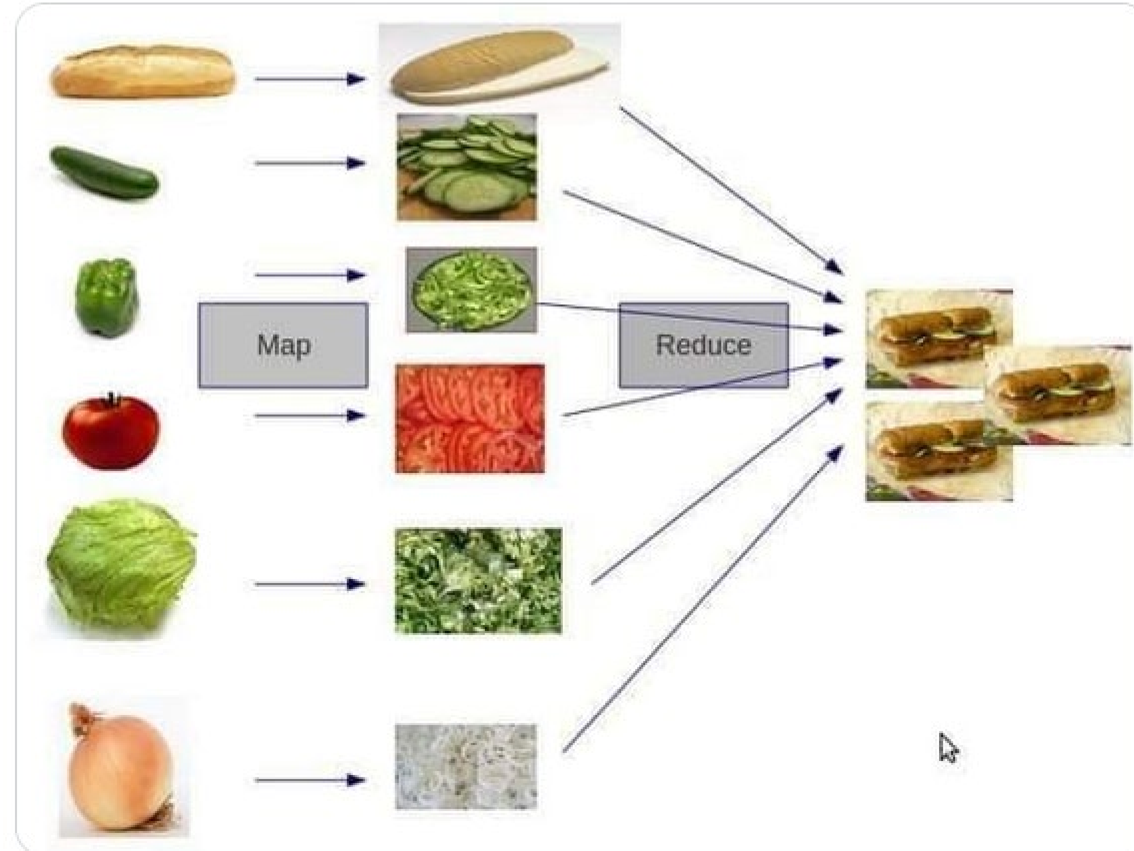
MapReduce



Exam 1 Logistics

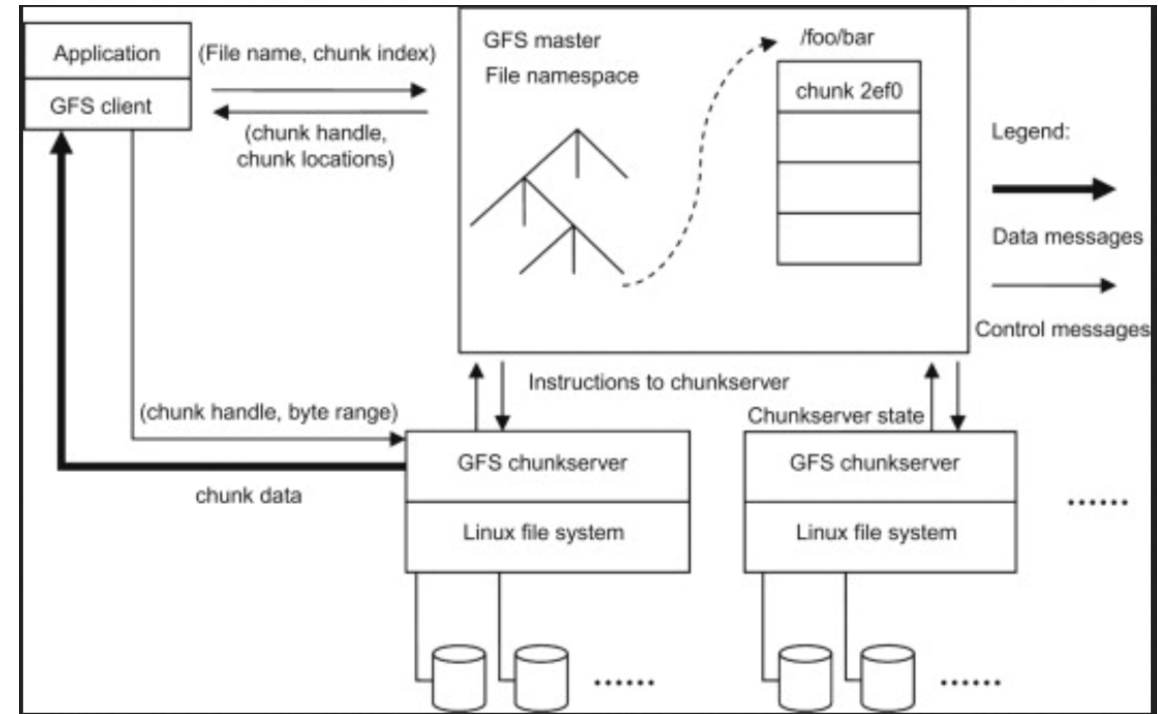
- Available Saturday 11AM Eastern (URL to be posted on Piazza)
- If you have questions:
  - Kevin: 11-3PM Eastern Saturday/Sunday
  - Emily: 1PM-5PM Eastern Saturday/Sunday
  - Kevin: 5-7PM Eastern Saturday/Sunday
  - Andrew: 9pm-11pm Eastern Saturday
  - Andrew: 10am-12pm Eastern Sunday
  - Andrew: 3am-4am Eastern Monday
- Private Piazza posts or Slack DMs *only* until Monday noon eastern

Outline

- **Review: GFS distributed system**
- MapReduce
- Writing MapReduce programs
- Fault Tolerance

Review: Google File System

- Use **multiple computers** to store more data than fits on one
    - GFS is a **distributed** filesystem among multiple **chunkservers** controlled by a **master server**
- Distributed systems difficult to reason about because of parallel operations
    - Reads are okay, but **writes**
    - **Concurrent writes** may threaten **consistency**: all replicas share same view of a file
      definition: all writes are reflected in the final written file



Review: Google Filesystem

- Sequential write: **One client** writes **sequential changes**
  - **Consistency** and **Definition** maintained

- Concurrent write: **Multiple clients** write **changes**
  - **Successful** if the writes occur *in the same order* on *all replicas*
  - **Consistency** is maintained, but *not necessarily* **definition**

- Failed write: **Multiple writes** that yield **inconsistent** replicas

Review: Google Filesystem

- Focus on **Appending files** rather than *writing changes*
  - Consider: Logging applications, ML training datasets only grow

- Use **atomic locking** to guarantee **consistency and definition**
  - Have **master server** maintain a *lock* when a client requests a write
  - No other server can write until the lock is released
    - NB: definition is maintained if *reads* are disabled while locked as well
    - This is **slow**

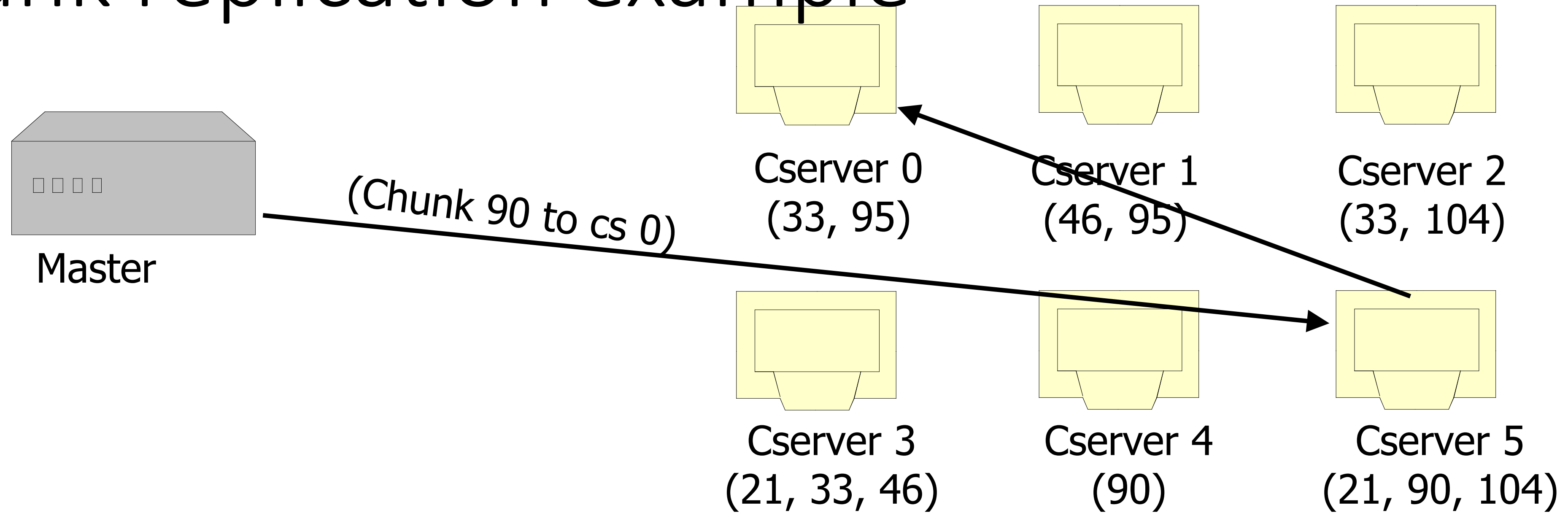
- Have **master server** assign a **primary** chunkserver to each chunk
  - If you want to write, send it to the *primary*
    - Primary figures our replication (and thus guarantees consistency)

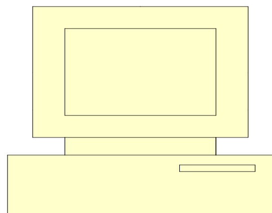
Chunkserver fault tolerance

- Chunkservers report to master every few seconds
  - *Heartbeat* message
- If the master loses the heartbeat, marks the server as down
- Master asks chunkservers to reduplicate data

- Rule of thumb: 3 replicas of each chunk

Chunk replication example





1. Always keep at least *k* copies of each chunk
2. Imagine cserver 4 dies; #90 lost
3. Master loses heartbeat, decrements #90's reference count. Asks cserver 5 to replicate #90 to cserver 0
4. Choosing replication target is tricky

Chunk replication design decisions

- How many copies of each chunk?
- How do you choose a chunkserver target?

- GFS decisions:
  - As a rule, 3 copies, but configurable
  - A traditional rack of machines generally contains a bunch of servers (~20-40) and a network switch
  - Lots of intra-rack bandwidth, limited inter-rack bandwidth
  - Keep most copies within the rack

Master failure

- What if the master node goes down?
  - Entire system is down (lol whoops)
- Minutes down time
  - Automatic failover later added
- 10 seconds
  - Best achieved
  - Still too high!

Master failure

- Master maintains critical data structures
  - filename -> chunkid map
  - chunkid -> location map
- Master writes a log to disk when data structures change
  - **Shadow master** consumes log and keeps copies of these data structures up-to-date
- If master goes down, shadow master provides *read-only access* until master restart

GFS strengths

- Store lots of data
- Fault tolerant
  - Too big to back up!
- High *throughput*
  - Can read lots of data from many chunkservers at same time

GFS weaknesses

- Bad for small files
  - Chunks are ~64MB
- Master node single point of failure
- High *latency*
  - Talk to two servers to fetch any data, might need multiple chunks

One-Slide Summary: MapReduce

- **MapReduce** is a programming paradigm for **distributed computation**
  - Even if you have **a lot of computers**, how do you **coordinate** them to **work together** on a **single task**?

- With MapReduce, you specify **Map** and **Reduce** functions that are applied over a **large input**
  - **Map**: Turn the input into <key, value> pairs
  - **Reduce**: Turn the <key, value> pairs into some desired output

- **Hadoop** is an implementation of MapReduce

Why MapReduce

- GFS: distributed system to **store more data** than possible on one computer

- MapReduce: distributed programming framework to do more **computation** than possible on one computer

Example MapReduce jobs

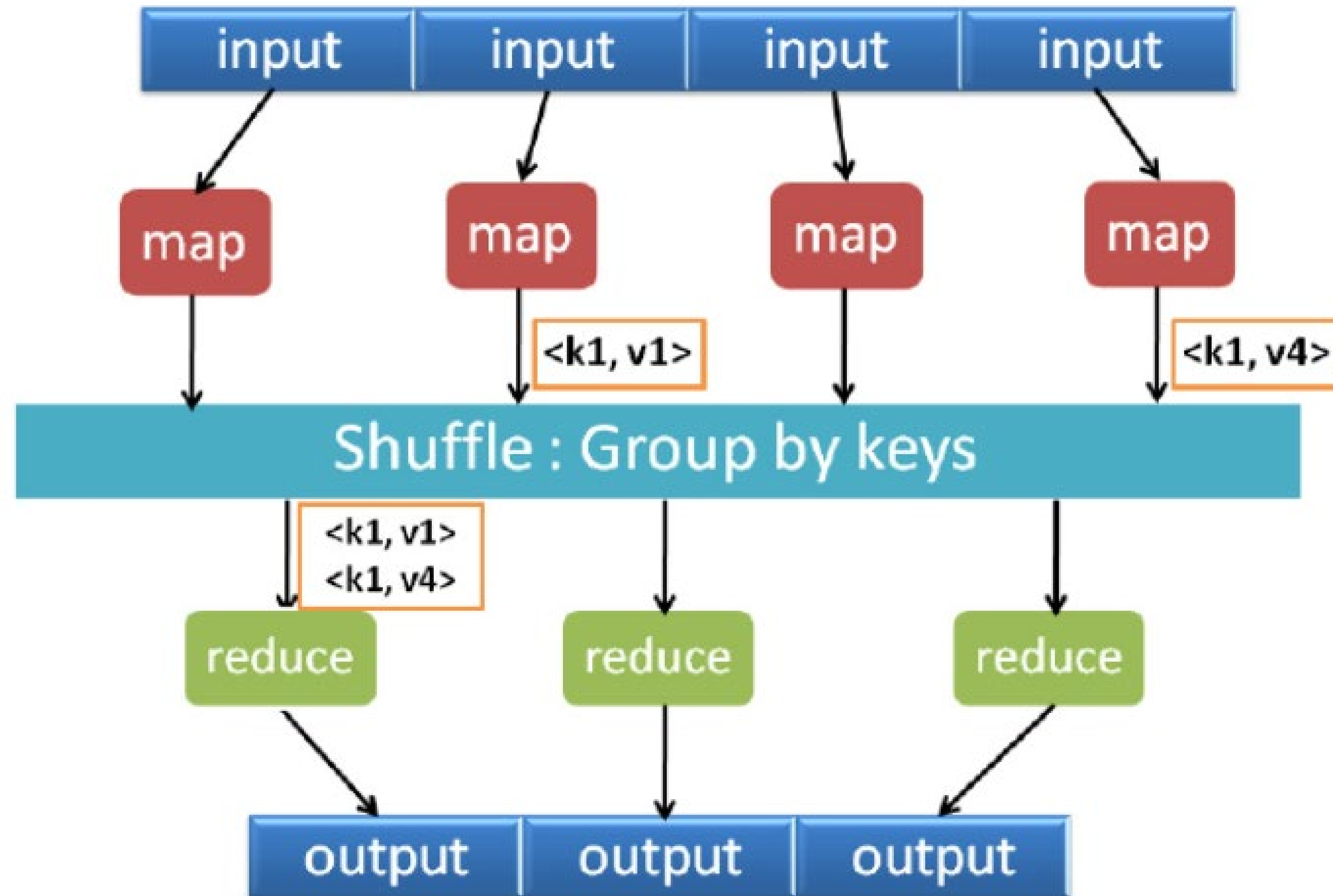
- Counting the number of words in 1 TB of text files
- Count number of requests to each web page, given a log
- Building a search index over large amount of files
- Lots of **independent counting** jobs where you can bring the work of multiple disparate systems together later

Big Idea

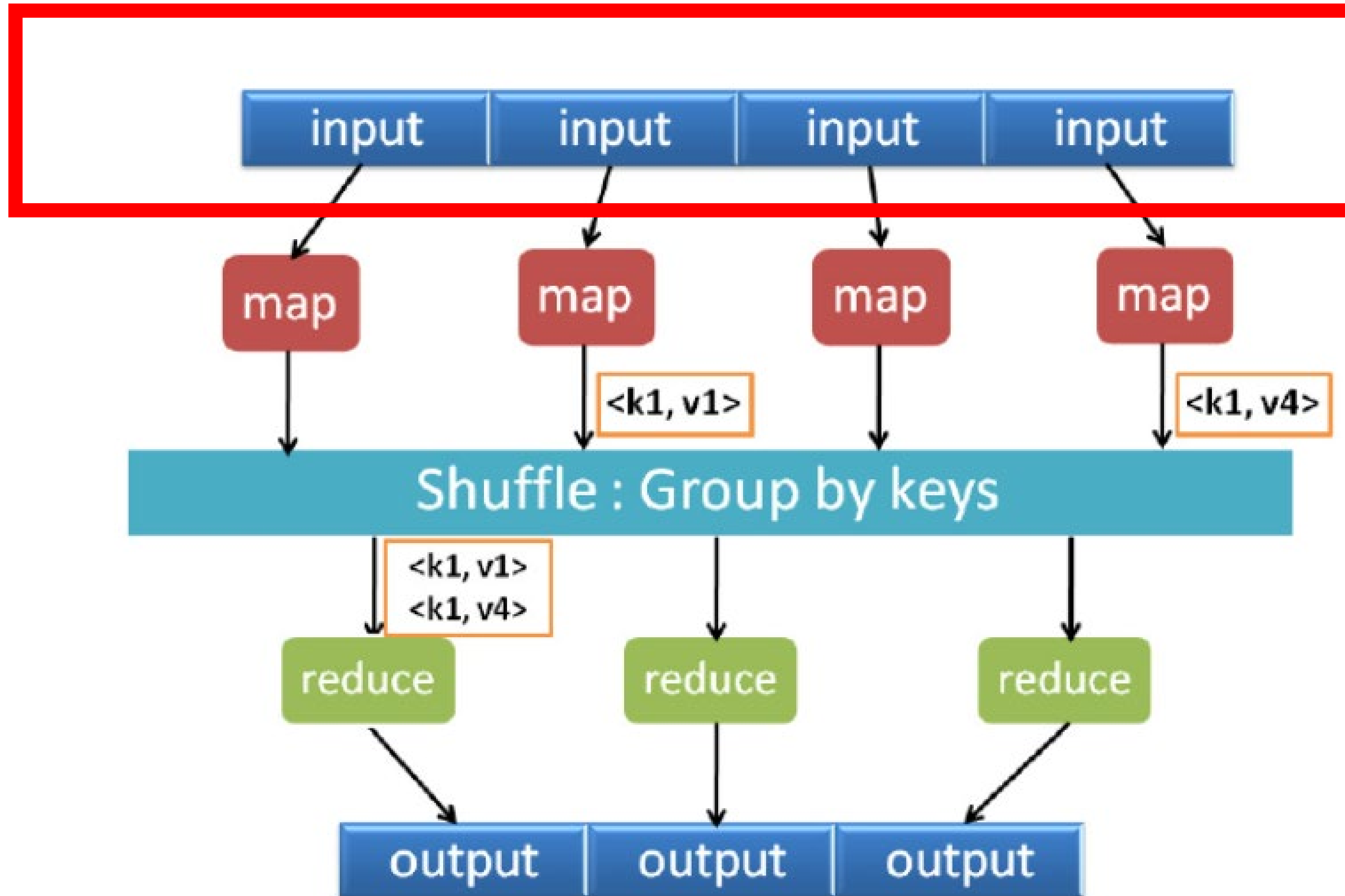
- Separate out work into multiple **phases**
  - **Map**
  - **Group**
  - **Reduce**

- Define those phases in a way that multiple computers can work on the phase without communicating
  - **Embarrasingly Parallel –** nodes in a system work independently in parallel
    - They’re “too embarrassed” to communicate with each other, they do things on their own

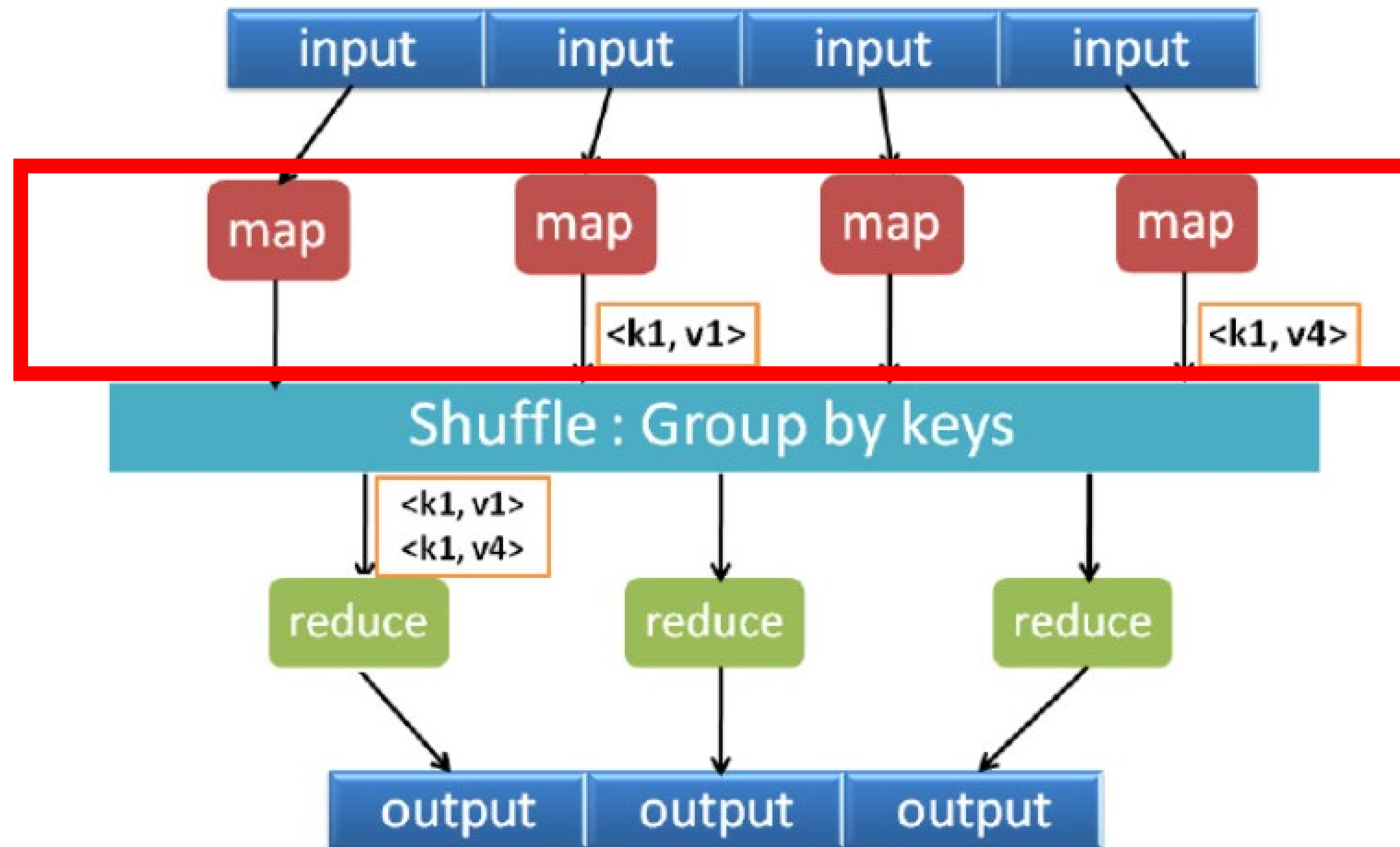
MapReduce Diagram



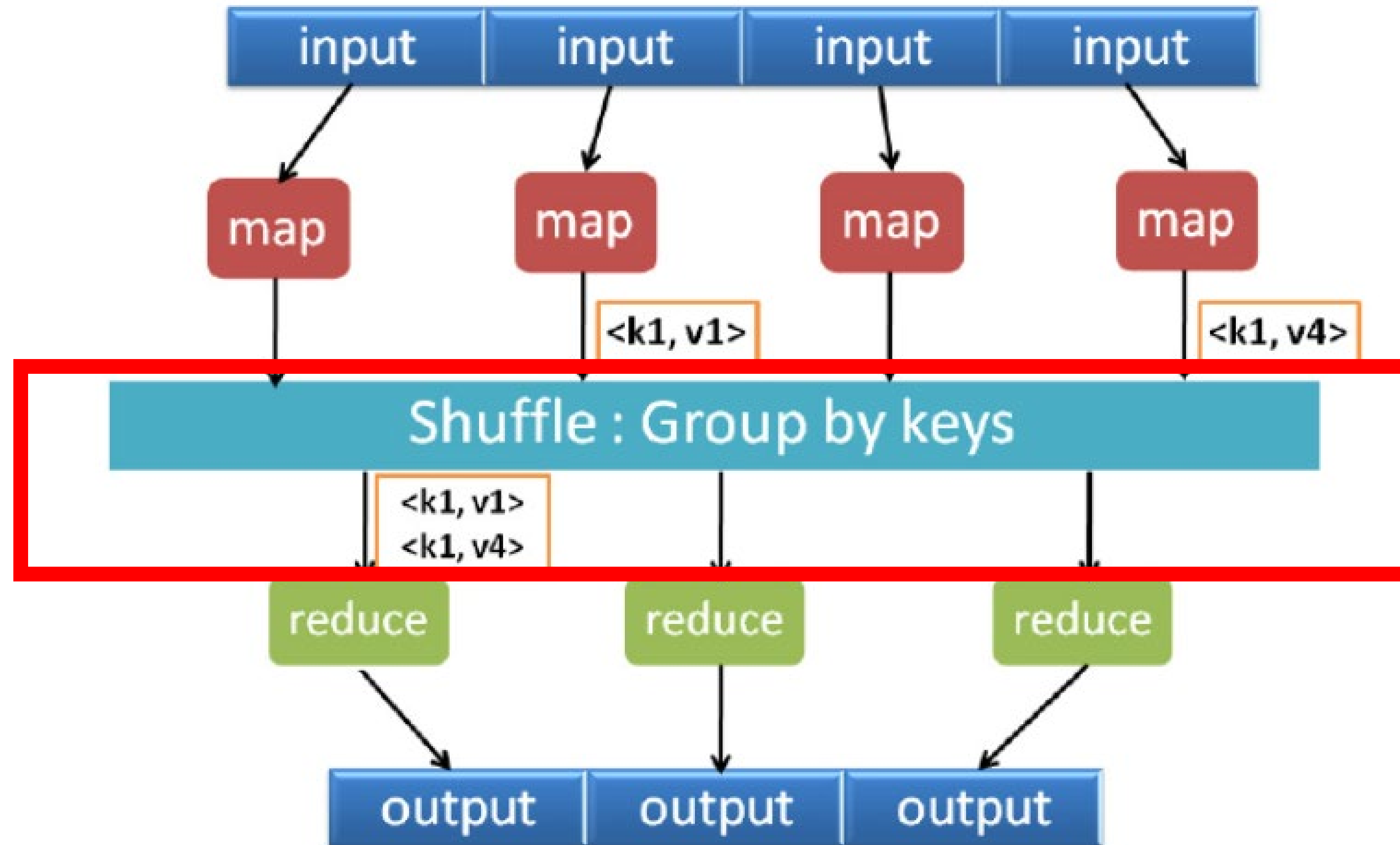
MapReduce Diagram



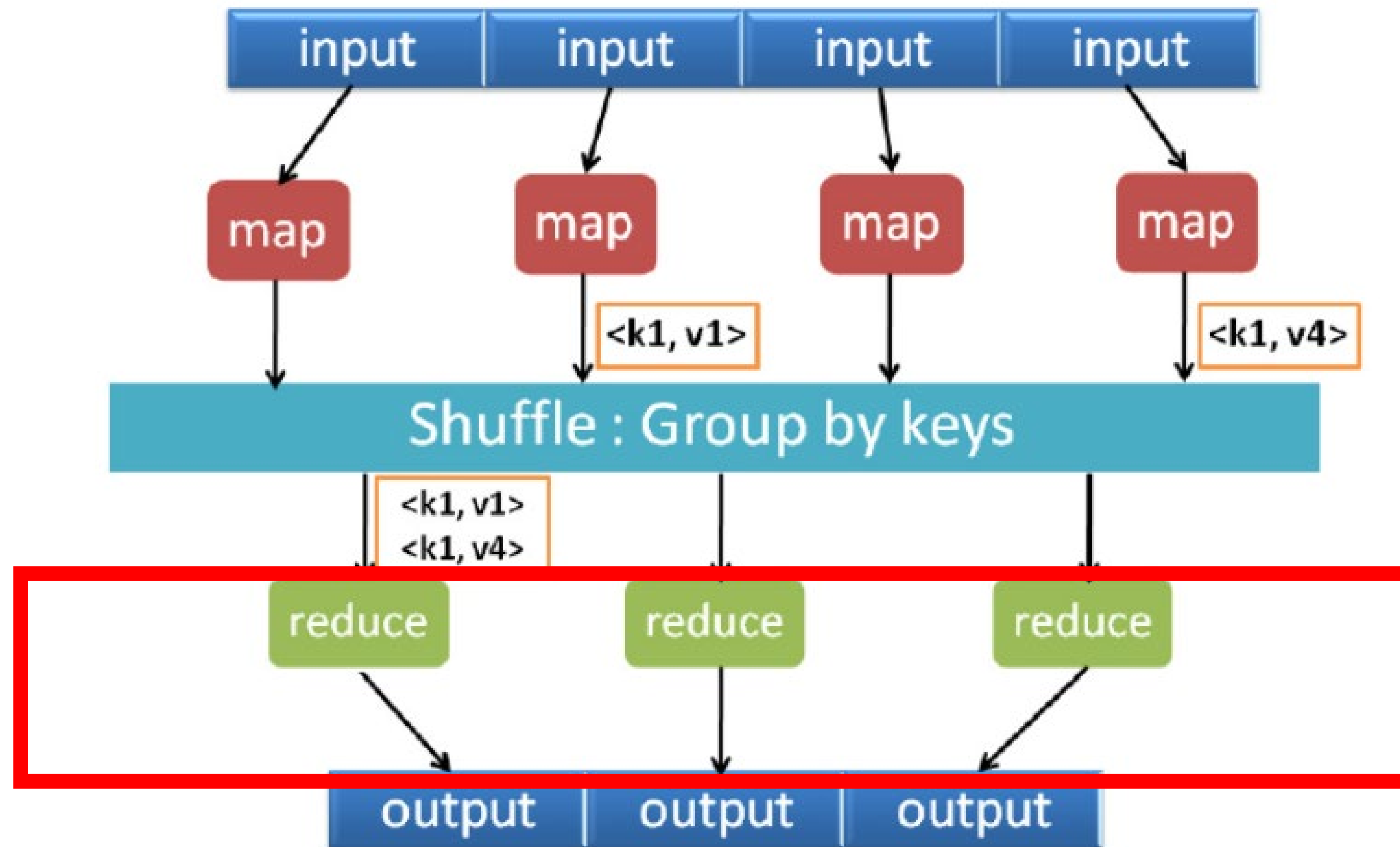
MapReduce Diagram



MapReduce Diagram



MapReduce Diagram



MapReduce by Analogy

- Imagine some frat or sorority has instituted an “initiation ritual”
- A pledge must compute:
  - How common are 1-character words? (‘a’, ‘I’, etc.)
  - How common are 2-character words? (‘an’, ‘be’, ‘is’, etc.)
  - … up to 10-character words
- Across the entire UM library
  - 12 million books
- The pledge can use 6,000 freshmen to help
- Output: <1, *number of 1-letter words*>, <2, *number of 2-letter words*>, etc.

MapReduce by Analogy

- Divide the 6000 helper freshmen into teams

- The **Mappers**
  - Thousands of people
- The **Grouper**
  - Just one person for now
- The **Reducers**
  - Around 10
- The Master
  - *You*

MapReduce by Analogy

- Each **mapper** freshman gets a "reading list" of 2,000 books
  - That's 12M books / ~6k freshmen
    - And a notepad

- **Map task:** write *one line* for *each word* in the reading list, as well as the *number of characters* in that word
    - 2, It
    - 3, was
    - 3, the
- … etc. many many many times

MapReduce by Analogy

- After the **mappers** are done, the **grouper** takes their output
- The **grouper** has a *10-page* notebook

- Recall: **mapper** produced lists of <X, string> pairs.
  - **Grouper** takes each pair and writes it in the corresponding page
    - Sheet 1: a, a, a, I, a, … many more
    - Sheet 2: if, if, an, if, at ... many more
    - ...
    - Sheet 10: schnozzles, mozzarella, etc.

  - **Output**: grouper has *binned* the mapper **values** together by X (the **key**)

MapReduce by Analogy

- Each of the **grouper's** 10 sheets goes to a **Reducer**

- Each **reducer** counts *the number of words* on their one assigned sheet, and writes the number in bold letters on the back
  - **Input:** Sheet 2: if, of, it, of, of, if, at, im, is, is, of, of …
  - **Output:** Some large number

- The **reducer** has reduced each group to the final output *for that group*
  - With 10 reducers, we'll get our final desired output
  - Count of words by word length!

MapReduce by Analogy: Key Observations

- The **mappers** can work independently
- The **reducers** can work independently

- The Grouper has a lot of work
  - But it’s easy work: just look at the <X, word> pair and copy “word” to **group** X

- **Mapper** step must complete before **grouping**
- **Grouping** step must complete before **Reducers**

MapReduce by Analogy

- Ideas for optimizations?

- Idea 1: the mappers don’t need write the words down, just "1" for each word of a certain length
  - It’s the count of each word-length we’re after, the word itself is irrelevant

- Idea 2: mappers do part of the reducer work: number of 1-letter words, 2-letter words, etc.
  - Each mapper can produce an “intermediate” sheet of groupings that can be reduced later, e.g., <X, count_of_words_of_length_X>, ….

MapReduce model

- Three phases

- **Map**: turn input into (key, value) pairs

- **Group**: put entries with the same key together

- **Reduce**: combine adjacent (key, value) pairs into a final answer

MapReduce framework

- Programmer provides:
  - Input: what data are you operating on?
  - map() function/program: how do you turn the Input into <k,v> pairs?
  - reduce() function/program: how do you turn the <k,v> pairs into Output?
- Some parts are always the same
  - Communication
  - Grouping
    - Given the interface required by Map and Reduce, this step is always the same!

Outline

- Review: GFS distributed system
- MapReduce
- **Writing MapReduce programs**
- Fault Tolerance

Today's Example

- Word count

- input: `["hello", "world", "hello"]`
- output: `[ ["hello", 2], ["world", 1] ]`

Map

- Map reads input from stdin* and produces a set of intermediate key/value pairs

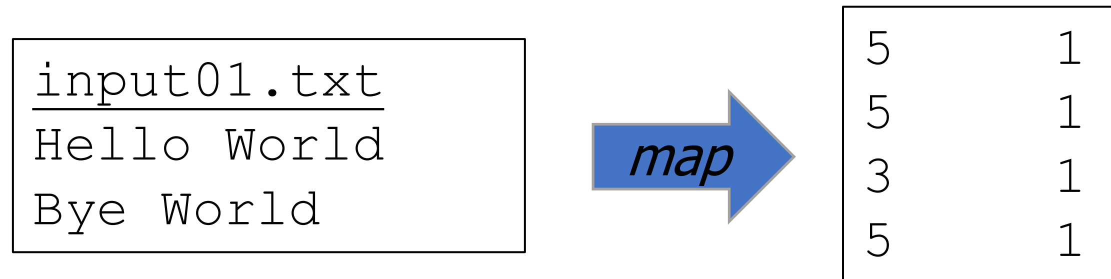
```
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        length = len(word)
        print(str(length) + "\t" + "1")
```

*In Hadoop, this is technically provided by a streaming interface to a GFS-equivalent: Hadoop Filesystem, HDFS

Map

```
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        length = len(word)
        print(str(length) + "\t" + "1")
```

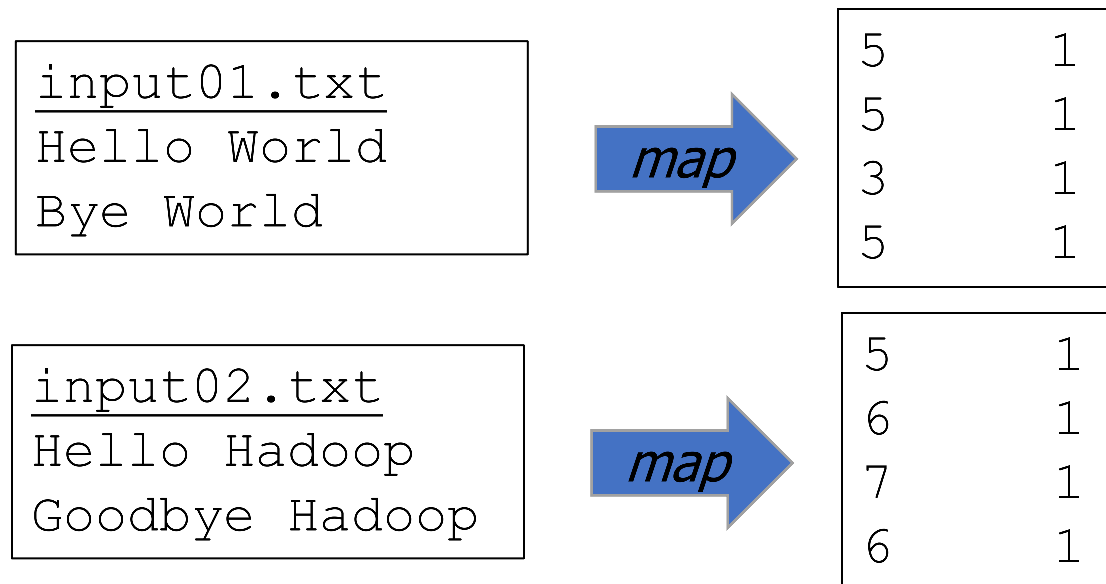
- Run map on segment of input
  - E.g., one input file



Map

```
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        length = len(word)
        print(str(length) + "\t" + "1")
```

- Run map on a second segment of input
  - E.g., second input file
- Key idea: do this in parallel, on a different machine



Group example

```
input01.txt
Hello World
Bye World
```

```
input02.txt
Hello Hadoop
Goodbye Hadoop
```

- Group by key
  - This is provided by MapReduce, you don't have to do anything here
- Order does not matter

map output

```
5    1
5    1
3    1
5    1
```

```
5    1
6    1
7    1
6    1
```

```
5    1
5    1
5    1
5    1
```

```
3    1
```

```
6    1
6    1
```

```
7    1
```

Reduce

- Reduce reads input from stdin* a list of intermediate key/value pairs. It merges them together to form a possibly smaller set of values.

```
import collections
import sys

counts = collections.defaultdict(int)

for line in sys.stdin:
    line = line.strip()
    length, count = line.split("\t")
    counts[length] += int(count)

for length, count in counts.items():
    print(str(length)+ "\t" + str(count))
```

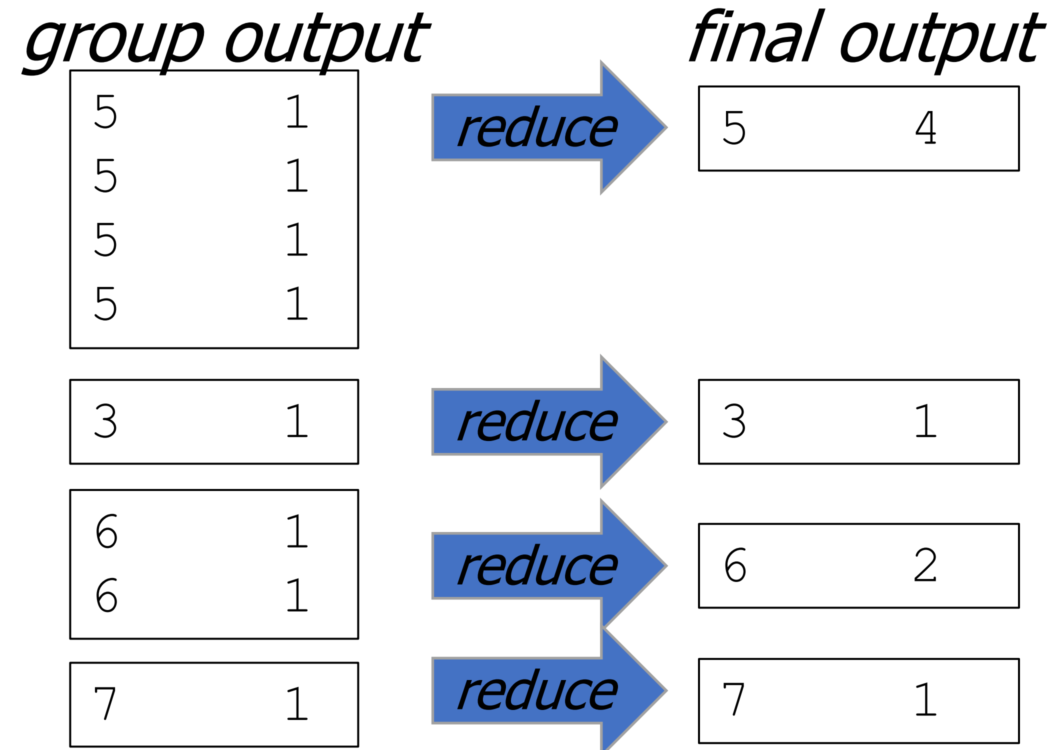

*As before, “stdin” would be the Hadoop Streaming Interface in a Hadoop deployment

Reduce

```
input01.txt
Hello World
Bye World
```

```
input02.txt
Hello Hadoop
Goodbye Hadoop
```

- Reduce each group
  - Output <word_length, occurrences> in this example
- Key idea: each group can be reduced in parallel



input01.txt
Hello World
Bye World

input02.txt
Hello Hadoop
Goodbye Hadoop

Reduce

- Multiple groups sent to same reducer also works
  - Just less parallelism
- Same output

group output

5	1
5	1
5	1
5	1
3	1

reduce →

final output

5	4
3	1

6	1
6	1
7	1

reduce →

6	2
7	1

Example, summarized

input01.txt
Hello World
Bye World

input02.txt
Hello Hadoop
Goodbye Hadoop

map →
5 1
5 1
3 1
5 1

map →
5 1
6 1
7 1
6 1

group →
5 1
5 1
5 1
5 1

3 1

6 1
6 1

7 1

reduce → 5 4

reduce → 3 1

reduce → 6 2

reduce → 7 1

Another Example: distributed Grep

- Describe map and reduce for distributed search
  - AKA `grep`
  - Print any line that contains the string "Hello"

Grep

- Map

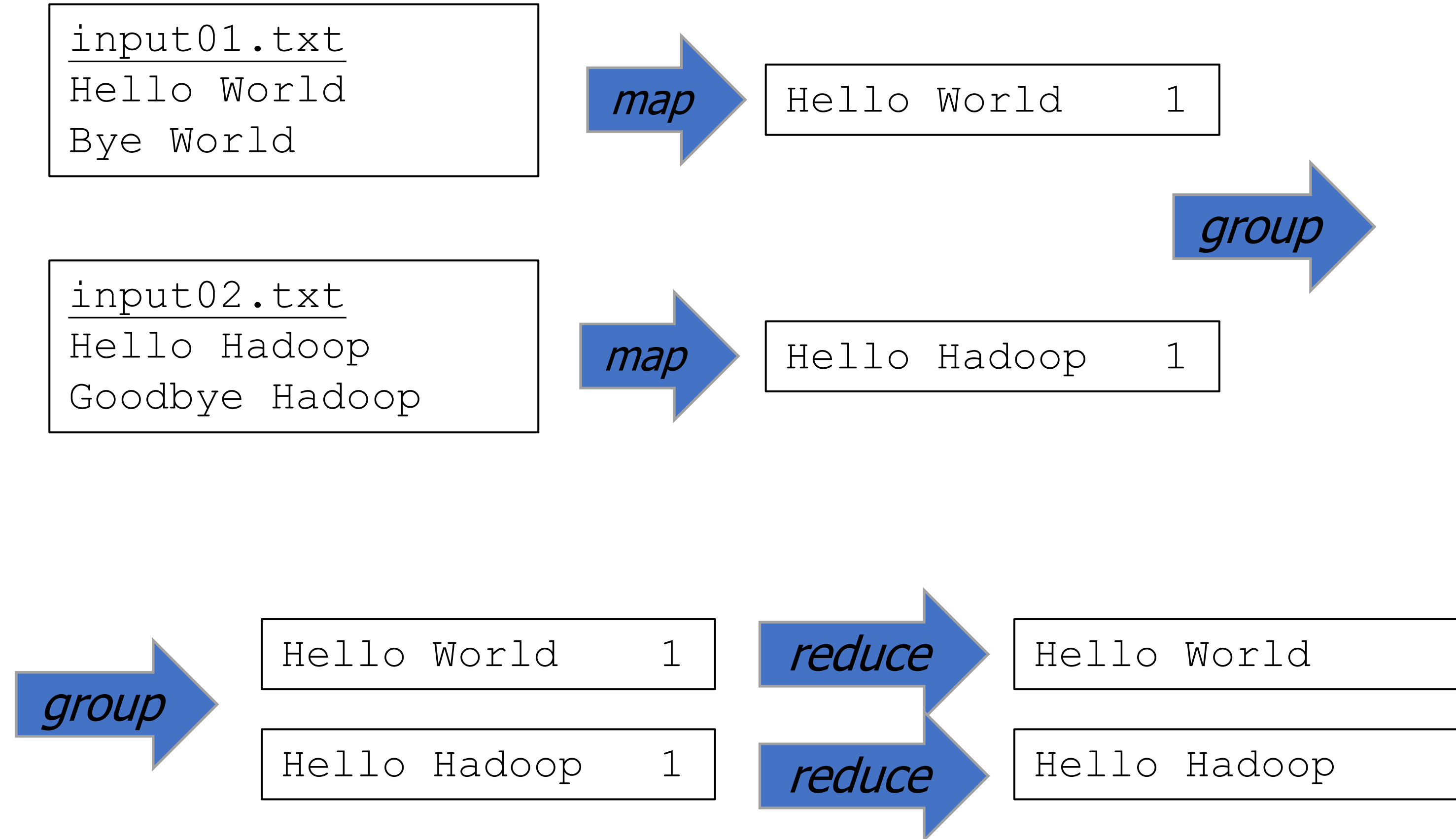
```
import sys
for line in sys.stdin:
    line = line.strip()
    if "Hello" in line:
        print(line + "\t" + "1")
```

- Reduce

```
import sys
for line in sys.stdin:
    line = line.strip()
    text, count = line.split("\t")
    print(text)
```

```python
# grep_map.py
import sys
for line in sys.stdin:
    line = line.strip()
    if "Hello" in line:
        print(line + "\t" + "1")
```

```python
# grep_reduce.py
for line in sys.stdin:
    line = line.strip()
    text, count = line.split("\t")
    print(text)
```



Too many reducers

- Map

```
import sys
for line in sys.stdin:
    line = line.strip()
    if "Hello" in line:
        print(line + "\t" + "1")
```

- Reduce

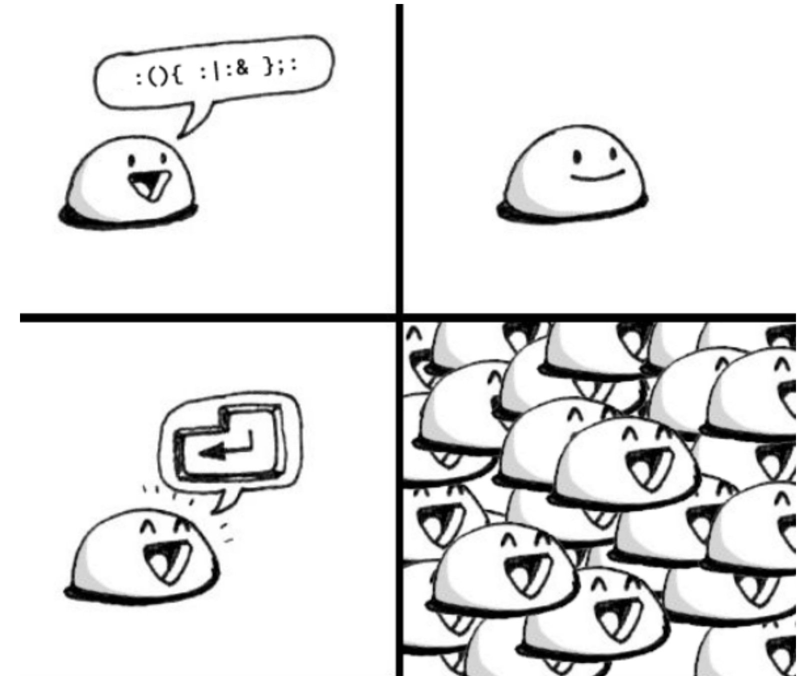
```
import sys
for line in sys.stdin:
    line = line.strip()
    text, count = line.split("\t")
    print(text)
```

- How many groups? How many reducers, in the worst case? How to fix?

Too many reducers

- How many groups?
  - One for every unique line
- How many reducers, in the worst case?
  - One for every unique line
- We get MANY reducers with VERY small inputs
  - Better to have a "moderate" number

Too many reducers



- Map

```
import sys
for line in sys.stdin:
    line = line.strip()
    if "Hello" in line:
        x = myhash(line) % 1024
        print(x + "\t" + line)
```

- Reduce

```
import sys
for line in sys.stdin:
    line = line.strip()
    x, text = line.split("\t")
    print(text)
```

- Use a hash function and modulo operator!
- Now we have 1024 reducers

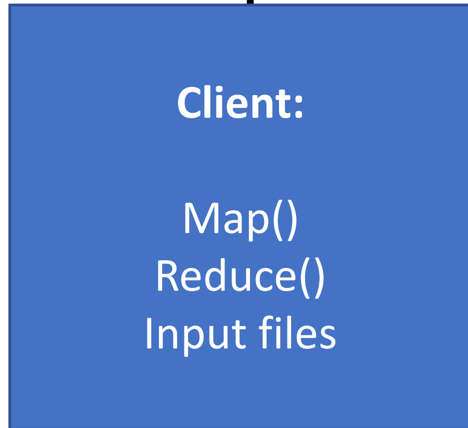
Outline

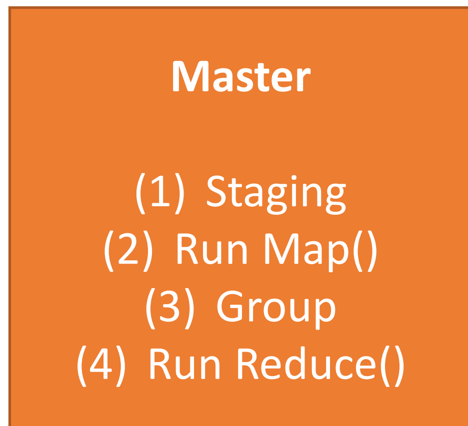
- Review: GFS distributed system
- MapReduce
- Writing MapReduce programs
- **Fault Tolerance**

MapReduce as Distributed System

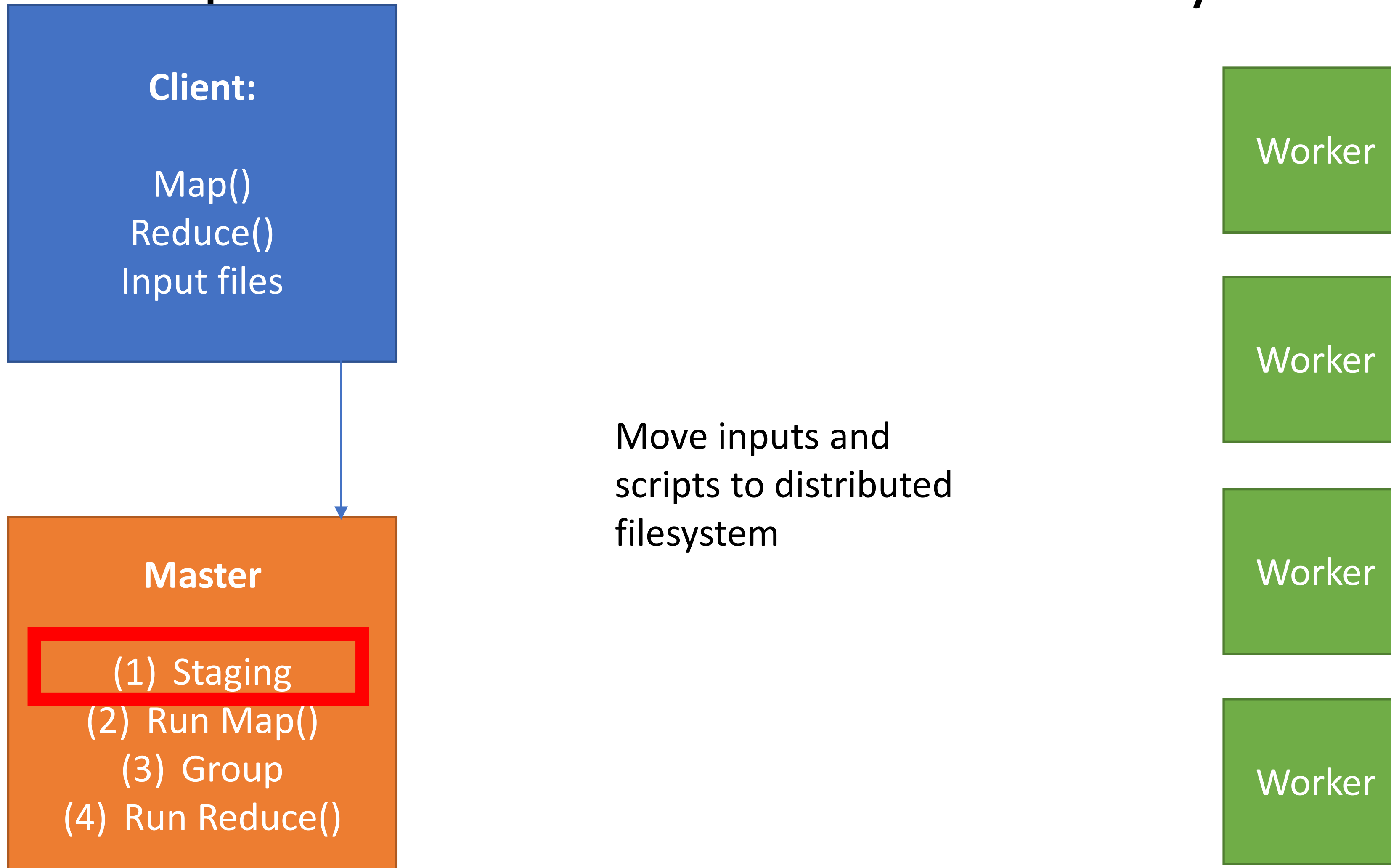
- Our Python system so far runs as a single program
- How to split it into multiple?

MapReduce as Distributed System





MapReduce as Distributed System



Move inputs and scripts to distributed filesystem

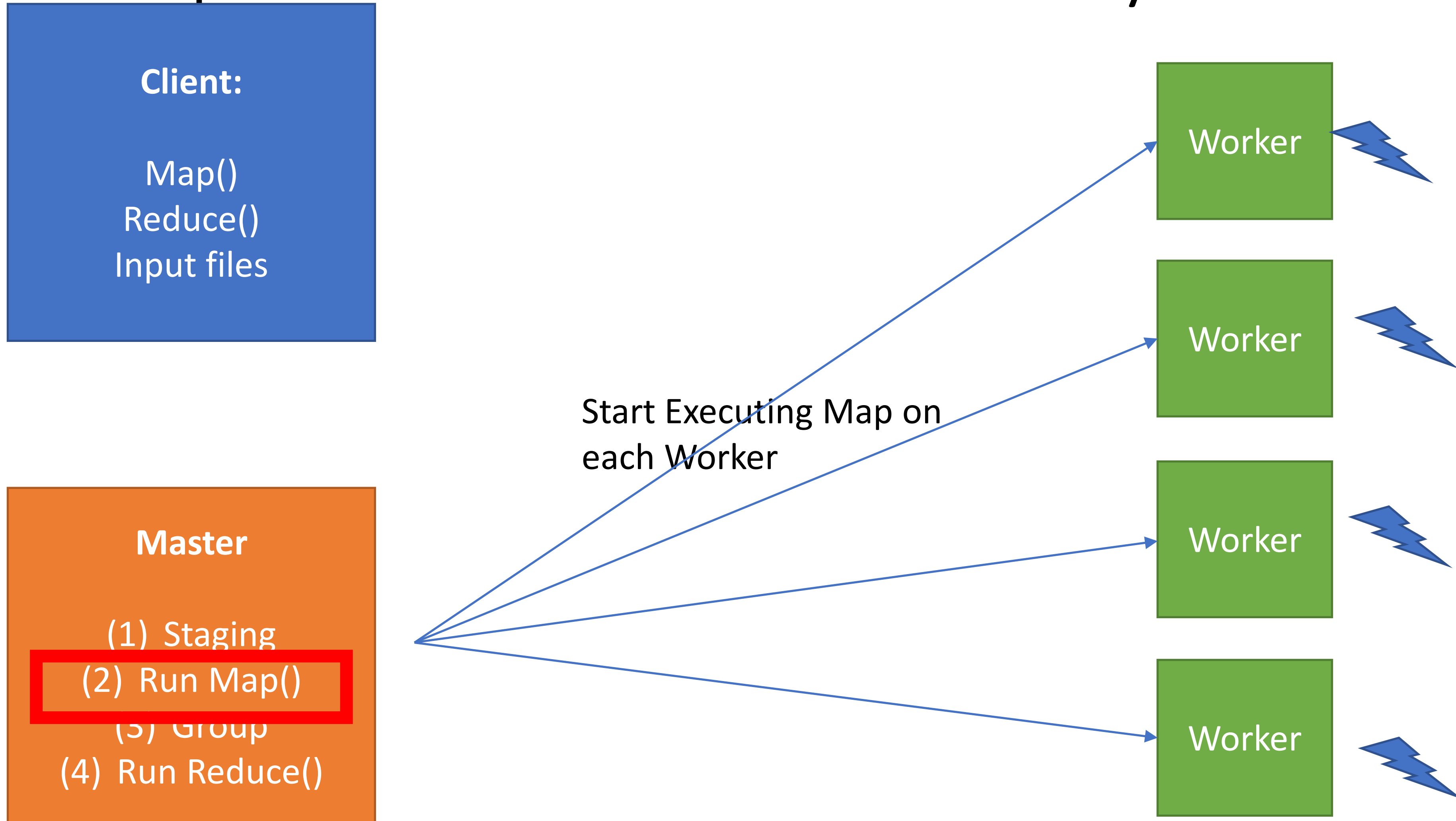
Worker

Worker

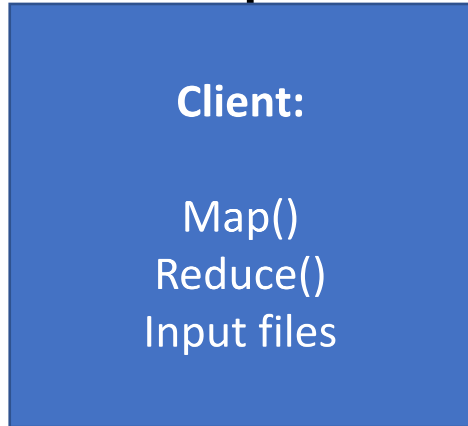
Worker

Worker

MapReduce as Distributed System

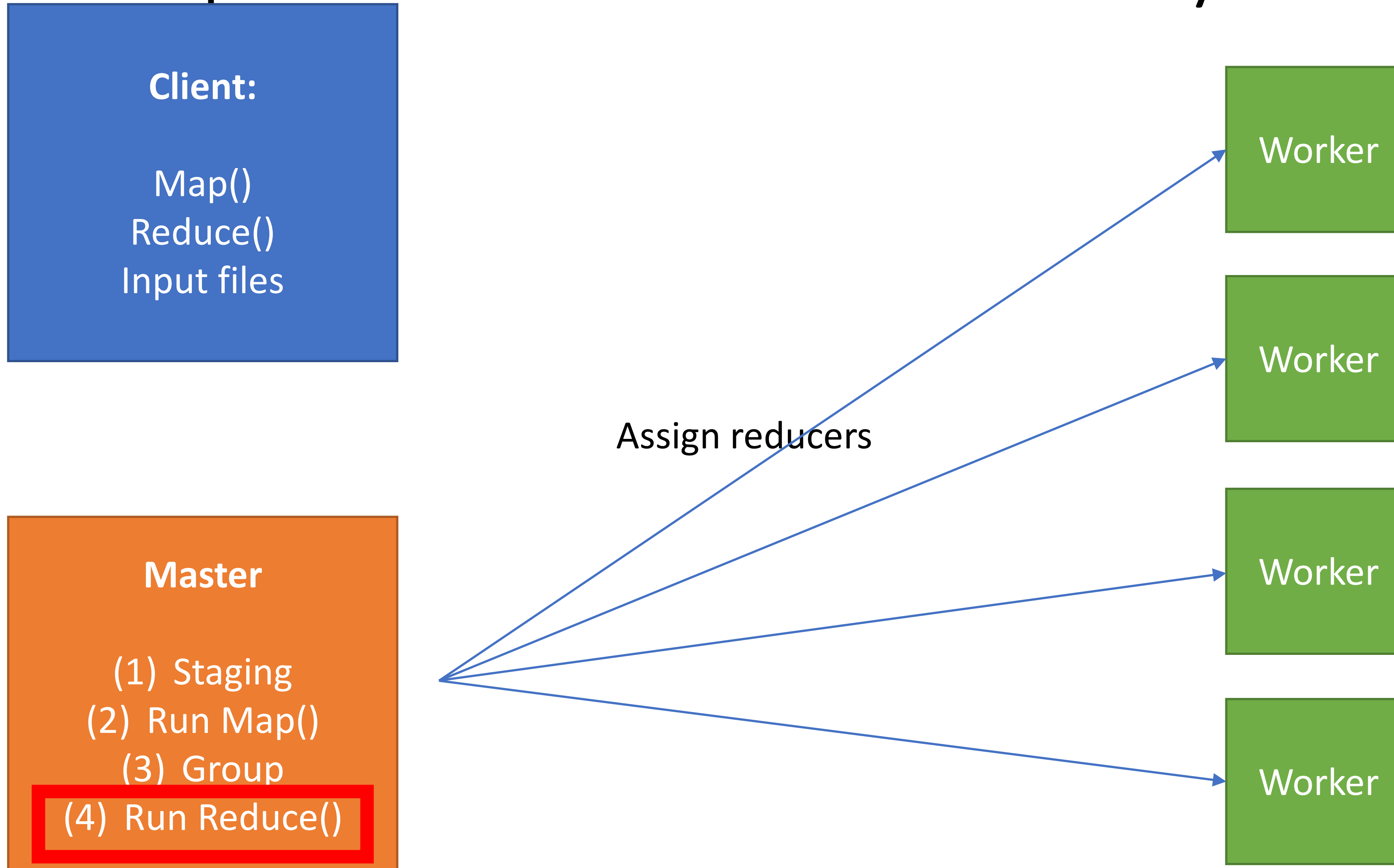


MapReduce as Distributed System



Master performs the Grouping step

MapReduce as Distributed System



P4 MapReduce jobs

- Master is coordinator and does grouping

- Workers execute shell commands
  - Can be Bash scripts, Python programs, ...
  - Data is shared using the file system

- Map and Reduce can be done on same pool of workers
  - Both are just shell commands operating on files

P4 vs. lecture exercises

- Today, map() and reduce() were functions, and the input/output lists
- In P4:
  - map and reduce are scripts
  - input and output are files

Fault tolerance

- How do we know if a machine goes down?

- Workers send periodic heartbeat messages to master
- Master keeps track of which workers are up
- Similar technique to Google File System

Fault tolerance

- What happens when a machine dies?
- Without MapReduce
  - Program (or query) is restarted
  - Not so hot if your job is in hour 23
- With MapReduce
  - If map worker dies
    - Just restart that task on a different box
    - You lose the map work, but no big deal
  - If reduce worker dies
    - Restart the reducer, using output from source mappers

Further reading

- Nice explanation from UC Berkeley
  - http://inst.eecs.berkeley.edu/~cs61a/book/chapters/streams.html#distributed-data-processing
- Some researchers disagree with MapReduce's popularity: “MapReduce: A Major Step Backwards”
  - https://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html
- Paper on Google's MapReduce framework "MapReduce: Simplified Data Processing on Large Clusters" by Jeffrey Dean and Sanjay Ghemawat
  - https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf