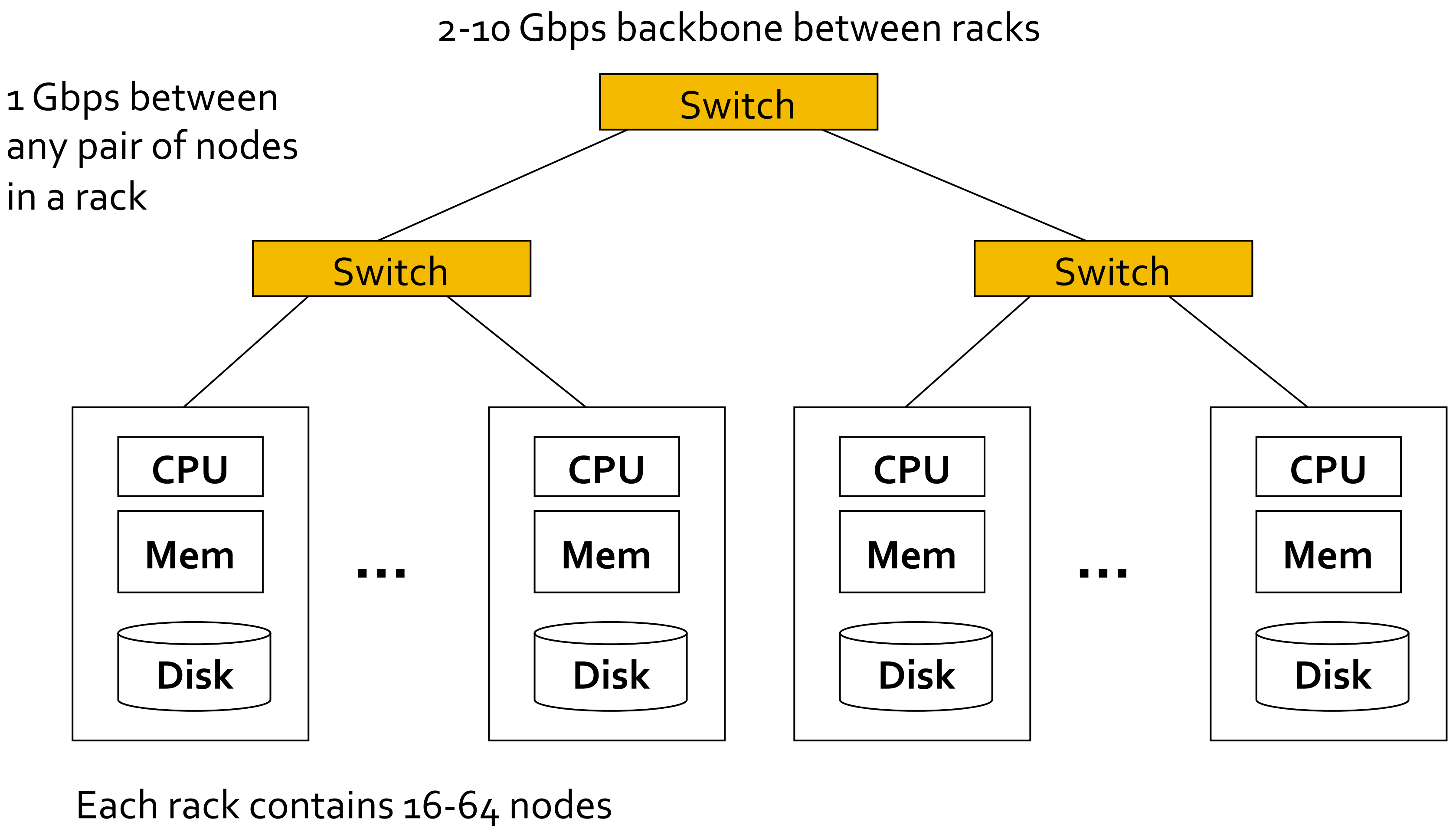
Map Reduce

Slides based on Lectures by A. Haeberlen, Z. Ives , J. Lin,
and other sources.

Motivation: Google Example

- 20+ billion web pages x 20KB = 400+ TB
- 1 computer reads 30-35 MB/sec from disk
  - ~4 months to read the web
- ~1,000 hard drives to store the web
- Takes even more to **do** something useful with the data!
- **Today, a standard architecture for such problems is emerging:**
  - Cluster of commodity Linux nodes
  - Commodity network (ethernet) to connect them

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

In 2011 it was guestimated that Google had 1M machines, http://bit.ly/Shh0RO

Large-scale Computing

- **Large-scale computing** for **data mining** problems on **commodity hardware**
- **Challenges:**
  - **How do you distribute computation?**
  - **How can we make it easy to write distributed programs?**
  - **Machines fail:**
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to loose 1/day
    - People estimated Google had ~1M machines in 2011
      - 1,000 machines fail every day!

Idea and Solution

- **Issue: Copying data over a network takes time**
- **Idea:**
  - Bring computation close to the data
  - Store files multiple times for reliability
- **Map-reduce** addresses these problems
  - Google’s computational/data manipulation model
  - Elegant way to work with big data
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS
  - **Programming model**
    - Map-Reduce

Storage Infrastructure

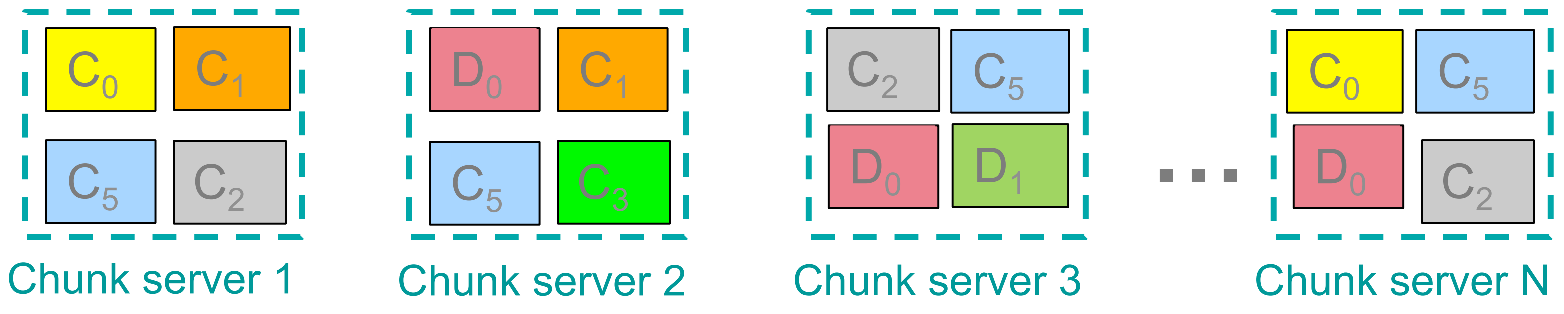
- **Problem:**
  - If nodes fail, how to store data persistently?
- **Answer:**
  - **Distributed File System:**
    - Provides global file namespace
    - Google GFS; Hadoop HDFS;
- **Typical usage pattern**
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends 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 Hadoop's HDFS
  - Stores metadata about where files are stored
  - Might be replicated
- **Client library for file access**
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

Distributed File System

- **Reliable distributed file system**
- Data kept in “chunks” spread across machines
- Each chunk replicated on different machines
  - Seamless recovery from disk or machine failure



Bring computation directly to the data!

Chunk servers also serve as compute servers

What is MapReduce?

- A famous distributed programming model
- In many circles, considered *the* key building block for much of Google's data analysis
  - A programming language built on it: Sawzall, http://labs.google.com/papers/sawzall.html
  - *… Sawzall has become one of the most widely used programming languages at Google. … [O]n one dedicated Workqueue cluster with 1500 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each. While running those jobs, 18,636 failures occurred (application failure, network outage, system crash, etc.) that triggered rerunning some portion of the job. The jobs read a total of 3.2x10^{15} bytes of data (2.8PB) and wrote 9.9x10^{12} bytes (9.3TB).*
  - Other similar languages: Yahoo's Pig Latin and Pig; Microsoft's Dryad
- Cloned in open source: Hadoop, http://hadoop.apache.org/

The MapReduce programming model

- Simple distributed functional programming primitives
- Modeled after Lisp primitives:
  - `map` (apply function to all items in a collection) and
  - `reduce` (apply function to set of items with a common key)
- We start with:
  - A user-defined function to be applied to all data,
    `map`: (key,value) → (key, value)
  - Another user-specified operation
    `reduce`: (key, {set of values}) → result
  - A set of *n* nodes, each with data
- All nodes run `map` on all of their data, producing new data with keys
  - This data is collected by key, then shuffled, and finally `reduced`
  - Dataflow is through temp files on GFS

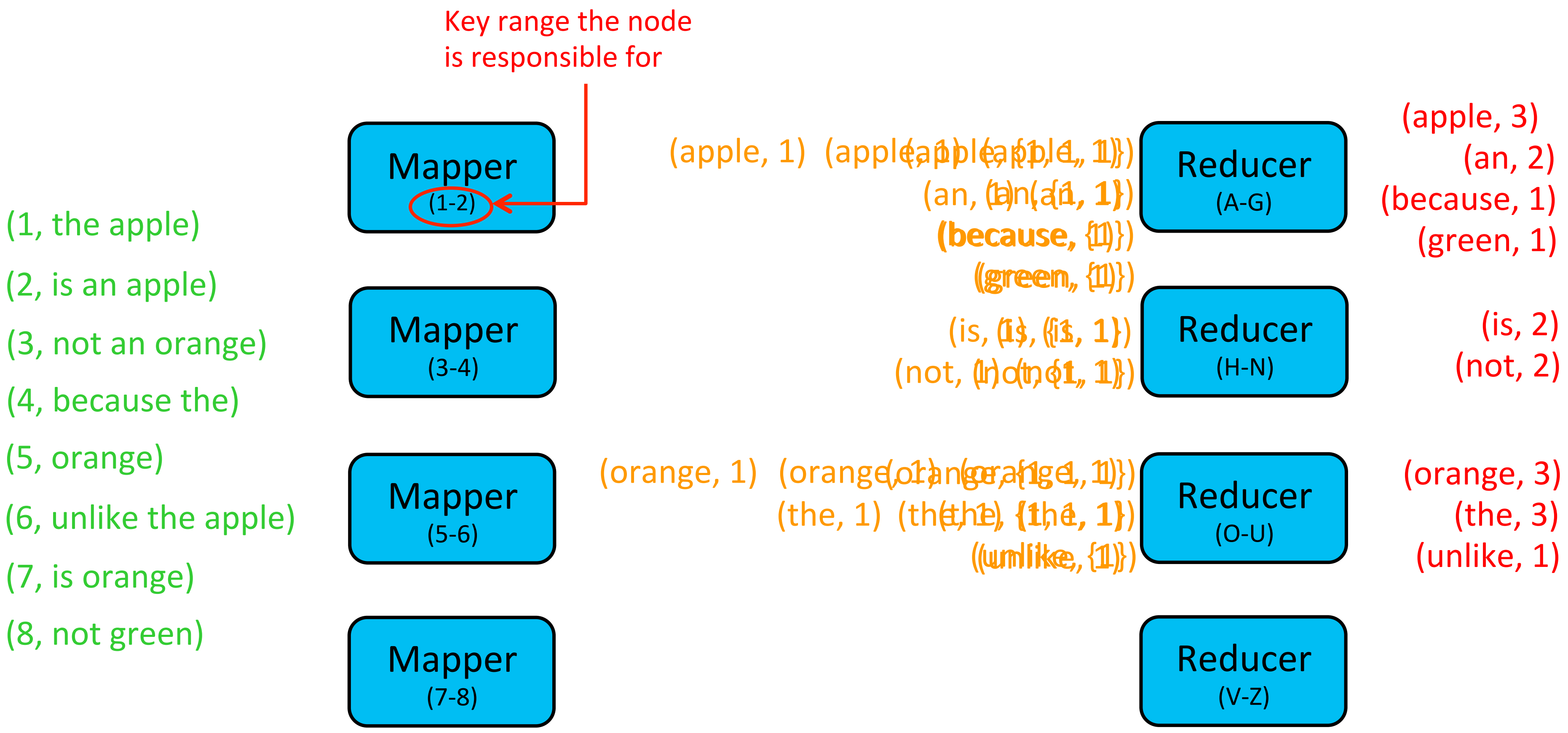
Simple example: Word count

```
map(String key, String value) {
  // key: document name, line no
  // value: contents of line
  for each word w in value:
    emit(w, "1")
}
```

```
reduce(String key, Iterator values) {
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  emit(key, result)
}
```

- Goal: Given a set of documents, count how often each word occurs
  - Input: Key-value pairs (document:lineNumber, text)
  - Output: Key-value pairs (word, #occurrences)
  - What should be the intermediate key-value pairs?

Simple example: Word count



1 Each mapper receives some of the KV-pairs as input

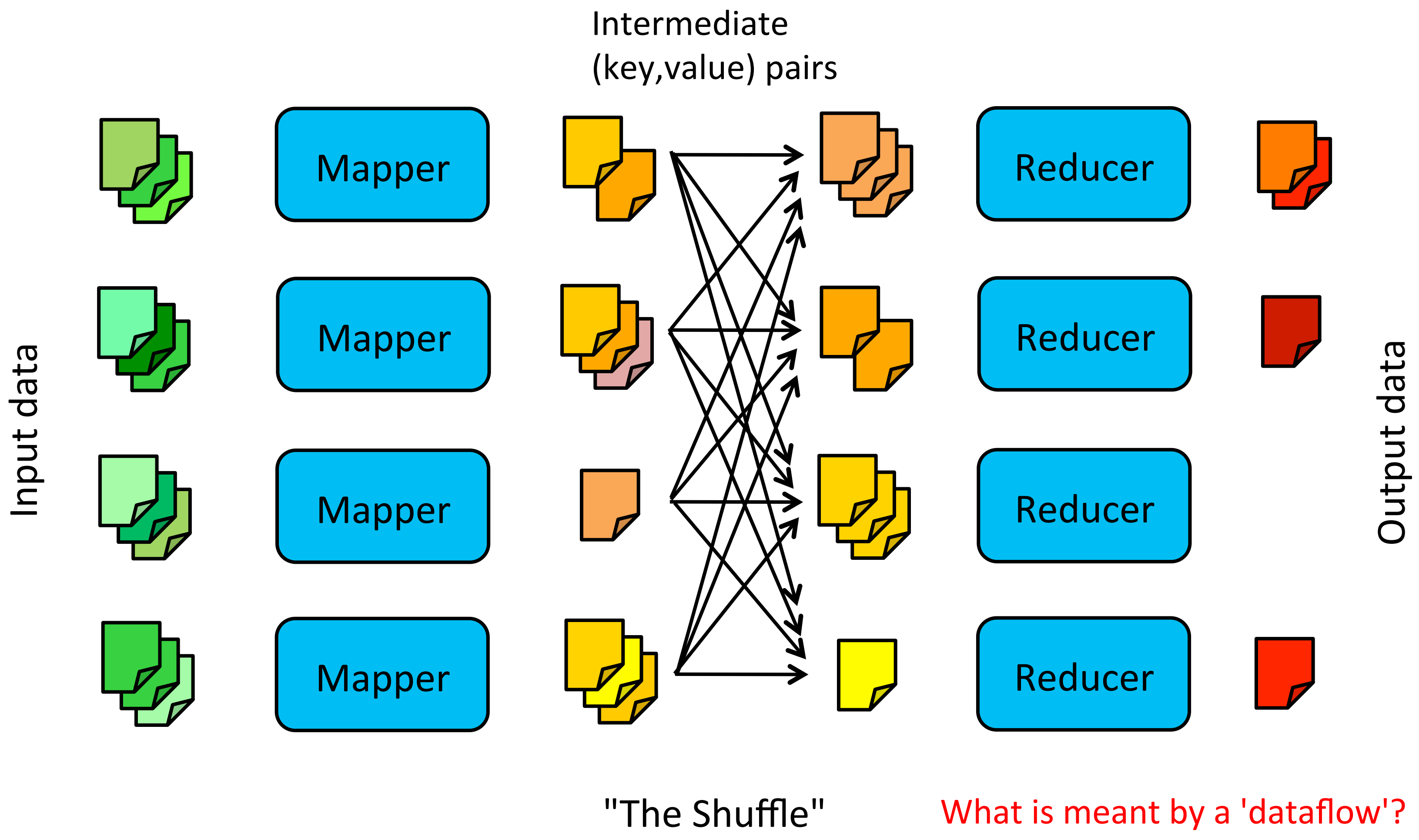
2 The mappers process the KV-pairs one by one

3 Each KV-pair output by the mapper is sent to the reducer that is responsible for it

4 The reducers sort their input by key and group it

5 The reducers process their input one group at a time

MapReduce dataflow



What is meant by a 'dataflow'?
What makes this so scalable?

More examples

- Distributed grep – all lines matching a pattern
  - Map: filter by pattern
  - Reduce: output set
- Count URL access frequency
  - Map: output each URL as key, with count 1
  - Reduce: sum the counts
- Reverse web-link graph
  - Map: output (target,source) pairs when link to target found in souce
  - Reduce: concatenates values and emits (target,list(source))
- Inverted index
  - Map: Emits (word,documentID)
  - Reduce: Combines these into (word,list(documentID))

Common mistakes to avoid

- Mapper and reducer should be stateless
  - Don't use static variables - after `map` + `reduce` return, they should remember nothing about the processed data!
  - Reason: No guarantees about which key-value pairs will be processed by which workers!

```
HashMap h = new HashMap();
map(key, value) {
  if (h.contains(key)) {
    h.add(key,value);
    emit(key, "X");
  }
}
```

Wrong!

- Don't try to do your own I/O!
  - Don't try to read from, or write to, files in the file system
  - The MapReduce framework does all the I/O for you:
    - All the incoming data will be fed as arguments to map and reduce
    - Any data your functions produce should be output via emit

```
map(key, value) {
  File foo =
    new File("xyz.txt");
  while (true) {
    s = foo.readLine();
    ...
  }
}
```

Wrong!

More common mistakes to avoid

```
map(key, value) {
  emit("FOO", key + " " + value);
}
```

Wrong!

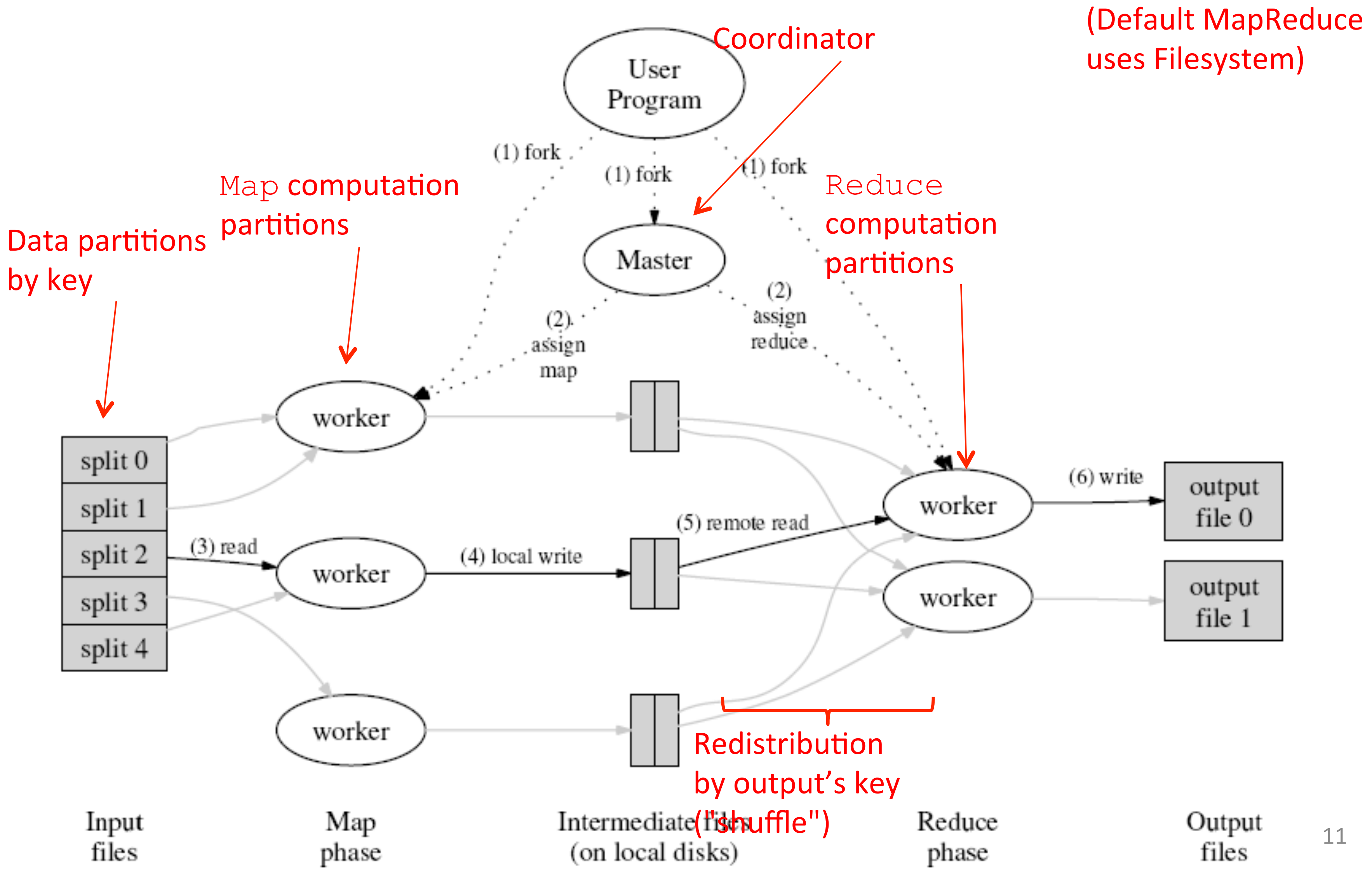
```
reduce(key, value[]) {
  /* do some computation on
  all the values */
}
```

- Mapper must not map too much data to the same key
  - In particular, don't map *everything* to the same key!!
  - Otherwise the reduce worker will be overwhelmed!
  - It's okay if some reduce workers have more work than others
    - Example: In WordCount, the reduce worker that works on the key 'and' has a lot more work than the reduce worker that works on 'syzygy'.

Designing MapReduce algorithms

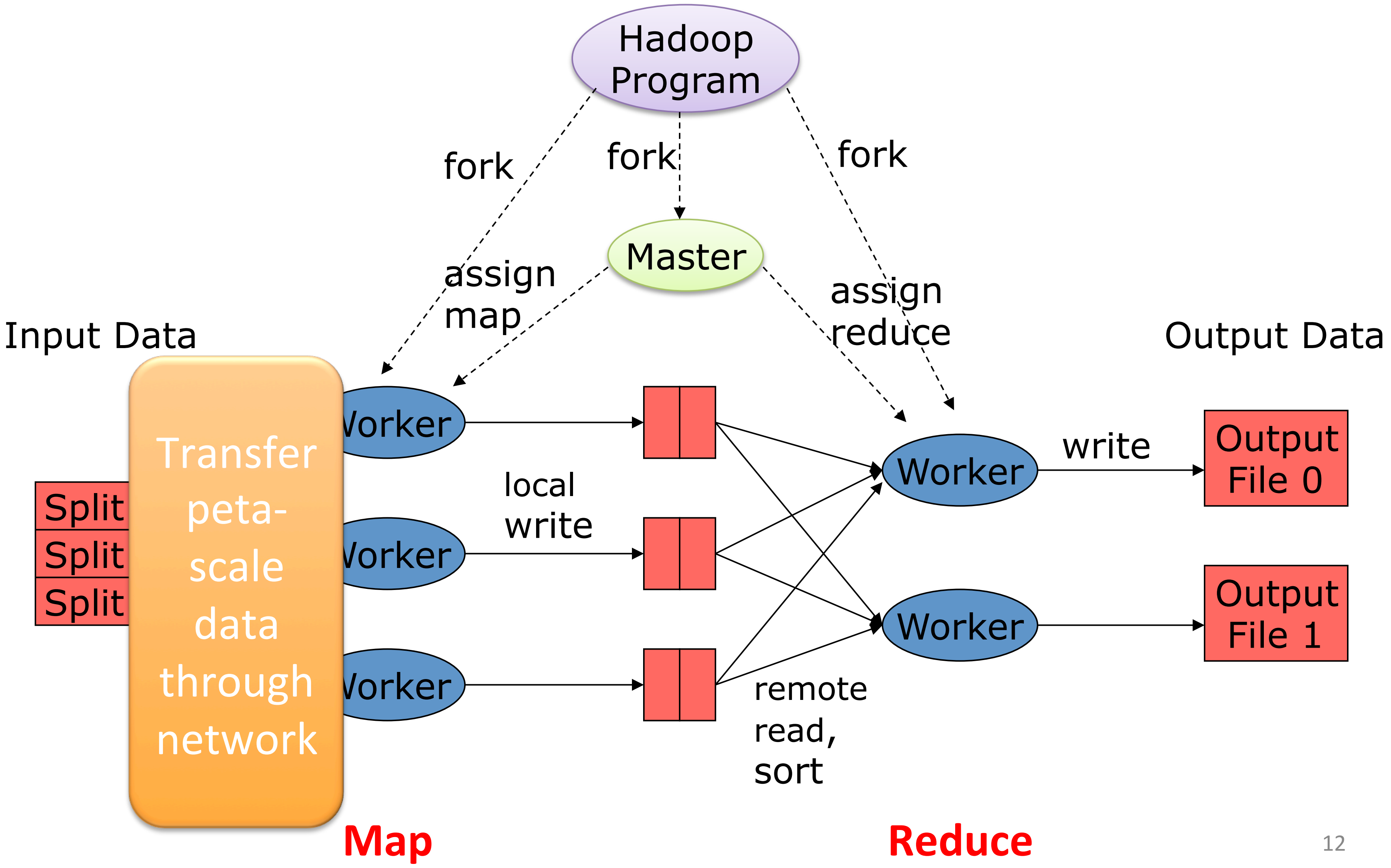
- Key decision: What should be done by `map`, and what by `reduce`?
  - `map` can do something to each individual key-value pair, but it can't look at other key-value pairs
    - Example: Filtering out key-value pairs we don't need
  - `map` can emit more than one intermediate key-value pair for each incoming key-value pair
    - Example: Incoming data is text, `map` produces (word,1) for each word
  - `reduce` can aggregate data; it can look at multiple values, as long as `map` has mapped them to the same (intermediate) key
    - Example: Count the number of words, add up the total cost, ...
- Need to get the intermediate format right!
  - If `reduce` needs to look at several values together, `map` must emit them using the same key!

More details on the MapReduce data flow



http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

MapReduce


Hadoop Program
fork
fork
fork
Master
assign map
assign reduce
Input Data
Output Data
Split
Split
Split
Transfer peta-scale data through network
Worker
Worker
Worker
local write
Worker
Worker
write
Output File 0
Output File 1
remote read, sort
Map
Reduce

What if a worker crashes?

- We rely on the file system being shared across all the nodes
- Two types of (crash) faults:
  - Node wrote its output and then crashed
    - Here, the file system is likely to have a copy of the complete output
  - Node crashed before finishing its output
    - The JobTracker sees that the job isn't making progress, and restarts the job elsewhere on the system
- (Of course, we have fewer nodes to do work…)
- But what if the master crashes?

Scale and MapReduce

- From a particular Google paper on a language built over MapReduce:
  - … Sawzall has become one of the most widely used programming languages at Google. …
    [O]n one dedicated Workqueue cluster with 1500 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each. While running those jobs, 18,636 failures occurred (application failure, network outage, system crash, etc.) that triggered rerunning some portion of the job. The jobs read a total of $3.2\text{x}10^{15}$ bytes of data (2.8PB) and wrote $9.9\text{x}10^{12}$ bytes (9.3TB).

Source: Interpreting the Data: Parallel Analysis with Sawzall (Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan)

Hadoop and Python

- Hadoop is an open source implementation of MapReduce
  - it is also free!!!
- Part a an ecosystem that includes, the file system (HDFS), database systems on top, machine learning algorithms, etc

MapReduce/GFS Summary

- Simple, but powerful programming model
- Scales to handle petabyte+ workloads
  - Google: six hours and two minutes to sort 1PB (10 trillion 100-byte records) on 4,000 computers
  - Yahoo!: 16.25 hours to sort 1PB on 3,800 computers
- Incremental performance improvement with more nodes
- Seamlessly handles failures, but possibly with performance penalties