# Map Reduce

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

# 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
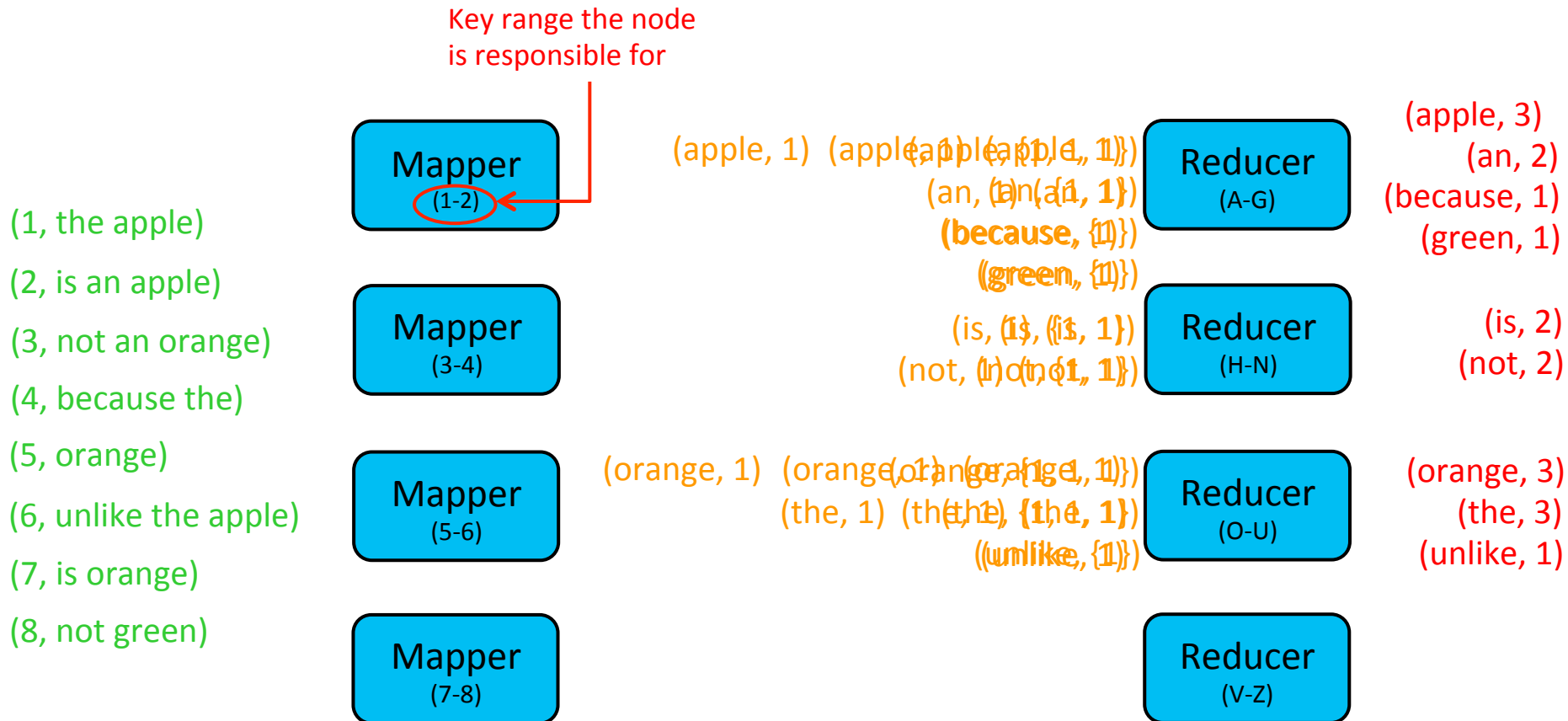
3

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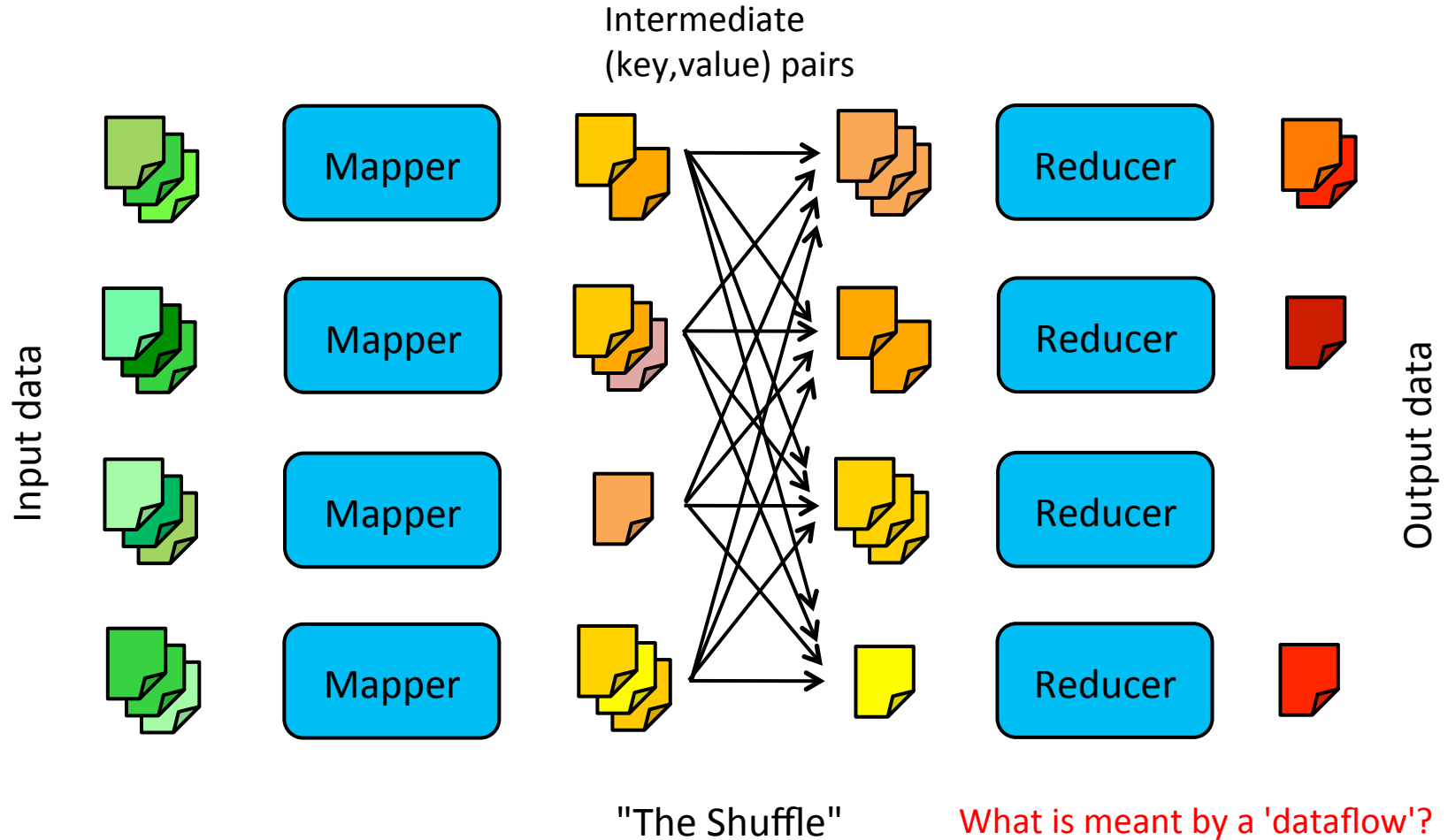
- Goal: Given a set of documents, count how often each word occurs
  – Input: Key-value pairs (document:lineNumber, text)
  – Output: Key-value pairs (word, #occurrences)
  – What should be the intermediate key-value pairs?

# Simple example: Word count

Key range the node
is responsible for

(1, the apple)

(2, is an apple)

(3, not an orange)

(4, because the)

(5, orange)

(6, unlike the apple)

(7, is orange)

(8, not green)

**Mapper**
(1-2)

**Mapper**
(3-4)

**Mapper**
(5-6)

**Mapper**
(7-8)

(apple, 1)  (apple, 1)  (apple, 1)  (apple, 1)
(an, 1)  (an, 1)  (an, 1)
(because, {1})
(green, {1})

(is, 1)  (is, 1)  (is, 1)
(not, 1)  (not, 1)  (not, 1)

(orange, 1)  (orange, 1)  (orange, 1)  (orange, 1)
(the, 1)  (the, 1)  (the, 1)  (the, 1)
(unlike, {1})

**Reducer**
(A-G)

**Reducer**
(H-N)

**Reducer**
(O-U)

**Reducer**
(V-Z)

(apple, 3)
(an, 2)
(because, 1)
(green, 1)

(is, 2)
(not, 2)

(orange, 3)
(the, 3)
(unlike, 1)

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

5

# MapReduce dataflow

Intermediate
(key,value) pairs

Input data

Mapper

Mapper

Mapper

Mapper

Reducer

Reducer
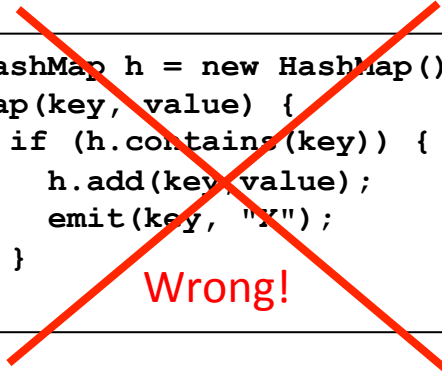
Reducer

Reducer

Output data

"The Shuffle"

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)

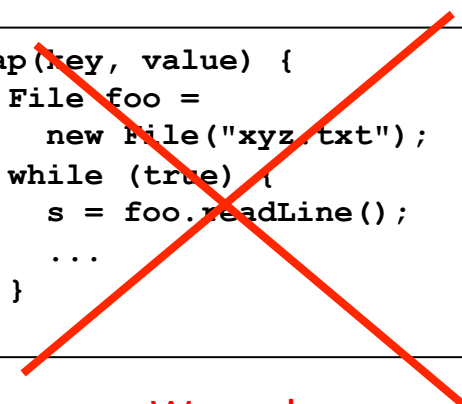  - Reduce: Combines these into (word,list(documentID))

# 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!

- 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

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

```
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);
}
```
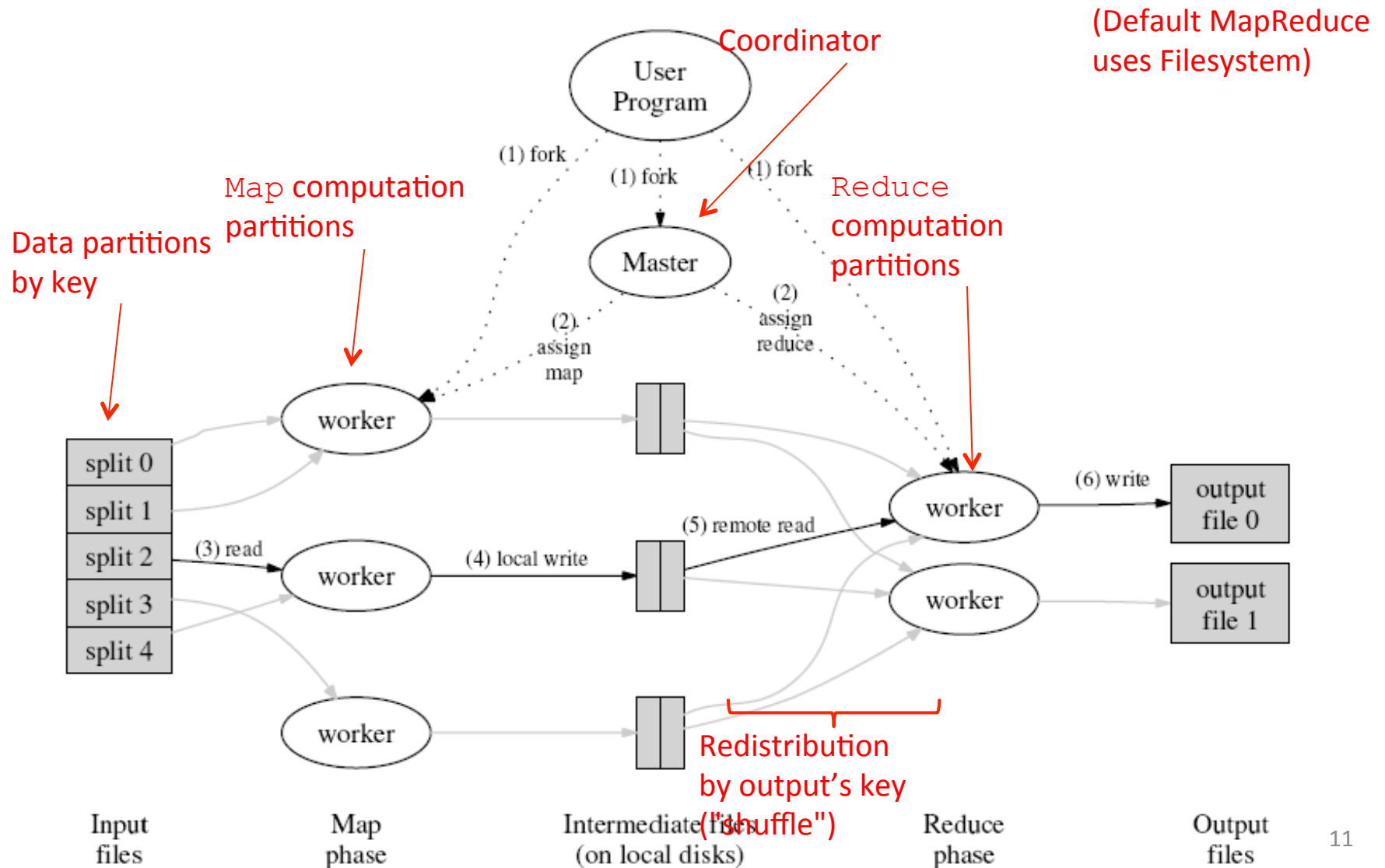
Wrong!

```
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'.
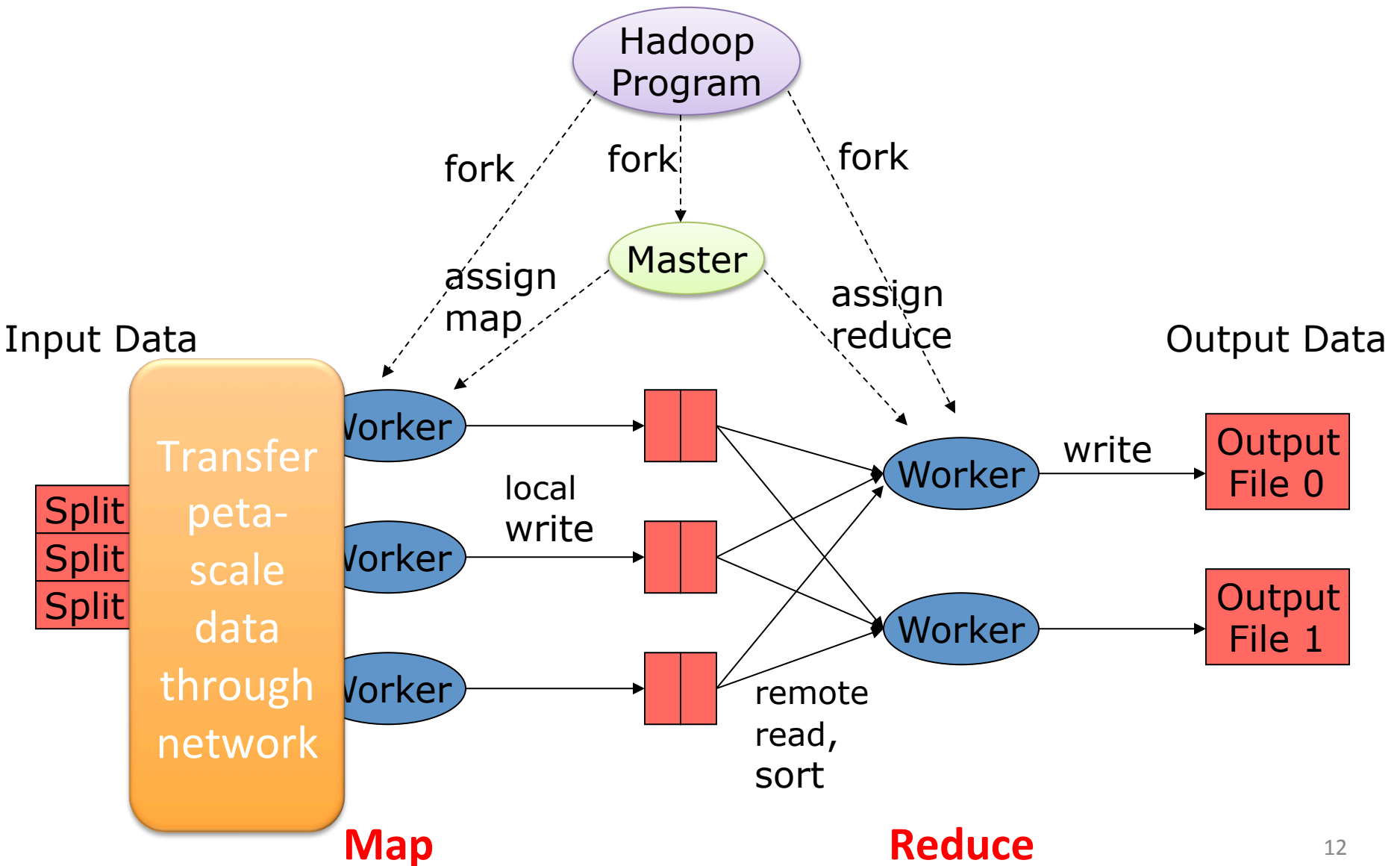
# Designing MapReduce algorithms

- 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!
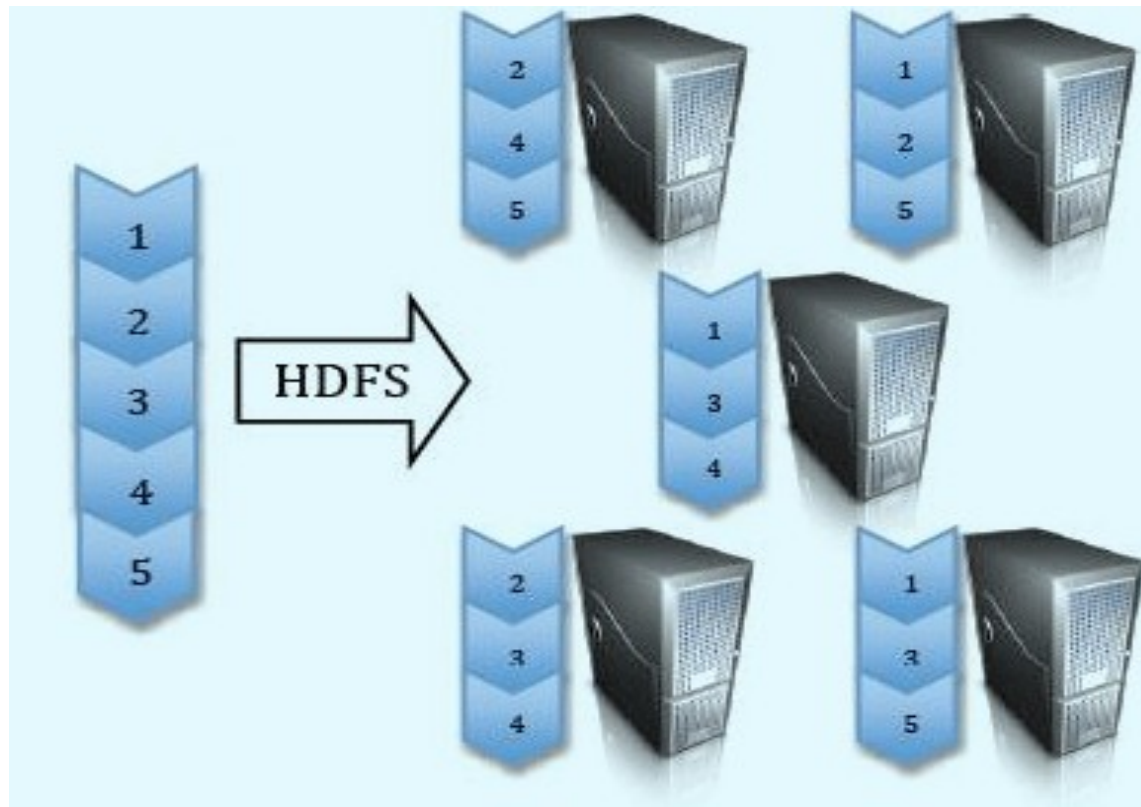
# More details on the MapReduce data flow



Coordinator

(Default MapReduce uses Filesystem)

Map computation partitions

Reduce computation partitions

Data partitions by key

Redistribution by output's key ("shuffle")

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

# MapReduce

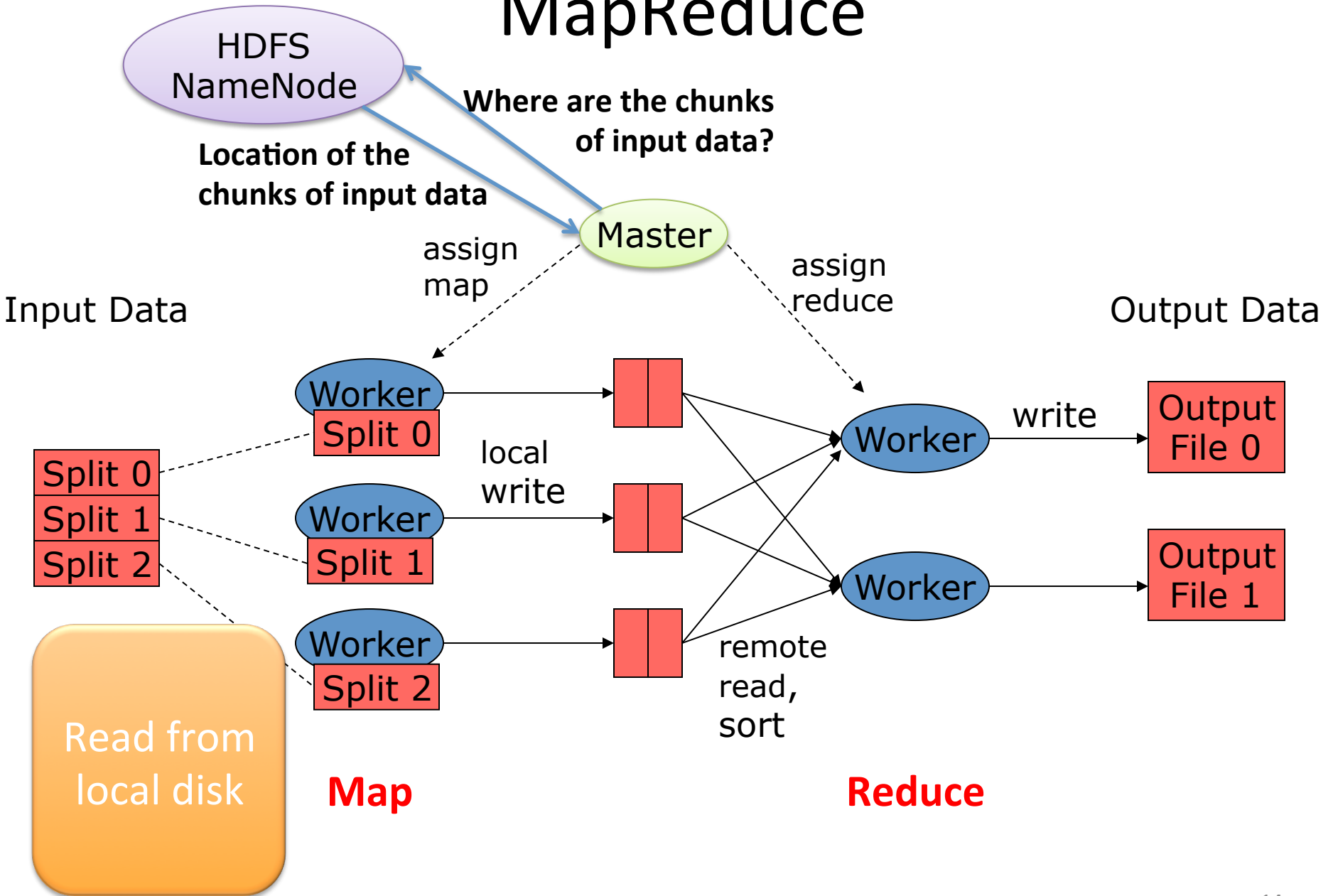# Google File System (GFS) Hadoop Distributed File System (HDFS)

- Split data and store 3 replica on commodity servers

# MapReduce

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

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  3  6  8

reduce      reduce      reduce

$r_1$ $s_1$        $r_2$ $s_2$        $r_3$ $s_3$
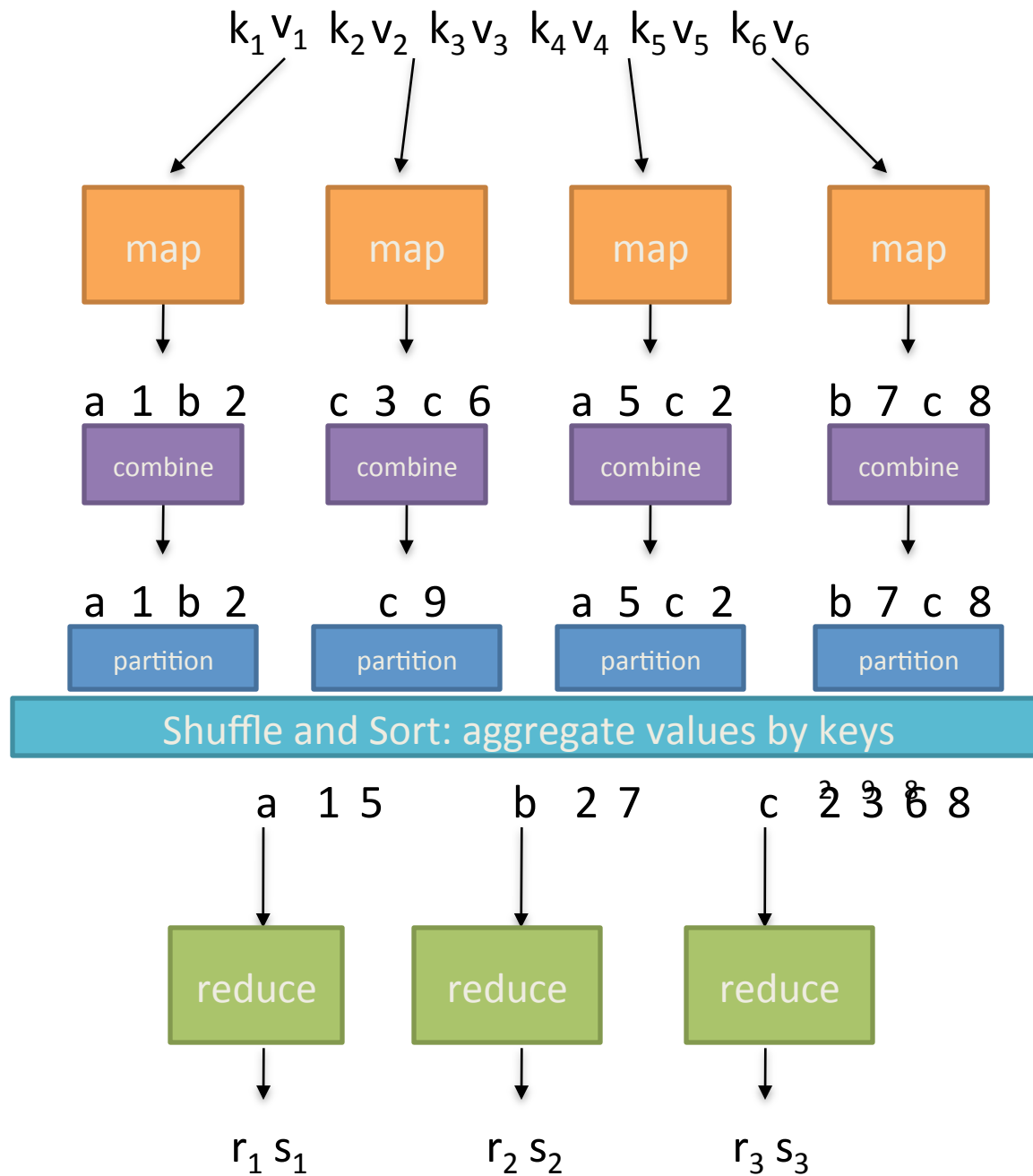
# What if a worker crashes?

- 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?

# Other challenges

- Locality
  - Try to schedule map task on machine that already has data
- Task granularity
  - How many map tasks? How many reduce tasks?
- Dealing with stragglers
  - Schedule some backup tasks
- Saving bandwidth
  - E.g., with combiners
- Handling bad records
  - "Last gasp" packet with current sequence number

# 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 $3.2 \times 10^{15}$ bytes of data (2.8PB) and wrote $9.9 \times 10^{12}$ 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

# Why a new file system?

- None designed for their failure model
- Few scale as highly or dynamically and easily
- Lack of special primitives for large distributed computation

# Components

- **Master (NameNode)**
  - Manages metadata (namespace)
  - Not involved in data transfer
  - Controls allocation, placement, replication

- **Chunkserver (DataNode)**
  - Stores chunks of data
  - No knowledge of GFS file system structure
  - Built on local linux file system
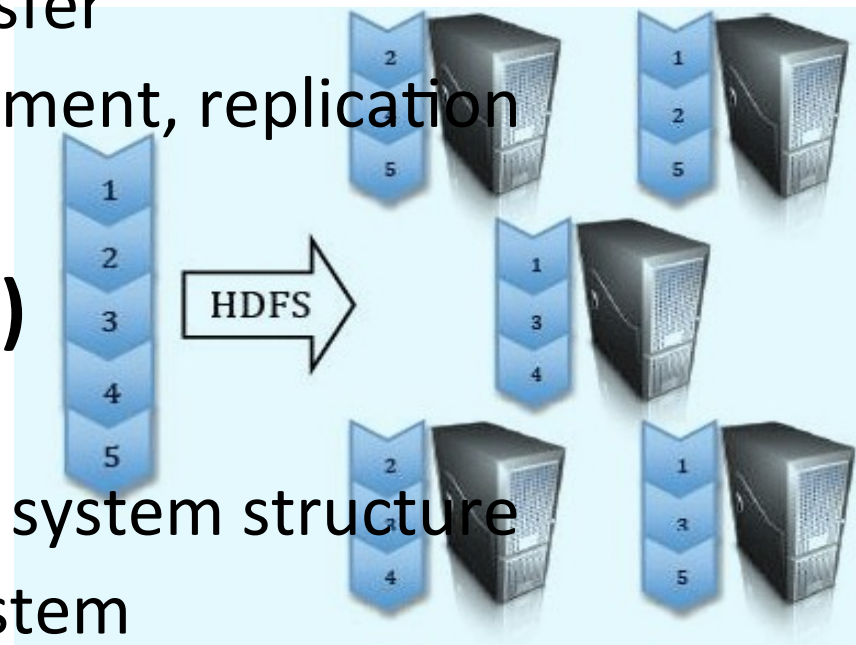
# MapReduce/GFS Summary

- 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