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 3.2×10^{15} bytes of data (2.8PB) and wrote 9.9×10^{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

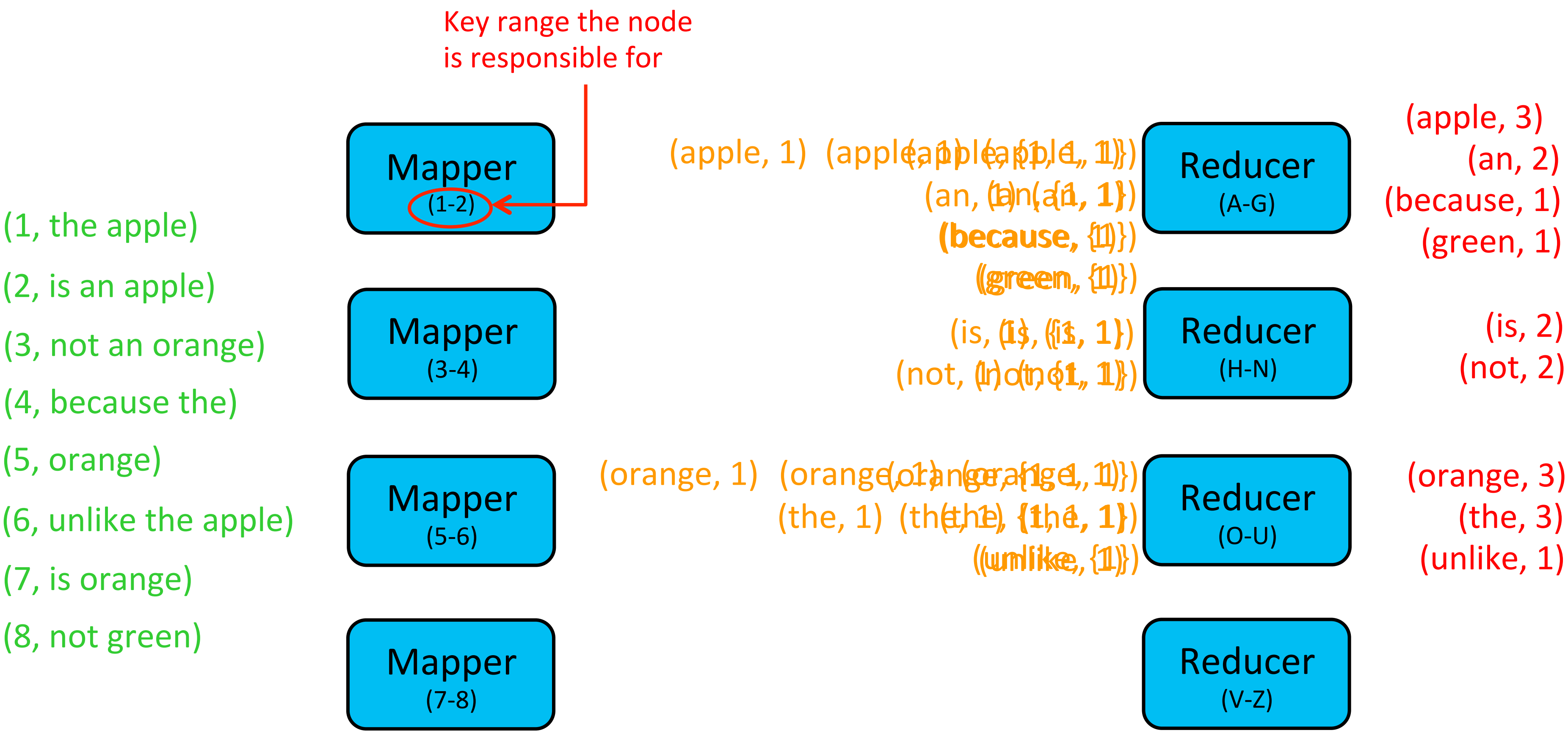
Simple example: Word count

```
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)
}
```

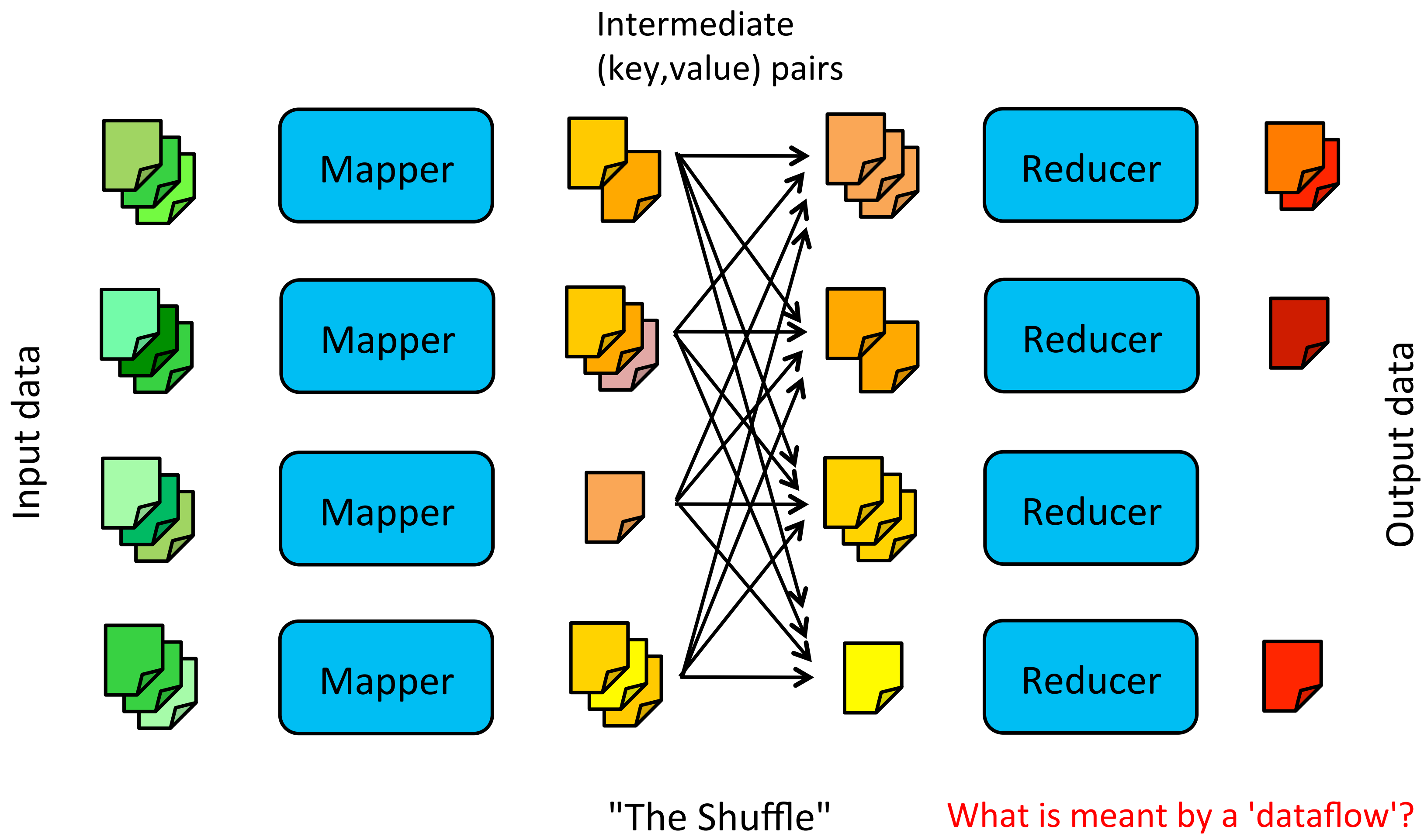
- Goal: Given a set of documents, count how often each word occurs
  - Input: Key-value pairs (document:lineNumber, text)
  - Output: Key-value pairs (word, #occurrences)
  - What should be the intermediate key-value pairs?

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!

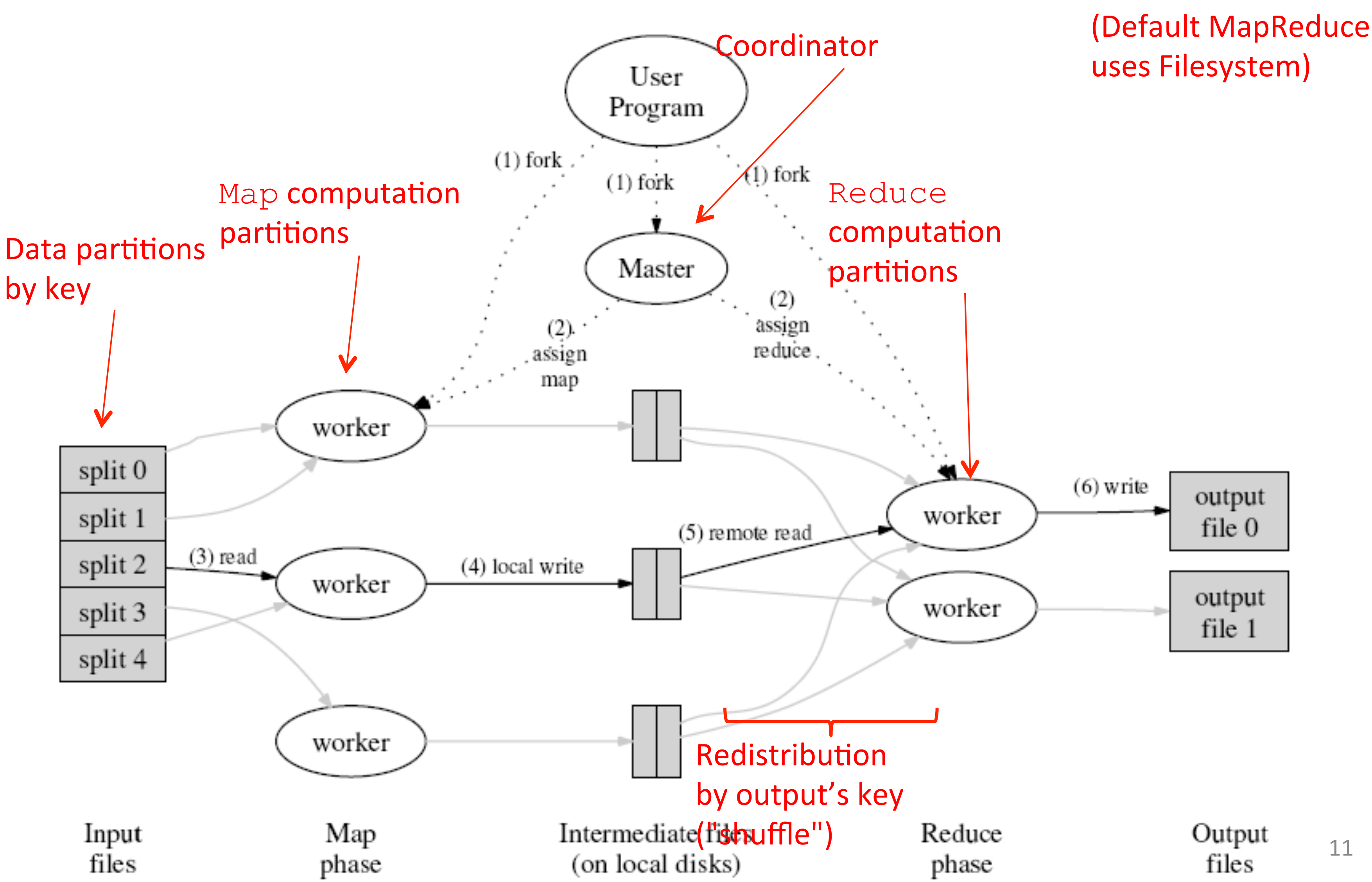
```
reduce(key, value[]) {
  /* do some computation on
  all the values */
}
```

- 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

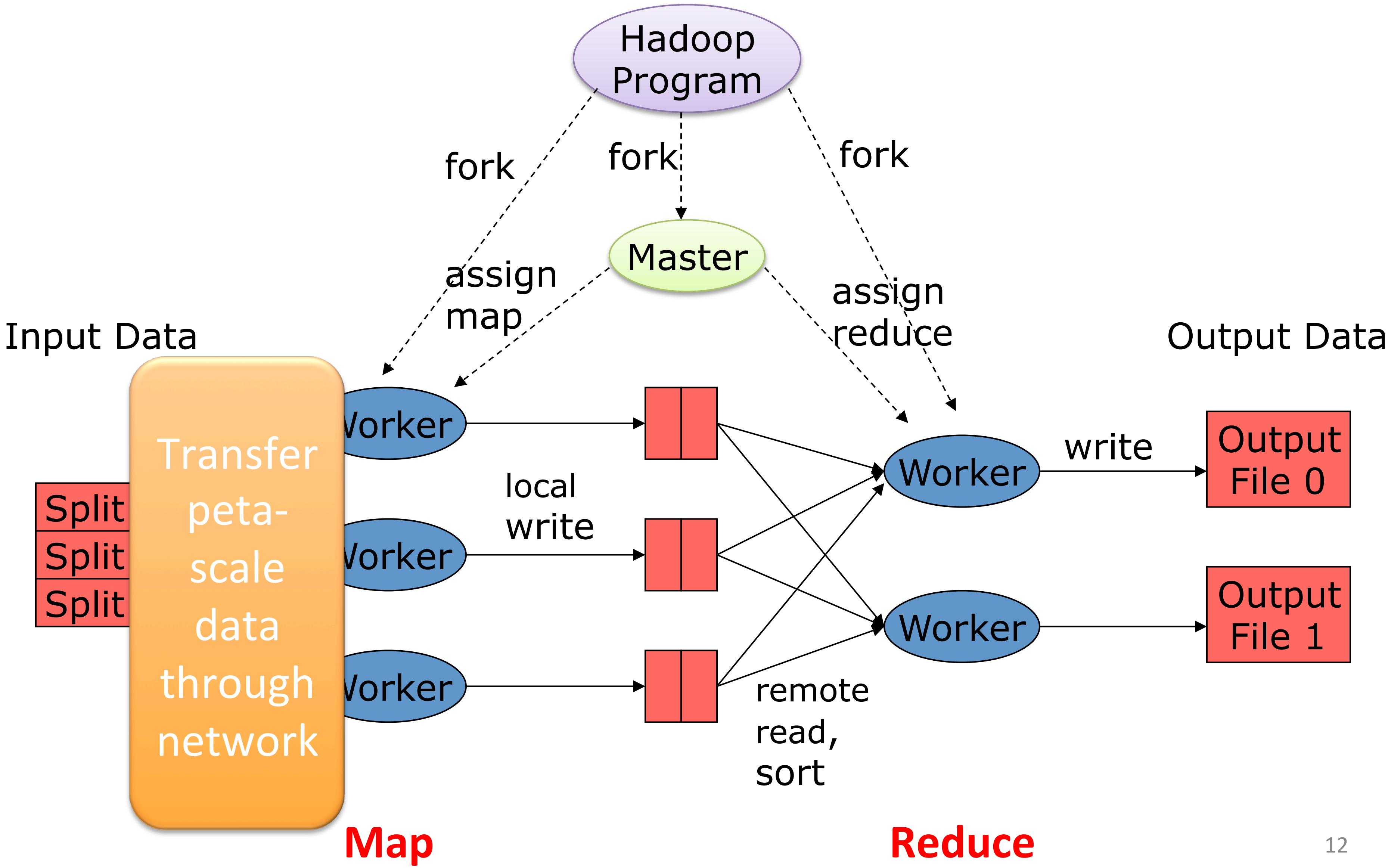
- Key decision: What should be done by `map`, and what by `reduce`?
  - `map` can do something to each individual key-value pair, but it can't look at other key-value pairs
    - Example: Filtering out key-value pairs we don't need
  - `map` can emit more than one intermediate key-value pair for each 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



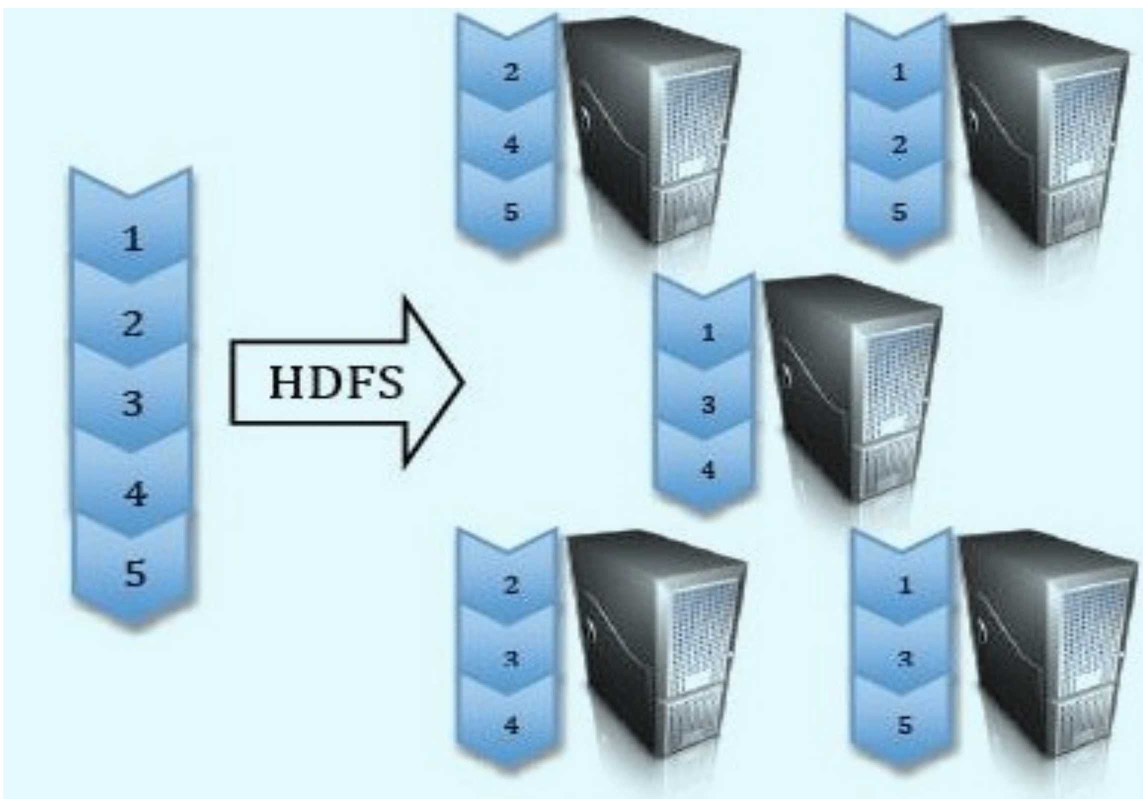
http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

MapReduce

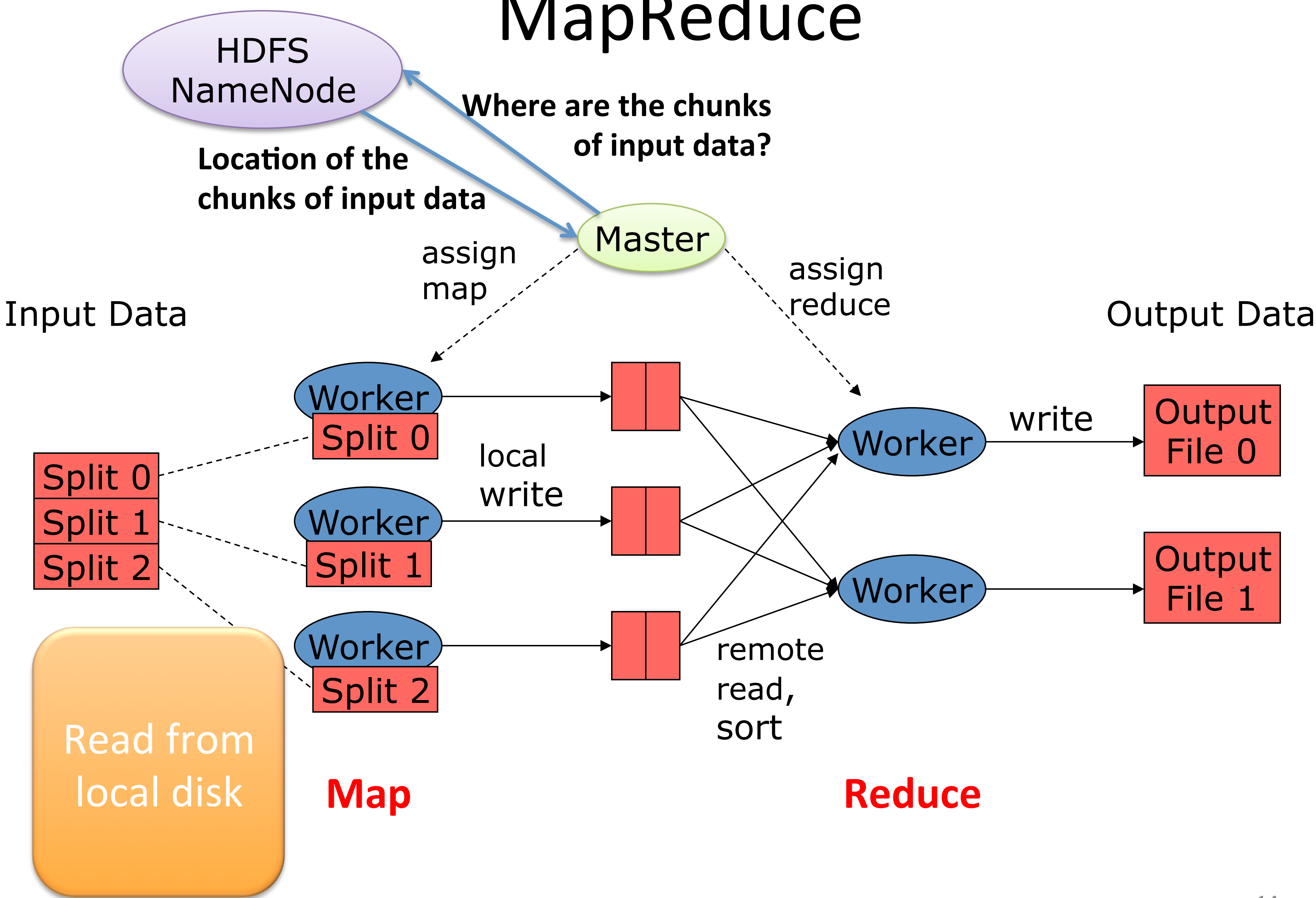

Hadoop Program
fork
fork
fork
Master
assign map
assign reduce
Input Data
Output Data
Split
Split
Split
Transfer peta-scale data through network
Worker
Worker
Worker
local write
Worker
Worker
write
remote read, sort
Output File 0
Output File 1
Map
Reduce

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

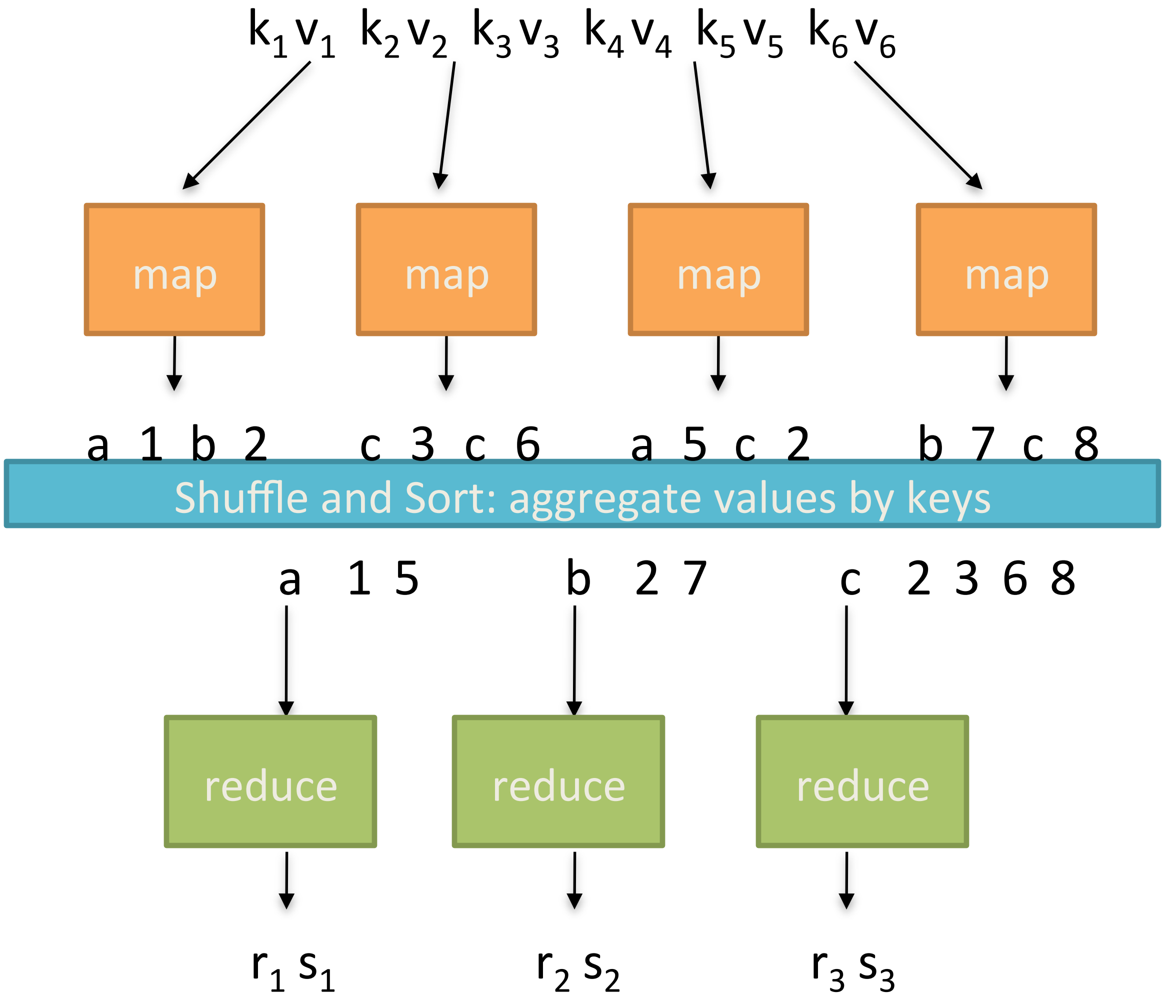
- Split data and store 3 replica on commodity servers



MapReduce


HDFS
NameNode
Where are the chunks
of input data?
Location of the
chunks of input data
Master
assign
map
assign
reduce
Input Data
Output Data
Worker
Split 0
Worker
Split 1
Worker
Split 2
Split 0
Split 1
Split 2
local
write
remote
read,
sort
Worker
Worker
write
Output
File 0
Output
File 1
Read from
local disk
Map
Reduce

MapReduce


k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce

- Programmers specify two functions:

  map $(k, v) \rightarrow <k', v'>*$

  reduce $(k', v') \rightarrow <k', v'>*$

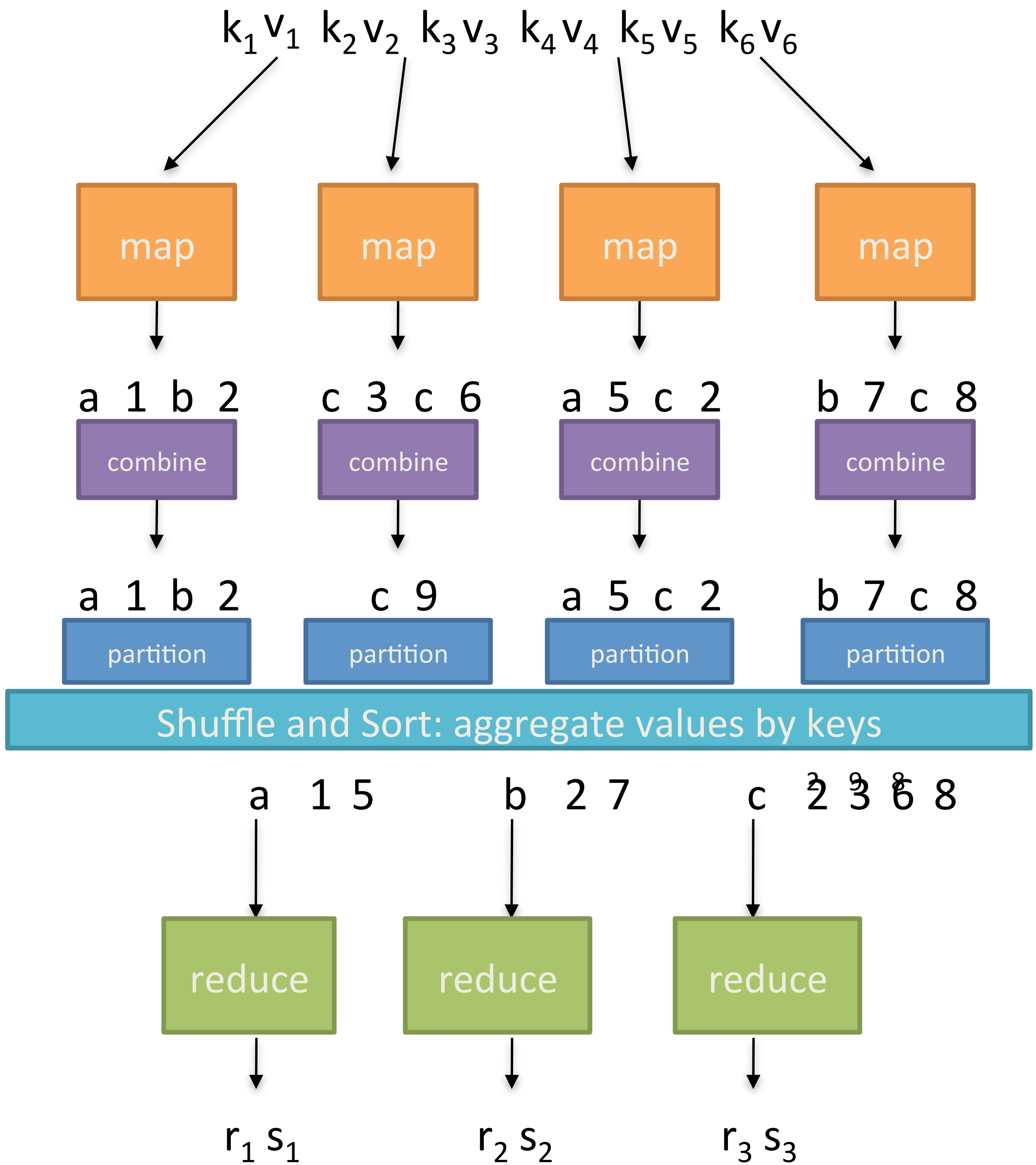
  - All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, programmers also specify:

  partition (k', number of partitions) → partition for k'

  - Often a simple hash of the key, e.g., hash(k') mod R
  - Divides up key space for parallel reduce operations

  combine $(k', v') \rightarrow <k', v'>*$

  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic



k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
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
r1 s1
r2 s2
r3 s3

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

Some additional details

- To make this work, we need a few more parts…

- The file system (distributed across all nodes):
  - Stores the inputs, outputs, and temporary results
  - We created a new file system…
- The driver program (executes on one node):
  - Specifies where to find the inputs, the outputs
  - Specifies what mapper and reducer to use
  - Can customize behavior of the execution
- The runtime system (controls nodes):
  - Supervises the execution of tasks
  - Esp. JobTracker

Some details

- Fewer computation partitions than data partitions
  - All data is accessible via a distributed filesystem with replication
  - Worker nodes produce data in key order (makes it easy to merge)
  - The master is responsible for scheduling, keeping all nodes busy
  - The master knows how many data partitions there are, which have completed – atomic commits to disk
- Locality: Master tries to do work on nodes that have replicas of the data
- Master can deal with stragglers (slow machines) by re-executing their tasks somewhere else

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 3.2×10^{15} bytes of data (2.8PB) and wrote 9.9×10^{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

Map and Reduce in Python

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

Map and Reduce in Python

```
#!/usr/bin/env python

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
```

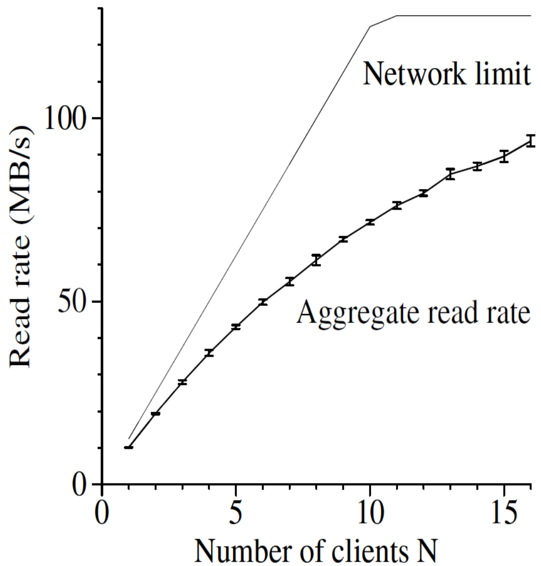
```
        count = int(count)
    except ValueError:
      # count was not a number, so silently
       # ignore/discard this line
     continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

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

What should expect from GFS

- Designed for Google's application
  - Control of both file system and application
  - Applications use a few specific access patterns
    - Append to larges files
    - Large streaming reads
  - Not a good fit for
    - low-latency data access
    - lots of small files, multiple writers, arbitrary file modifications
- Not POSIX, although mostly traditional
  - Specific operations: RecordAppend

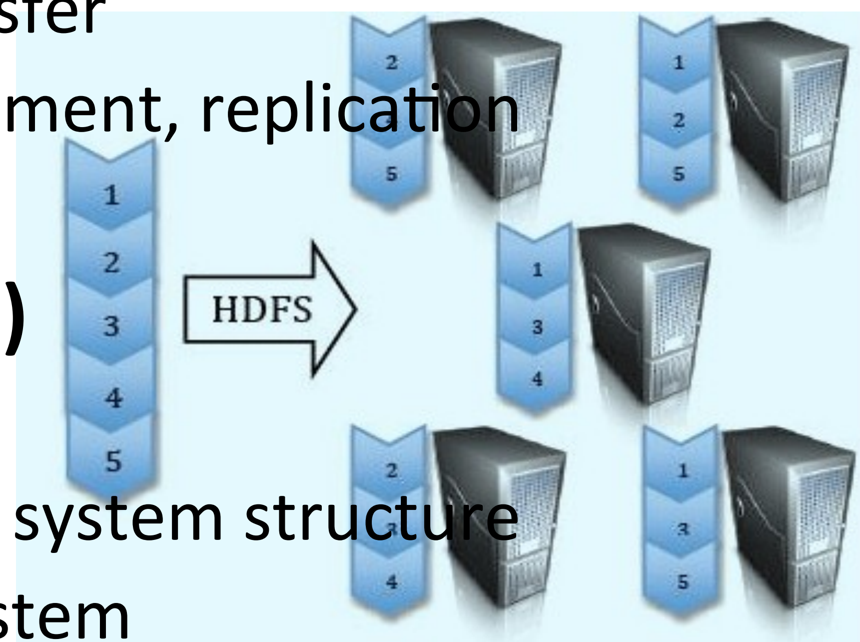


- Different characteristic than transactional or the "customer order" data : "write once read many (WORM)"
  - e.g. web logs, web crawler's data, or healthcare and patient information
  - WORM inspired MapReduce programming model
- Google exploited this characteristics in its Google file system [SOSP'03]
  – Apache Hadoop: Open source project HDFS

Components

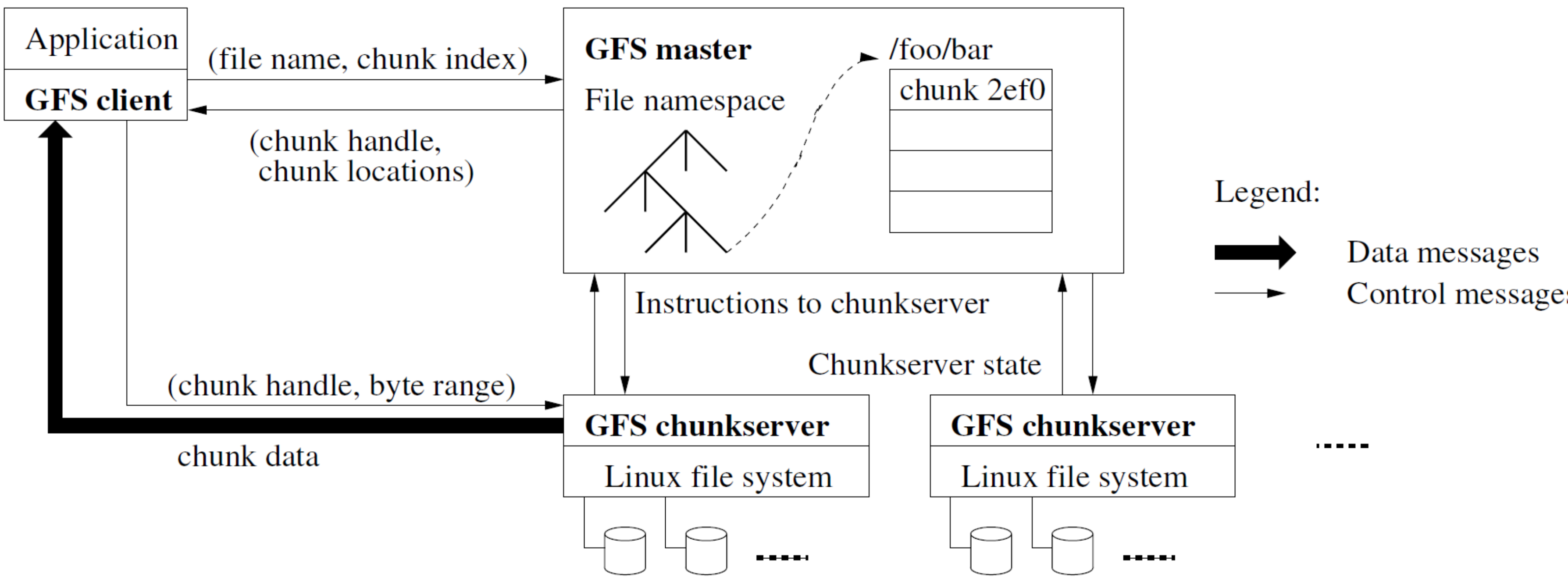
- **Master (NameNode)**
  - Manages metadata (namespace)
  - Not involved in data transfer
  - Controls allocation, placement, replication

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



www.cse.buffalo.edu/~okennedy/courses/cse704fa2012/2.2-HDFS.pptx

GFS Architecture



Write operation

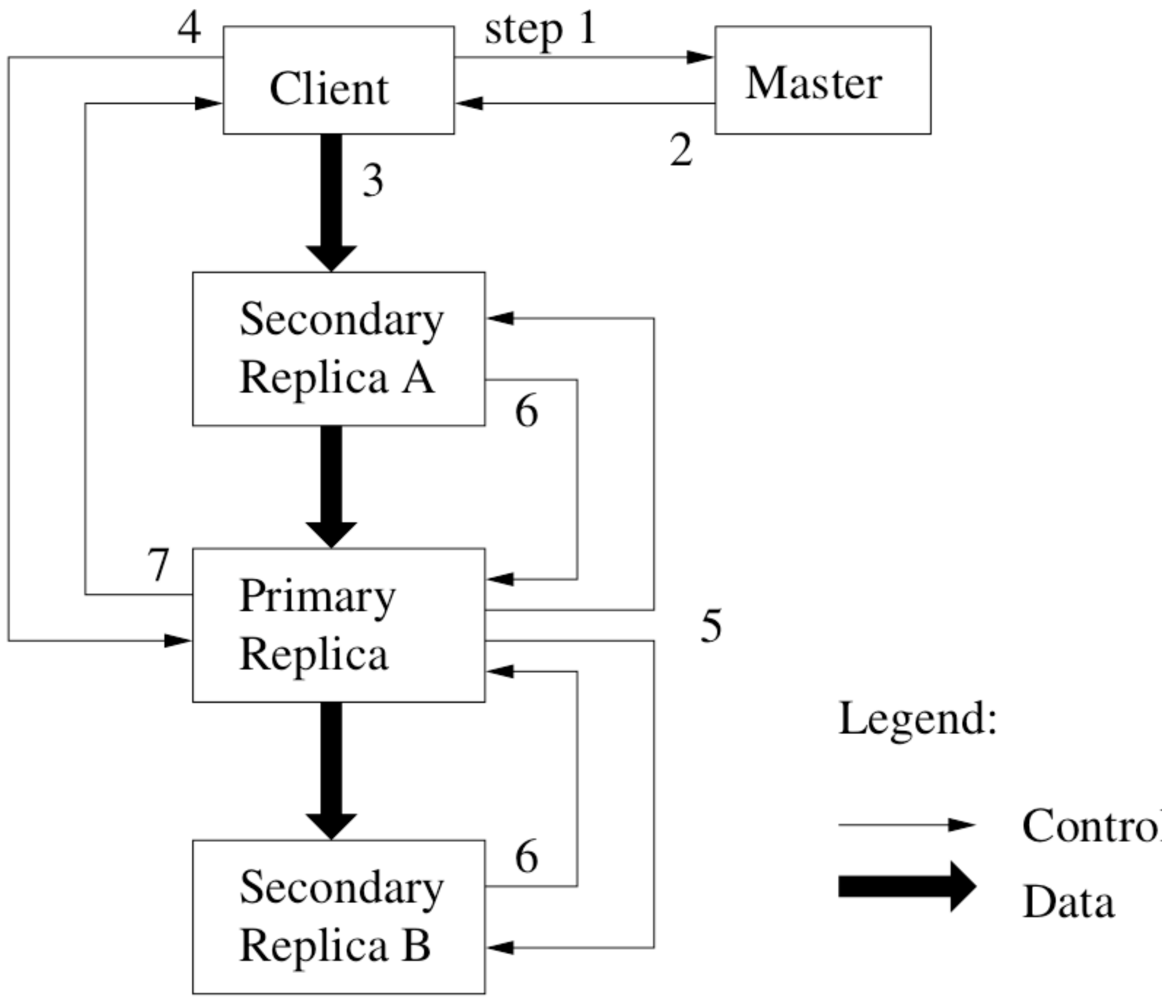
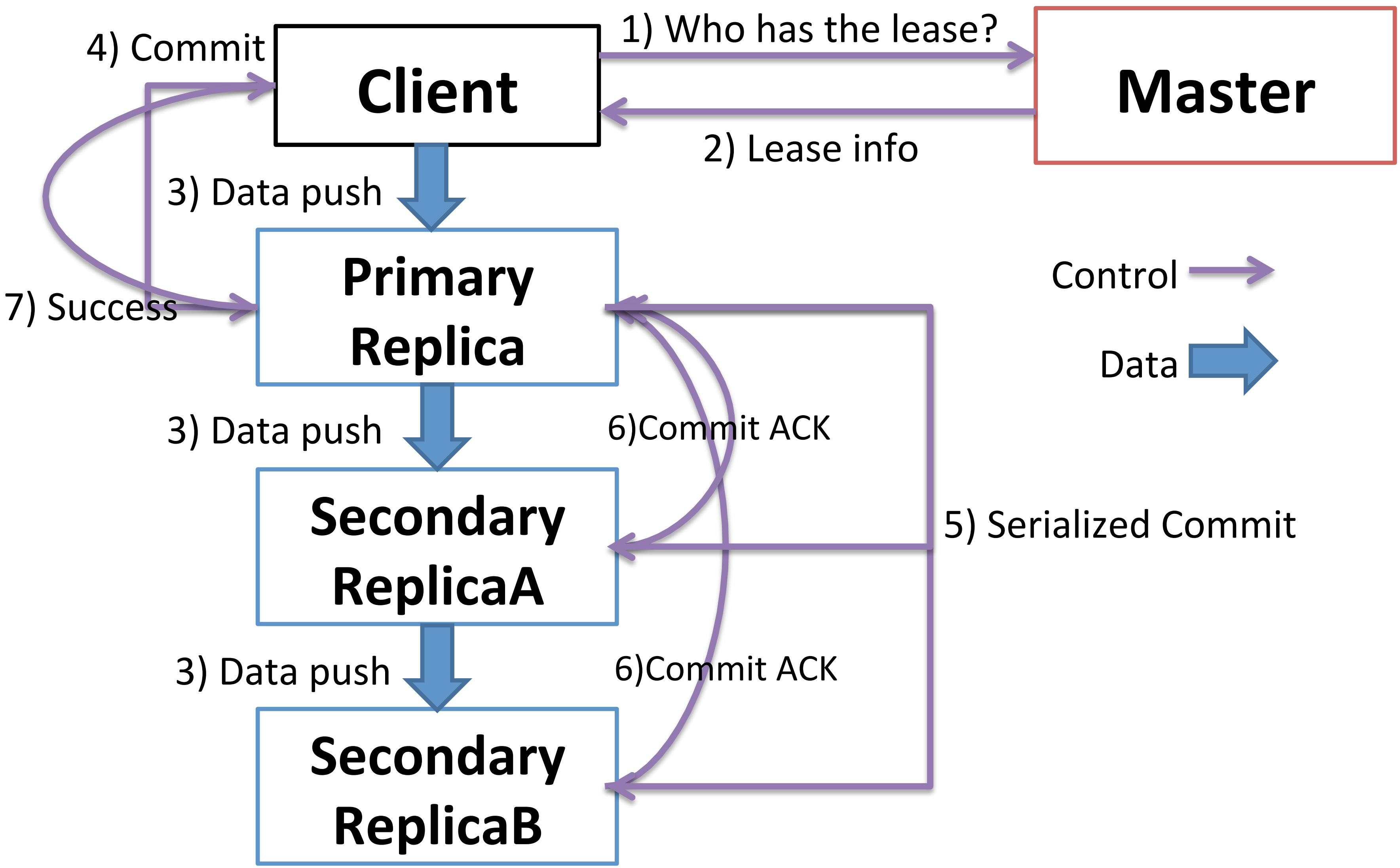


Figure 2: Write Control and Data Flow

Write(filename, offset, data)



Contents

- Motivation
- Design overview
  - Write Example
  - Record Append
- **Fault Tolerance & Replica Management**
- Conclusions

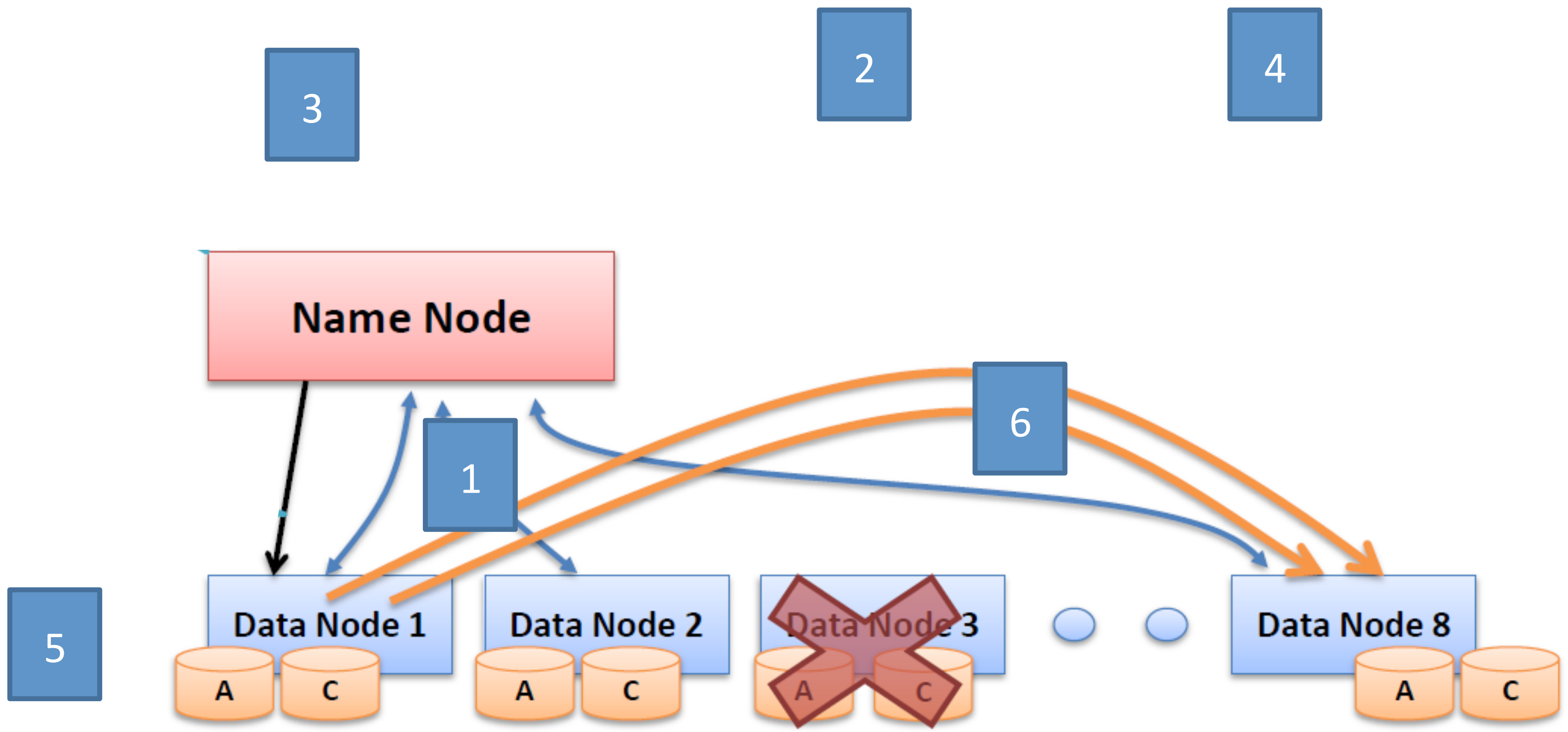
Fault tolerance

- Replication
  - High availability for reads
  - User controllable, default 3 (non-RAID)
  - Provides read/seek bandwidth
  - Master is responsible for directing re-replication if a data node dies
- Online checksumming in data nodes
  - Verified on reads

Replica Management

- Bias towards topological spreading
  - Rack, data center
- Rebalancing
  - Move chunks around to balance disk fullness
  - Gently fixes imbalances due to:
    - Adding/removing data nodes

Missing Replicas



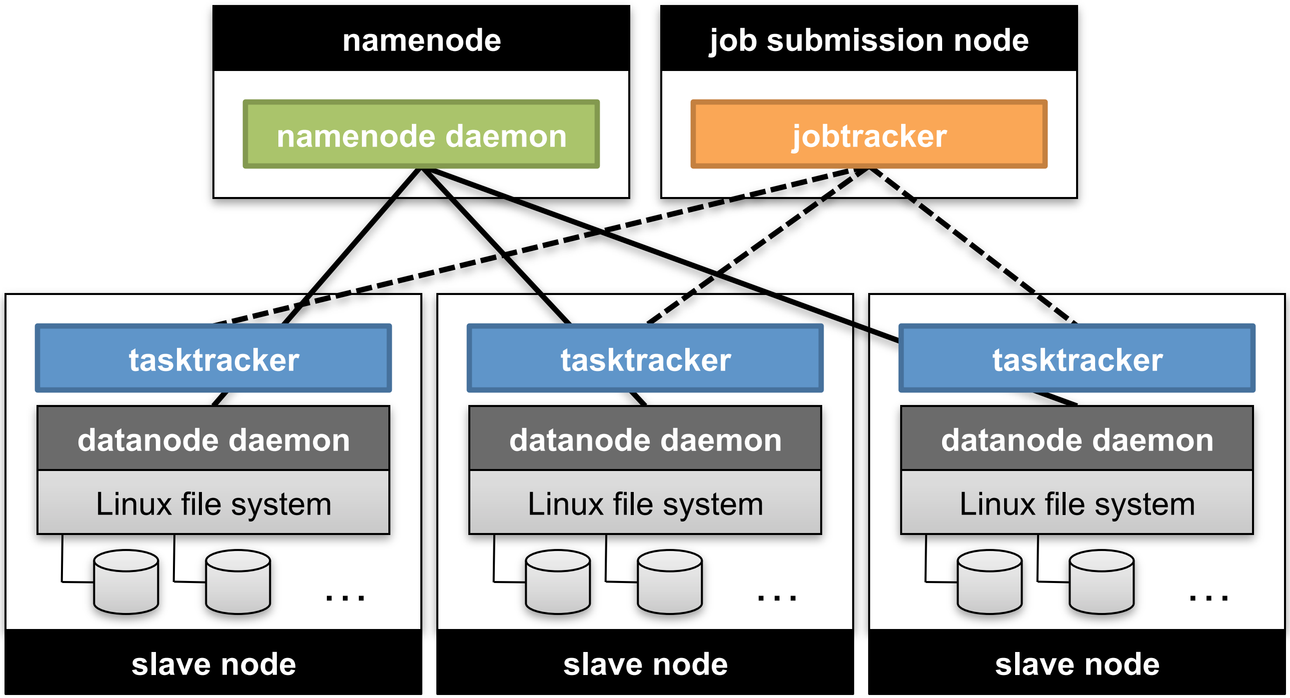
Garbage Collection

- Master does **not** need to have a strong knowledge of what is stored on each data node
  - Master regularly scans namespace
  - After GC interval, deleted files are removed from the namespace
  - Data node periodically polls Master about each chunk it knows of.
  - If a chunk is forgotten, the master tells data node to delete it.

Limitations

- Master is a central point of failure
- Master can be a scalability bottleneck
- Latency when opening/stating thousands of files
- Security model is weak

Putting everything together…



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