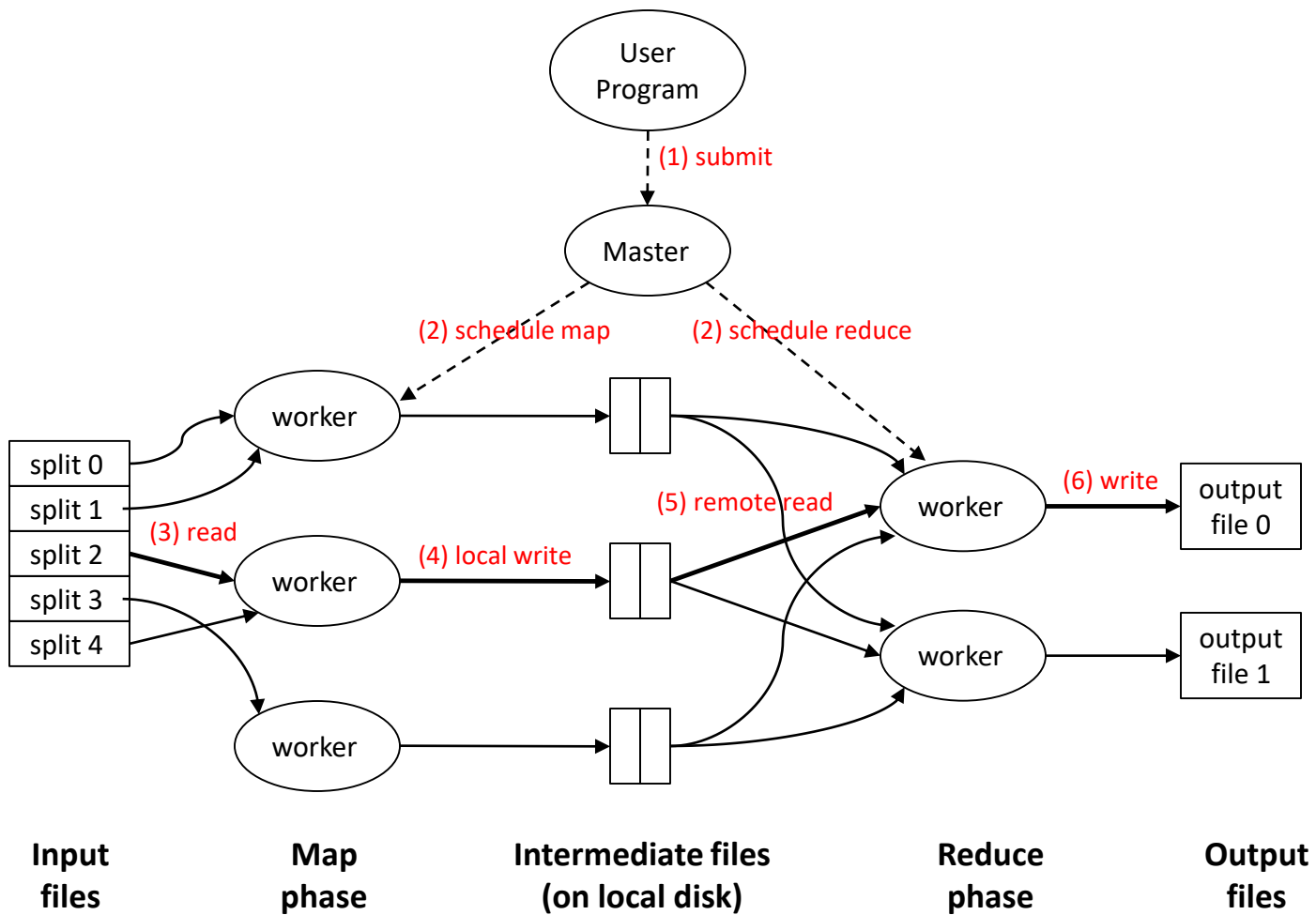
Hadoop Basics

Hadoop

- Open-source implementation of Google's GFS and MapReduce
  - Java-based software
  - Managed by the Apache foundation
  - Originally developed for Apache Nutch (open-source Web search engine), a part of Apache Lucene (text indexing platform)
  - Yahoo! has been a main contributor of the development of Hadoop

Hadoop

- Components
  - Hadoop File System (HDFS)
  - MapReduce
  - Pig (data exploration), Hive (data warehousing): higher-level languages for describing MapReduce applications
  - HBase: column-oriented distributed DBMS
  - ZooKeeper: coordination service for distributed applications

2004

Paper on MapReduce inspires Doug Cutting to create his open source implementation, which names after his son's toy elephant.

2005

Dec — Nutch ported to map/reduce

2006

Jan — Doug Cutting joins Yahoo

Feb — Factored out of Nutch

April — Sorts 1.9TB on 188 nodes in 47 hours

May — Yahoo sets up research cluster

2008

Jan — Hadoop gets top-level Apache status

Feb — Available to integrate with Amazon EC2/S3

April — Wins Terrabyte sort benchmark

Aug — Runs 4000 node cluster

Source: Hadoop Illuminated

Hadoop's centre Cloudera is formed for Hadoop based services

Yahoo makes code for its Hadoop distribution available

Facebook claim largest Hadoop cluster - 21PB

MapR announces their distribution of Hadoop

Yahoo and Benchmark Capital form Hortonworks

Facebook expand to record 30PB and starts real-time Hadoop

Oracle announce their Big Data offering - 'Big Data Appliance

Hadoop 1.0 - 'Ready for Enterprise'

2009
March
June

2010

2011
May
June
July
Oct
Dec

Source: Hadoop Illuminated

Hadoop History

- 12/2004 – Google GFS paper published
- 07/2005 – Nutch uses MapReduce
- 02/2006 – Becomes Lucene subproject
- 04/2007 – Yahoo! on 1000-node cluster
- 01/2008 – An Apache Top Level Project
- 07/2008 – A 4000 node test cluster
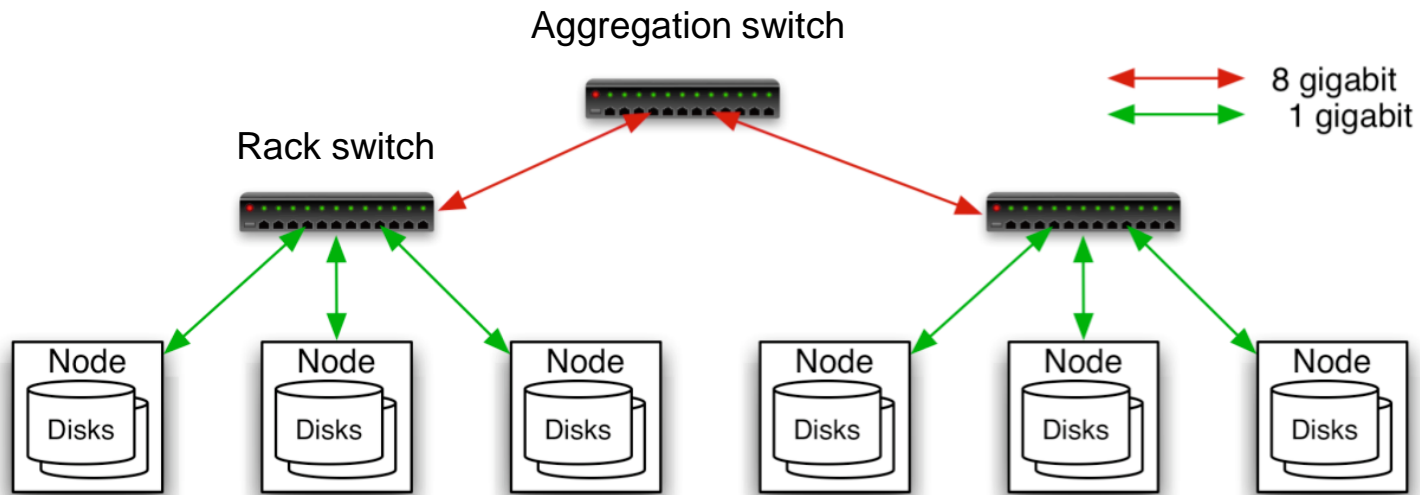- 09/2008 – Hive becomes a Hadoop subproject

Hadoop History

- 02/2009 – The Yahoo! Search Webmap is a Hadoop application that runs on more than 10,000 core Linux cluster and produces data that is now used in every Yahoo! Web search query.

- 06/2009 – On June 10, 2009, Yahoo! made available the source code to the version of Hadoop it runs in production.

- xx/2010 – In 2010 Facebook claimed that they have the largest Hadoop cluster in the world with 21 PB of storage. On July 27, 2011 they announced the data has grown to 30 PB.

Who use Hadoop?

- Amazon/A9
- Facebook
- Google
- IBM
- Joost
- Last.fm
- New York Times
- PowerSet
- Veoh
- Yahoo!
- ......

http://wiki.apache.org/hadoop/PoweredBy

Typical Hadoop Cluster

Aggregation switch

8 gigabit

1 gigabit

Rack switch

Node Disks — Node Disks — Node Disks — Node Disks — Node Disks — Node Disks

- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 GBps bandwidth in rack, 8 GBps out of rack
- Node specs (Yahoo terasort):
  8 x 2.0 GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/aw-apachecon-eu-2009.pdf

Hadoop API

- Different APIs to write Hadoop programs:
  - A rich **Java** API (main way to write Hadoop programs)
  - A Streaming API that can be used to write map and reduce functions in any programming language (using standard inputs and outputs), e.g., in Python
  - A C++ API (Hadoop Pipes)
  - With a higher language level (e.g., Pig, Hive)

Hadoop API

- ## Mapper
  - void map(K1 key, V1 value, OutputCollector<K2, V2> output, Reporter reporter)
  - void configure(JobConf job)
  - void close() throws IOException

- ## Reducer/Combiner
  - void reduce(K2 key, Iterator<V2> values, OutputCollector<K3,V3> output, Reporter reporter)
  - void configure(JobConf job)
  - void close() throws IOException

- ## Partitioner
  - void getPartition(K2 key, V2 value, int numPartitions)

WordCount.java

```java
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

    ……

}
```

WordCount.java

```java
public static class Map extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();
  public void map(
      LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      output.collect(word, one);
    }
  }
}
```

WordCount.java

```java
public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterator<IntWritable> values,
      OutputCollector<Text, IntWritable> output,
      Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

WordCount.java

```java
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
}
```
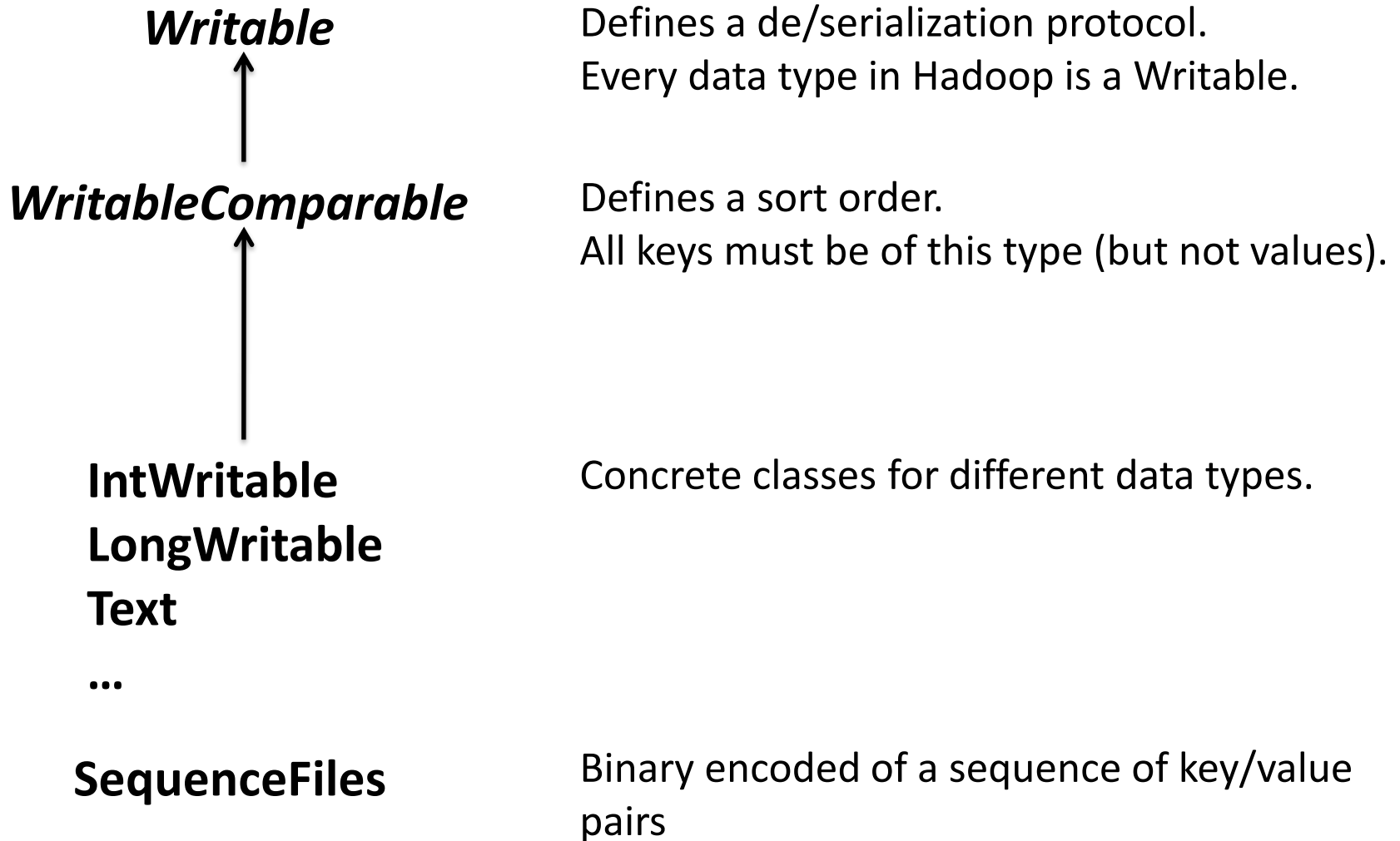
Basic Data Types in Hadoop

Writable

Defines a de/serialization protocol.
Every data type in Hadoop is a Writable.

WritableComparable

Defines a sort order.
All keys must be of this type (but not values).

IntWritable
LongWritable
Text
…

Concrete classes for different data types.

SequenceFiles

Binary encoded of a sequence of key/value pairs

Complex Data Types in Hadoop

- The easy way:
  - Encoded it as Text, e.g., (a, b) = "a:b"
  - Use regular expressions to parse and extract data
  - Works, but pretty hack-ish

Complex Data Types in Hadoop

- The hard way:
  - Define a custom implementation of *WritableComparable*
  - Must implement: *readFields*, *write*, *compareTo*
 - Computationally efficient, but slow for rapid prototyping

A Few Gotchas

- Avoid object creation, at all costs
- Execution framework reuses value in reducer
- Passing parameters into mappers and reducers
  - DistributedCache for larger (static) data

Basic Cluster Components

- One of each:
  - Namenode (NN)
  - Jobtracker (JT)
- Set of each per slave machine:
  - Tasktracker (TT)
  - Datanode (DN)

# Putting Everything Together
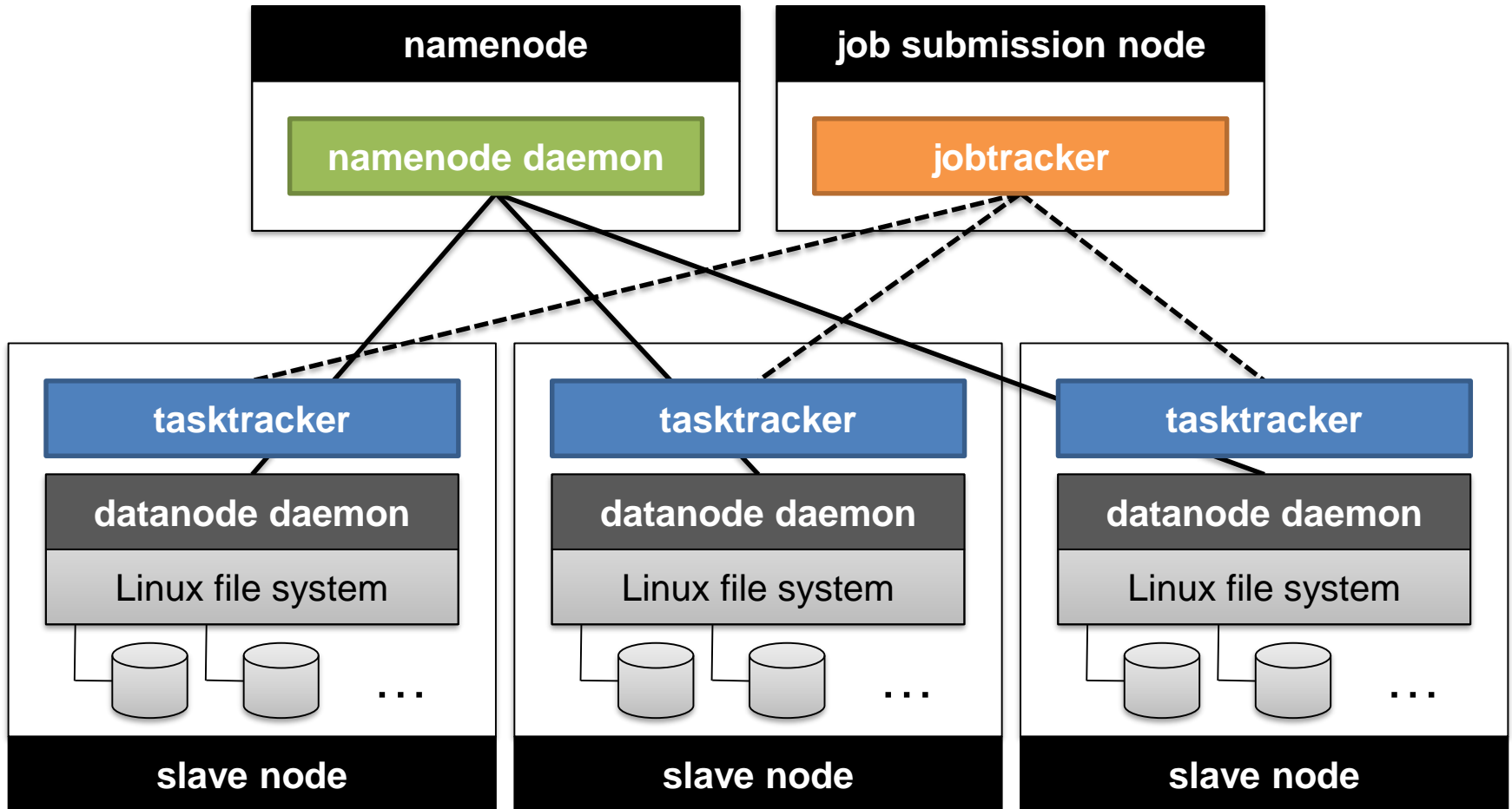
Anatomy of a Job

- MapReduce program in Hadoop = Hadoop job
  - Jobs are divided into map and reduce tasks
  - An instance of running a task is called a task attempt
 - Multiple jobs can be composed into a workflow

Anatomy of a Job

- Job submission process
  - Client (i.e., driver program) creates a job, configures it, and submits it to job tracker
  - JobClient computes input splits (on client end)
  - Job data (jar, configuration XML) are sent to JobTracker
  - JobTracker puts job data in shared location, enqueues tasks
  - TaskTrackers poll for tasks
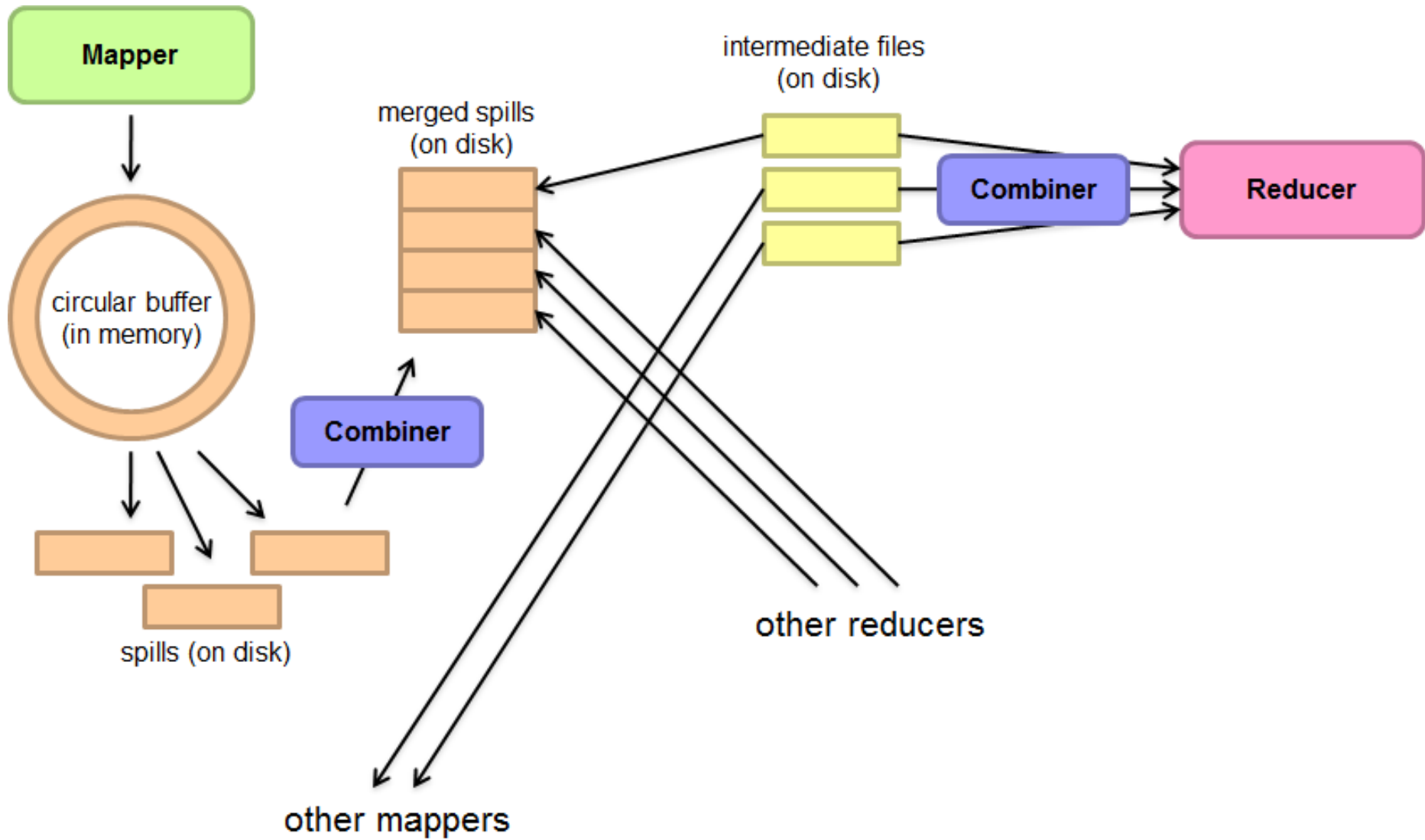 - Off to the races…
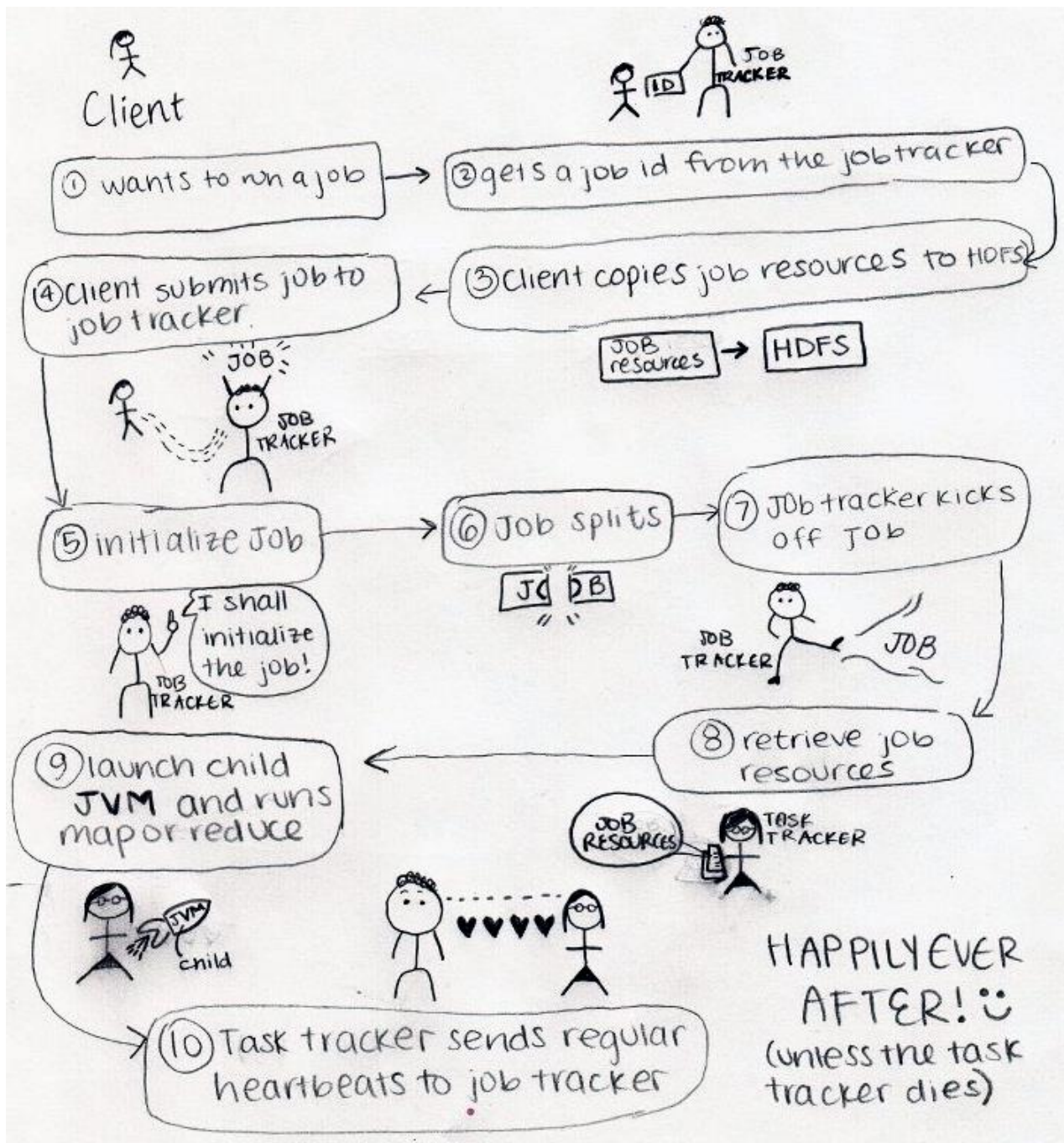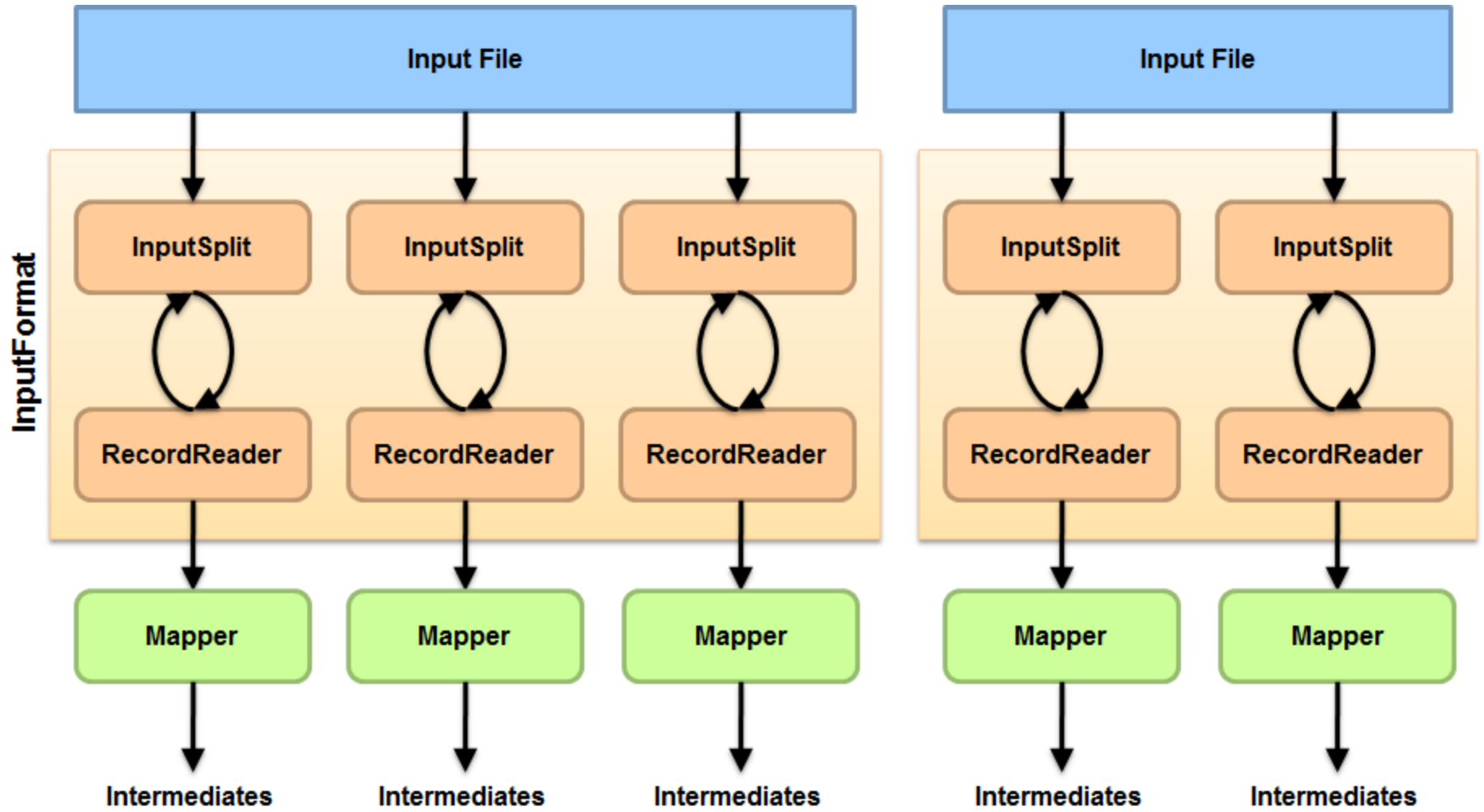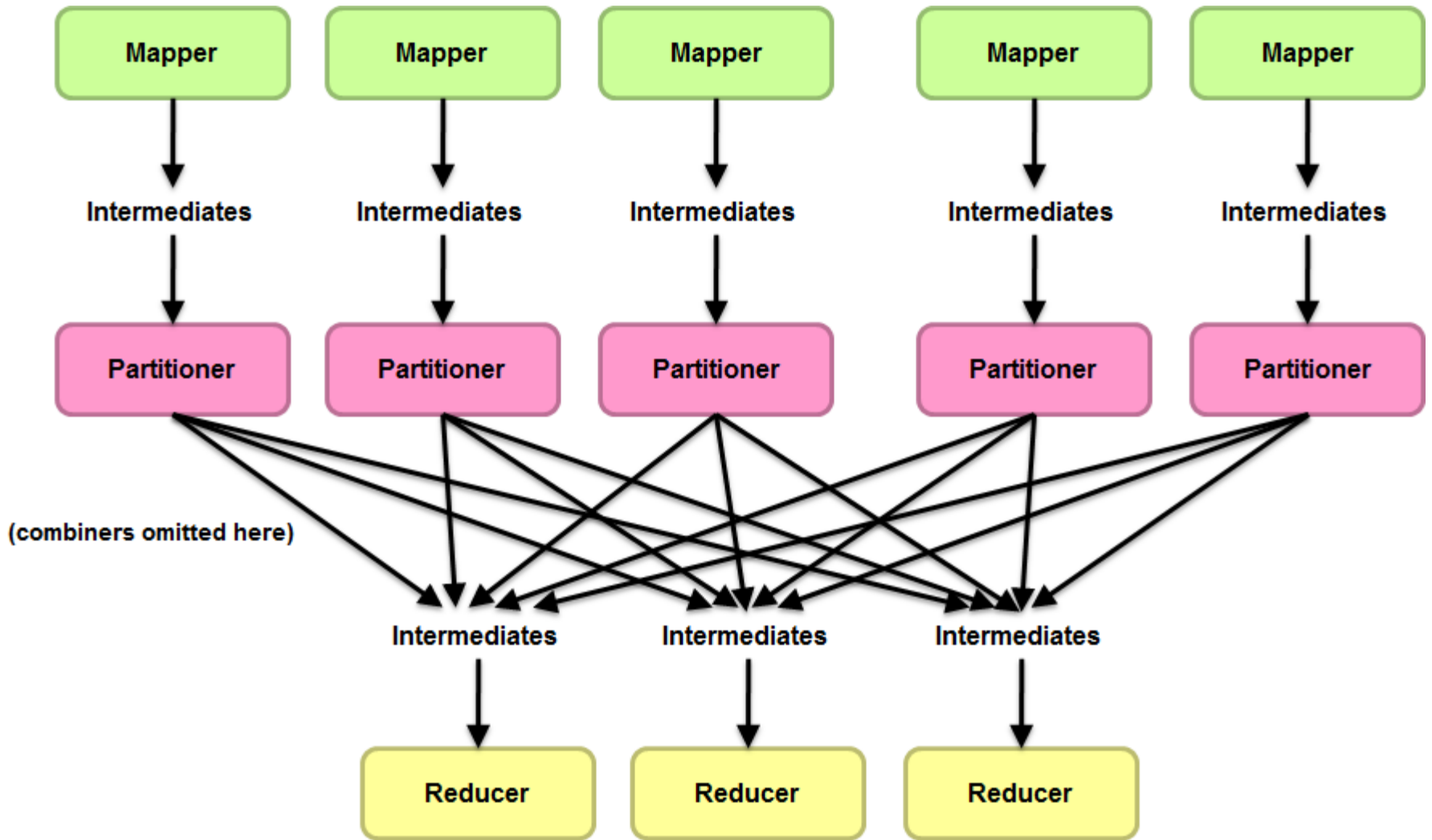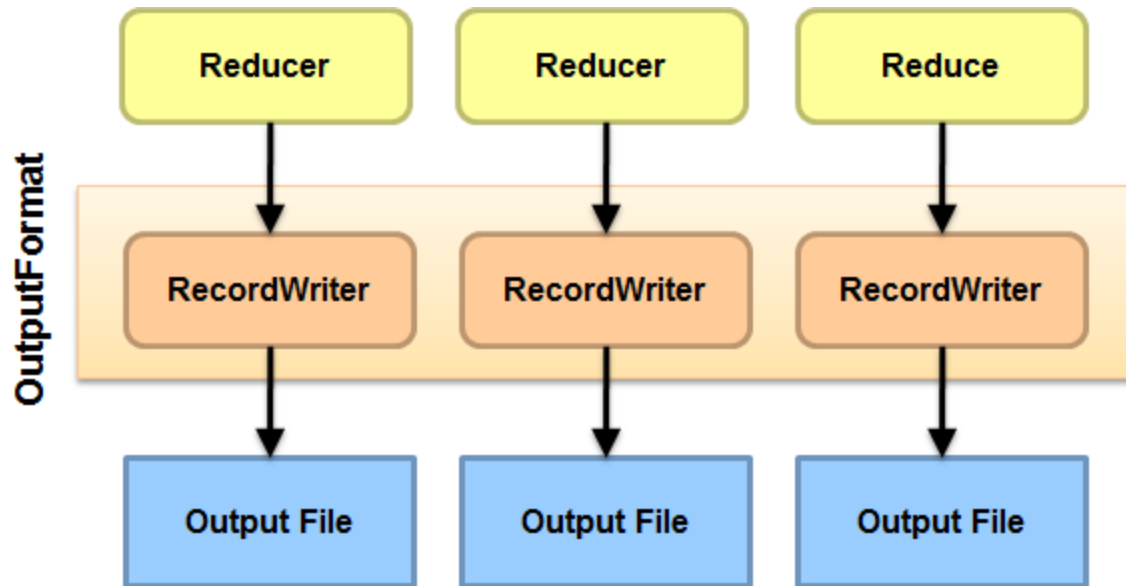
Input and Output

- InputFormat:
  - TextInputFormat
  - KeyValueTextInputFormat
  - SequenceFileInputFormat
  - ...
- OutputFormat:
  - TextOutputFormat
  - SequenceFileOutputFormat
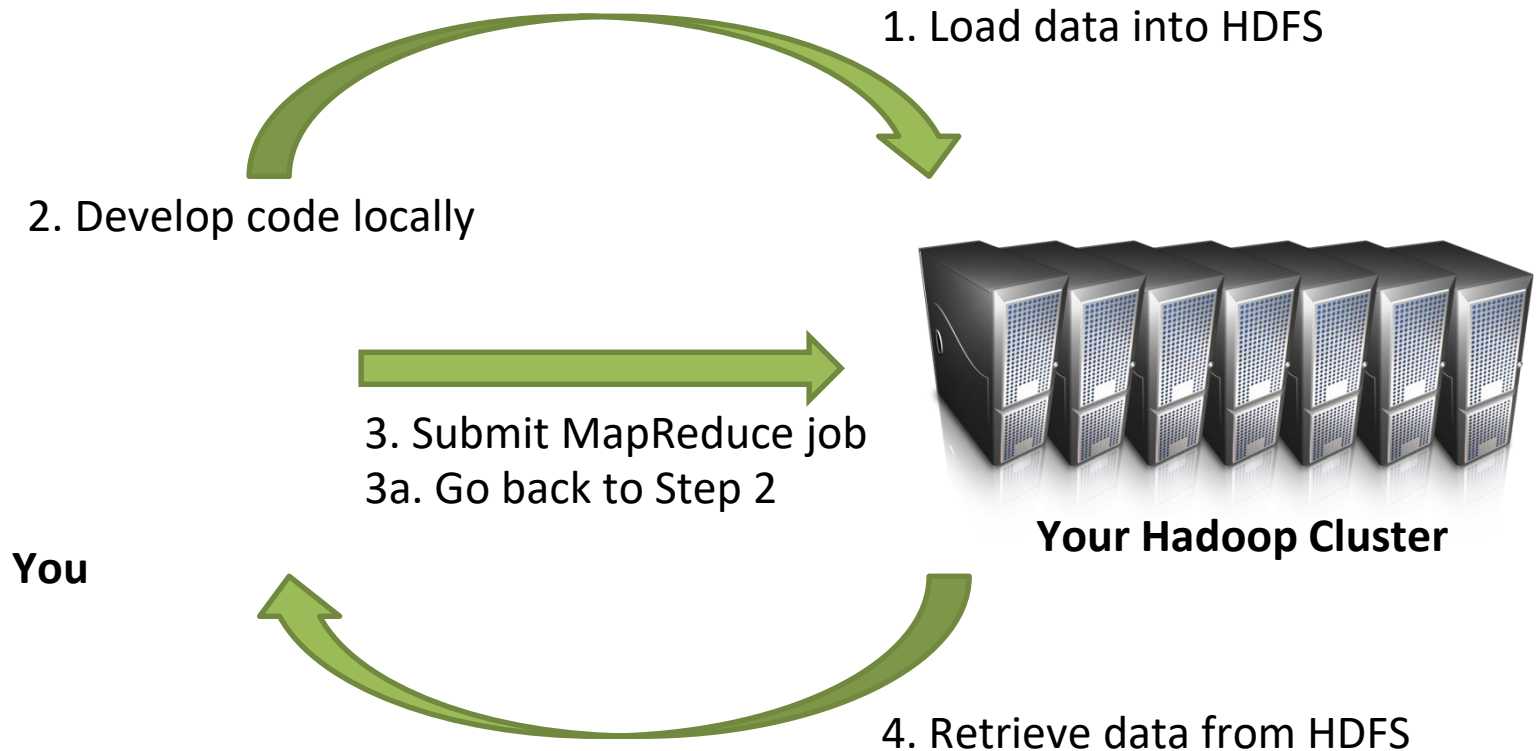 - ...

Shuffle and Sort in Hadoop

- Map side
  - Map outputs are buffered in memory in a circular buffer
  - When buffer reaches threshold, contents are "spilled" to disk
  - Spills merged in a single, partitioned file (sorted within each partition): combiner runs here

Shuffle and Sort in Hadoop

- Reduce side
  - First, map outputs are copied over to reducer machine
  - "Sort" is a multi-pass merge of map outputs (happens in memory and on disk): combiner runs here
  - Final merge pass goes directly into reducer

Mapper

circular buffer
(in memory)

spills (on disk)

Combiner

merged spills
(on disk)

intermediate files
(on disk)

Combiner

Reducer

other reducers

other mappers

Hadoop Workflow

1. Load data into HDFS

2. Develop code locally

3. Submit MapReduce job
3a. Go back to Step 2

You

Your Hadoop Cluster

4. Retrieve data from HDFS

On AWS: with EC2

0. Allocate Hadoop cluster
1. Load data into HDFS

EC2

2. Develop code locally

3. Submit MapReduce job
3a. Go back to Step 2

You

Your Hadoop Cluster

4. Retrieve data from HDFS
5. Clean up!

Uh oh. Where did the data go?

On AWS: with EC2 and S3

Copy from S3 to HDFS

EC2
(Cloud Computer)

S3
(Persistent Store)

Your Hadoop Cluster

Copy from HFDS to S3
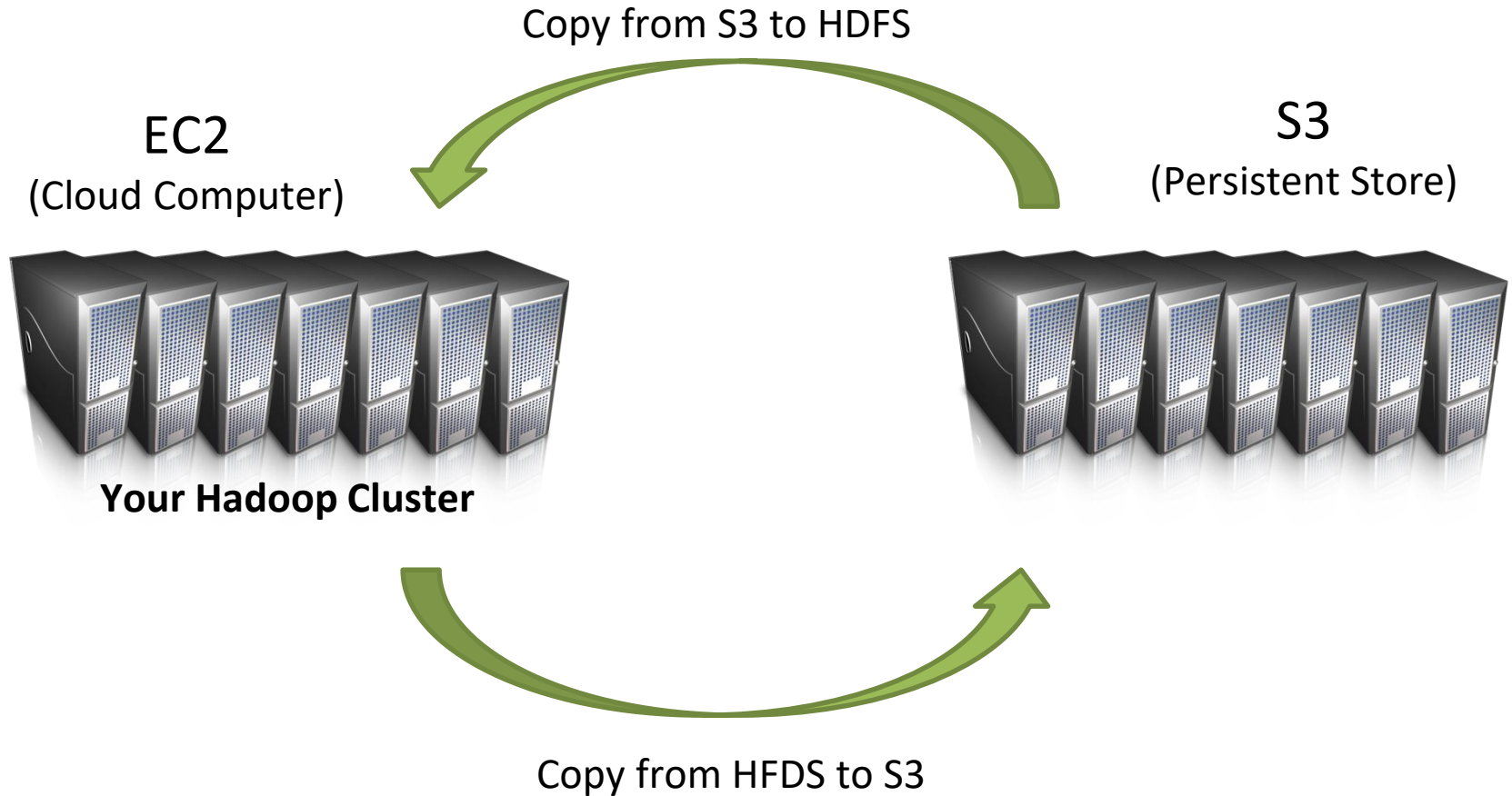
Testing and Executing a Hadoop job

- Required environment:
  - JDK on client
  - JRE on all Hadoop nodes
  - Hadoop distribution (HDFS + MapReduce) on client and all Hadoop nodes
  - SSH servers on each tasktracker, SSH client on jobtracker (used to control the execution of tasktrackers)
  - An IDE (e.g., Eclipse + plugin) on client

Testing and Executing a Hadoop job

- Three different execution modes:
  - **local**: one mapper, one reducer, run locally from the same JVM as the client
  - **pseudo-distributed**: mappers and reducers are launched on a single machine, but communicate over the network
  - **distributed**: over a cluster for real runs

Testing and Executing a Hadoop job

- Task JVM Reuse
  - By default, each map and reduce task (of a given split) is run in a separate JVM
  - When there is a lot of initialization to be done, or when splits are small, it might be useful to reuse JVMs for subsequent tasks
  - Of course, only works for tasks run on the same node

Testing and Executing a Hadoop job

- Hadoop in the cloud
  - Possibly to set up one's own Hadoop cluster
  - But often easier to use Hadoop clusters in the cloud that support MapReduce:
    - Amazon EMR etc.
  - Not always easy to know the cluster's configuration (in terms of racks, etc.) when on the cloud, which hurts data locality in MapReduce

Debugging Hadoop Programs

- First, take a deep breath
- Start small, start locally
  - Debugging in local mode is the easiest
  - Remote debugging possible but complicated to set up (impossible to know in advance where a map or reduce task will be executed)
- Learn to use the Web interface with status information about the job
- Throw RuntimeExceptions

Debugging Hadoop Programs

- Where does println go?
  - Standard output and error channels saved on each node, accessible through the Web interface
- Don't use println, use logging
- Counters can be used to track side information across a MapReduce job
  - e.g., the number of invalid input records
- IsolationRunner allows to run in isolation part of the MapReduce job

Debugging at Scale

- Your program works on small datasets, but it won't scale... Why?
  - Memory management issues (object creation and buffering)
  - Too much intermediate data
  - Mangled input records

Debugging at Scale

- Real-world data is messy!
  - For example, WordCount
    - How many unique words there are in Wikipedia?
  - There's no such thing as "consistent data"
  - Watch out for corner cases
  - Isolate unexpected behavior, bring local

Hadoop Zen

- This is bleeding edge technology (i.e., immature!)
  - Bugs, undocumented features, inexplicable behavior
  - Data loss(!)
- Don't get frustrated (take a deep breath)...
  - Those W$*#T@F! moments
- RTFM, RTFC

Hadoop Zen

- Be patient …
  - We will inevitably encounter "situations" along the way
- Be flexible …
  - We will have to be creative in workarounds
- Be constructive …
  - Tell me how I can make everyone's experience better

Help Us to Help You

- If your code failed to run on AWS EMR and you would like to contact the TA, please tell us
  - whether the *dummy* mapper/reducer that does nothing can complete successfully
  - whether the *identity* mapper/reducer that just reads each line of stdin and then writes it to stdout can complete successfully, and if so what the output files look like
  - what the error messages in the log files (especially sys.log) say

Take Home Messages

- MapReduce Basics
- Hadoop Basics