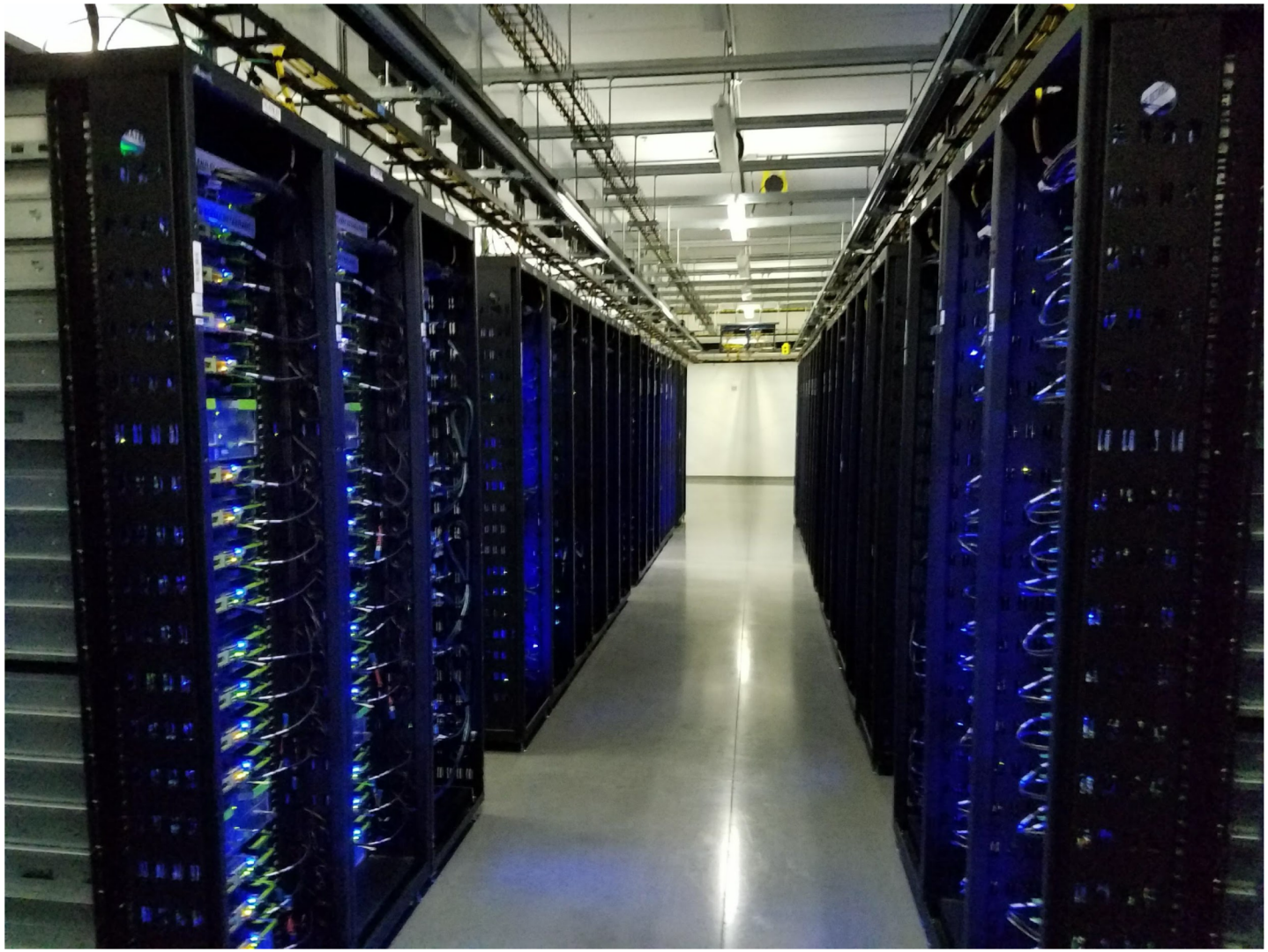
MapReduce



Andrew DeOrio

Agenda

- Distributed systems introduction
  - MapReduce motivation
- MapReduce programming model
- MapReduce execution model
- Example
- Fault tolerance
- Summary

Distributed systems

- Distributed system: Multiple computers cooperating on a task

- MapReduce: Distributed system for compute
  - Run a program that would be too slow on one computer
  - Today
- Google File System: Distributed system for storage
  - Store more data than fits on one computer
  - Next time

Distributed system implementation

- How are distributed systems implemented?
- Networking for communication
  - Coming soon
- Threads and processes for parallelization
  - Coming soon

Datacenter

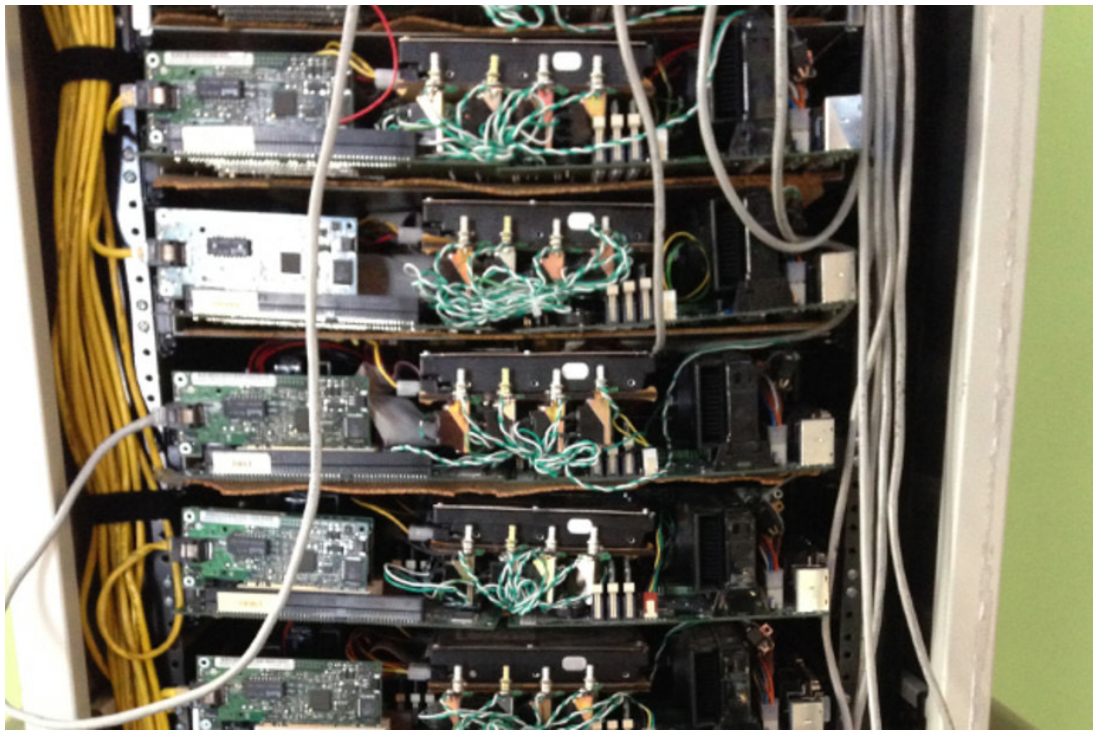
- Where are distributed systems implemented?
- Datacenter: Special building full of computers

- Very loud fan noise
- Hot in some spots and really cold in others
- Unique smell

- Lots of computers means lots of failures

Datacenter computer failures

- Old strategy: buy a few expensive, reliable servers
- New strategy: buy lots of cheap, unreliable servers
  - Use software to make them look reliable
- Modern distributed systems need to solve this problem



Big program examples

- What kinds of programs are too big to run on one computer?

- Count the number of requests to each web page
  - Given several TB of log files
- Build a search engine inverted index ("look up table")
  - Over several PB of HTML files
- Count the frequency of words
  - In several PB of text files
  - Part of search engine inverted index construction

Why learn about distributed systems?

- Distributed systems are the solution for dealing with the scale of the web

- In P4, you will implement a distributed system
  - MapReduce framework

- Goal: See what the challenges are and what solutions look like

Not a distributed system

- Write a program to count the frequency of words using Python

```
$ cat input.txt
Hello World
Bye World
Hello Hadoop
Goodbye Hadoop

$ cat input.txt | python3 wc.py
Hello 2
World 2
Bye 1
Hadoop 2
Goodbye 1
```

Not a distributed system

```
import sys
import collections

word_count = collections.defaultdict(int)

for line in sys.stdin:
    words = line.split()
    for word in words:
        word_count[word] += 1

for word, count in word_count.items():
    print(word + "\t" + count)
```

`defaultdict` automatically initializes values to zero

Time and space complexity

- Now, run your program on the entire web
  - Around 10 Billion pages (wikipedia)
  - Google and Bing do this as part of web search inverted index construction ("look up table")
- Time complexity?
- Space complexity?
- Would it run faster if we had 1000 computers?

Time and space complexity

- Now, run your program on the entire web
  - Around 10 Billion pages (wikipedia)
- Time complexity?
  - O(n), where n is the number of web pages
  - Assume web page length is a constant
- Space complexity?
  - O(m), where m is the number of unique words
- Would it run faster if we had 1000 computers?
  - No, because our program is not parallel

Parallel example

- Split input into two files
- Run same program on two computers

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

```
word_count = collections.defaultdict(int)
for line in sys.stdin:
    words = line.split()
    for word in wors:
        word_count[word] += 1
```

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

```
word_count = collections.defaultdict(int)
for line in sys.stdin:
    words = line.split()
    for word in wors:
        word_count[word] += 1
```

- Problem: Two computers can't share a data structure

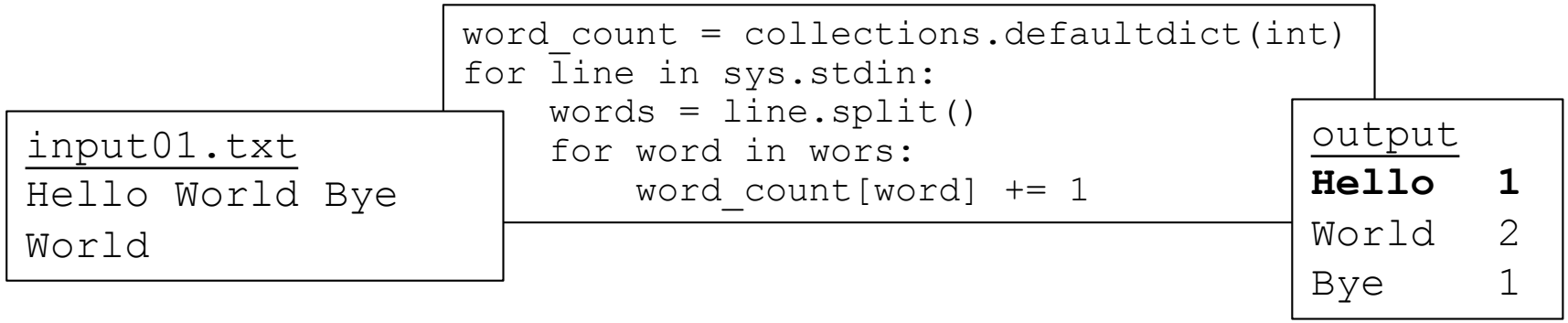
Stateless AKA pure functions

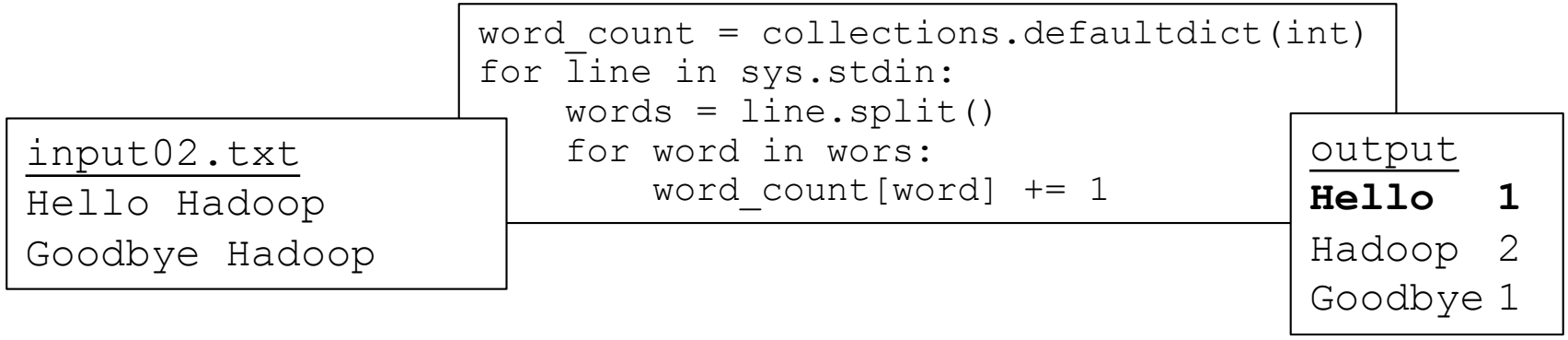
- Problem: Two computers can't share a data structure*
  - One computer can't access the memory of another computer
- Insight: programs that do not share data structures are easier to run in parallel
  - Can't modify any shared data, only read input and produce output
  - Stateless AKA pure function

* A shared data structure could be implemented using a message passing system but it would be slow. The network is a bottle neck.

Parallel example

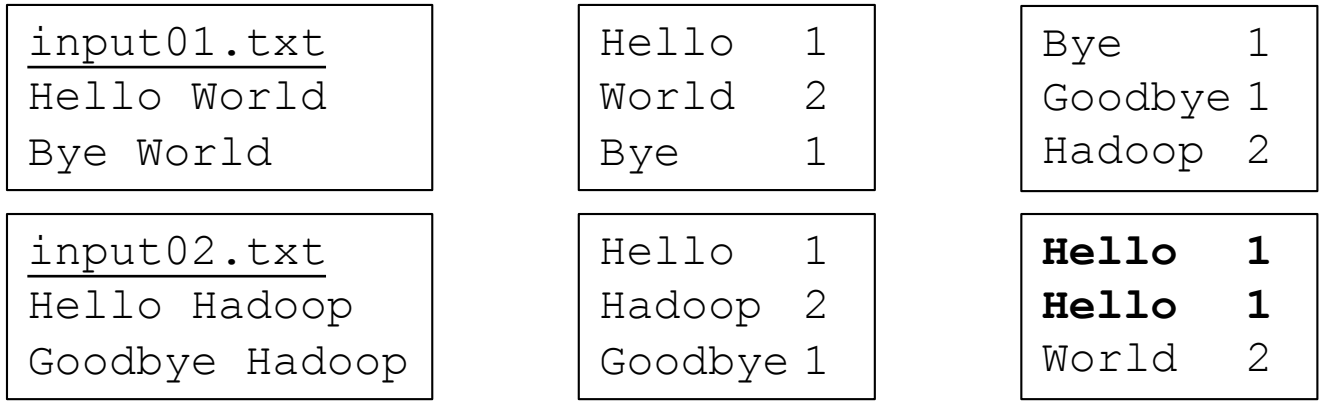
- Two computers, two data structures, two outputs
  - Later, we'll refer to this as *mapping*
- Problem: need to "put together" the output





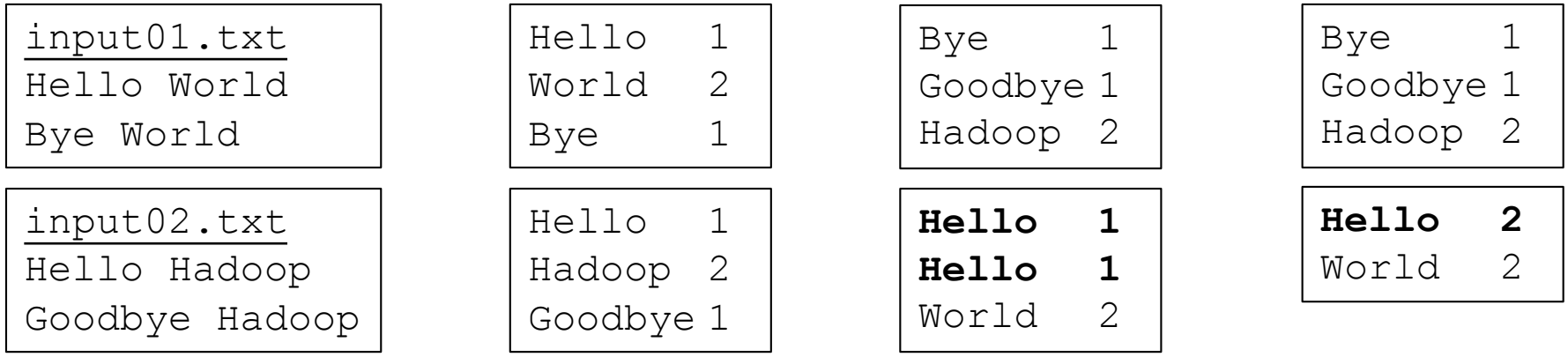
Group

- Insight: Output of first program is in the form <*key*> <*value*>
- Group by key
- Guarantee that every line with the *same key* is in the *same group*



Reduce

- After we have groups, we can "put together the output"
- This is called *reducing*
  - Runs in parallel, just like map
- Write this program



Reduce

- In this example, the second program is very similar to the first
- No shared data structures

```
Bye     1
Goodbye 1
Hadoop  2
```

```
word_count = collections.defaultdict(int)
for line in sys.stdin:
    word, count = line.split()
    word_count[word] += count
for word, count in word_count.items():
    print(word + "\t" + count)
```

```
Bye     1
Goodbye 1
Hadoop  2
```

```
Hello   1
Hello   1
World   2
```

```
word_count = collections.defaultdict(int)
for line in sys.stdin:
    word, count = line.split()
    word_count[word] += count
for word, count in word_count.items():
    print(word + "\t" + count)
```

```
Hello   2
World   2
```

New application: Web access log stats

- Count the number of visits to each page in TBs of server logs
- Solution is a lot like the previous word count example

```
server1-nginx.log
2021-05-10 09:51 141.213.74.63 index.html
2021-05-10 09:52 35.7.44.69 index.html
```

```
server2-nginx.log
2021-05-10 10:05 141.213.74.63 syllabus.html
2021-05-10 11:19 35.7.44.69 p1-insta485-static.html
```

New application and old application

- Old application: word count
- New application: web access log stats

- Some code will be identical
  - The communication and coordination between machines

- More problems
  - Don't know how many machines until run time
  - What happens if one machine dies during a long running computation? We'd have to start over!

- We'd have to solve all the same parallelization problems again!
  - Enter MapReduce

MapReduce

- MapReduce Framework handles
  - Parallelization over many machines
  - Segment and distribute input
  - Fault tolerance
- MapReduce provides a clean abstraction for programmers
- Programmer writes two programs
  - Map
  - Reduce

Agenda

- Distributed systems introduction
  - MapReduce motivation
- **MapReduce programming model**
- MapReduce execution model
- Example
- Fault tolerance
- Summary

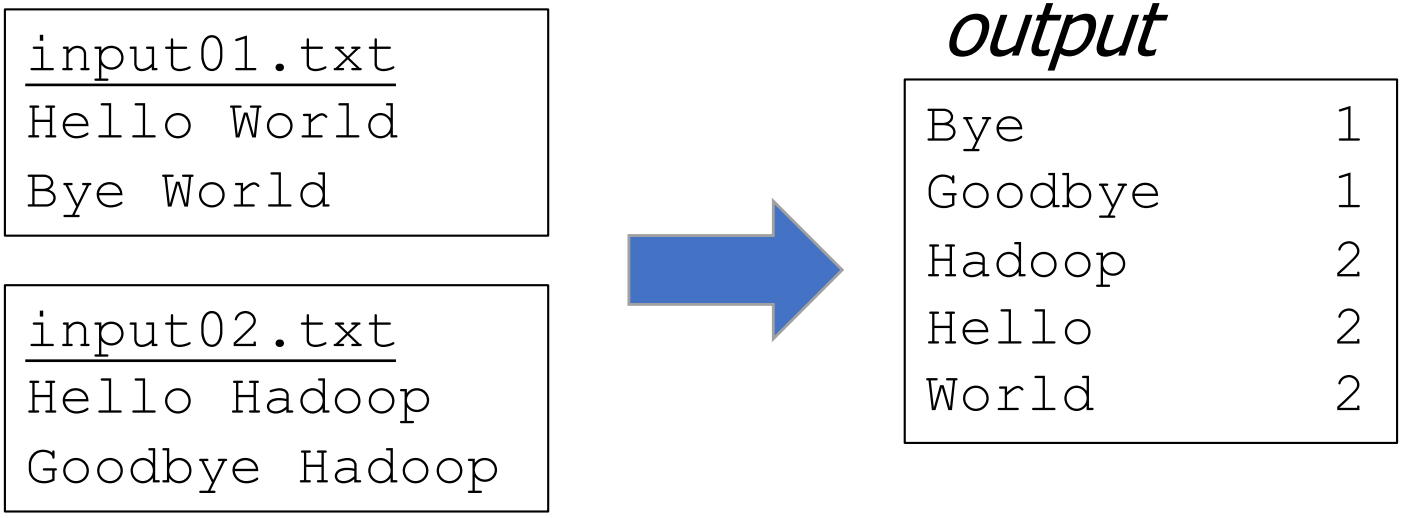
MapReduce programming model

- Our examples will use the Hadoop MapReduce streaming interface

- Map is a program
  - Each line of input is from an input file
  - Each line of output is `<key>\t<value>`
- Group is provided by MapReduce framework
- Reduce is a program
  - Each line of input is `<key>\t<value>`
  - Each line of output is up to the programmer

A second look at word count

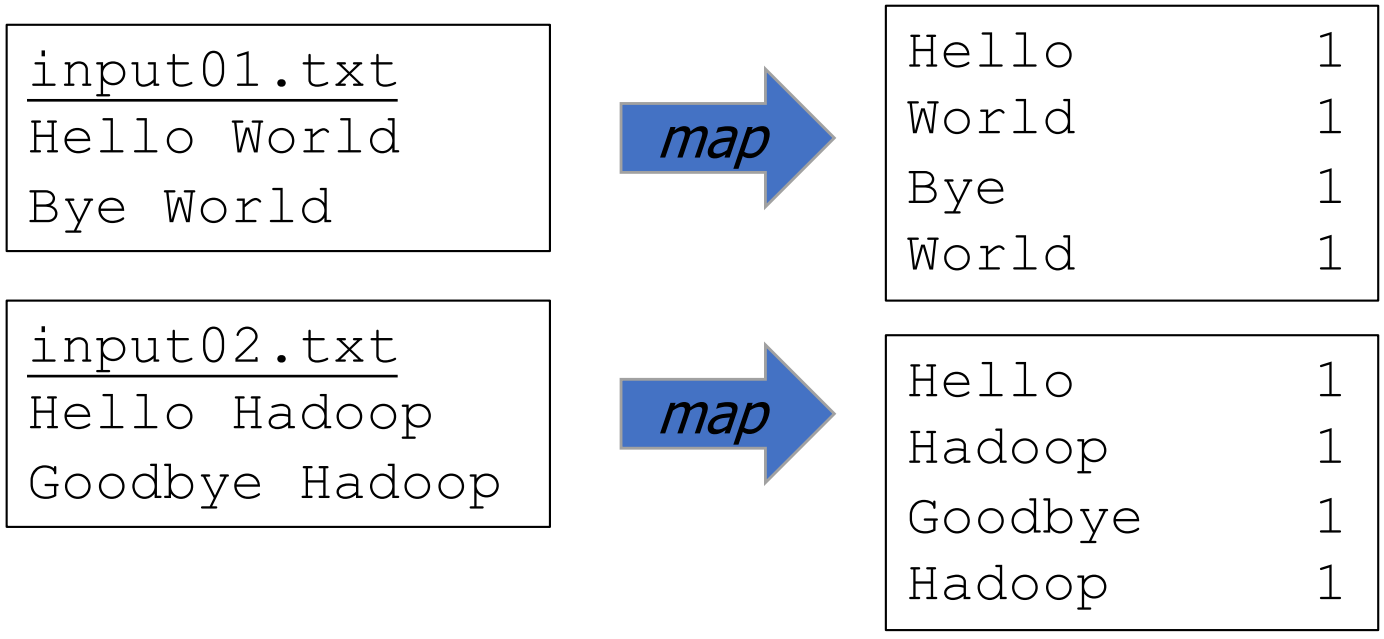
- Count the number of occurrences of each word in a collection of documents
- We'll solve the same problem in a slightly different way



Word count map

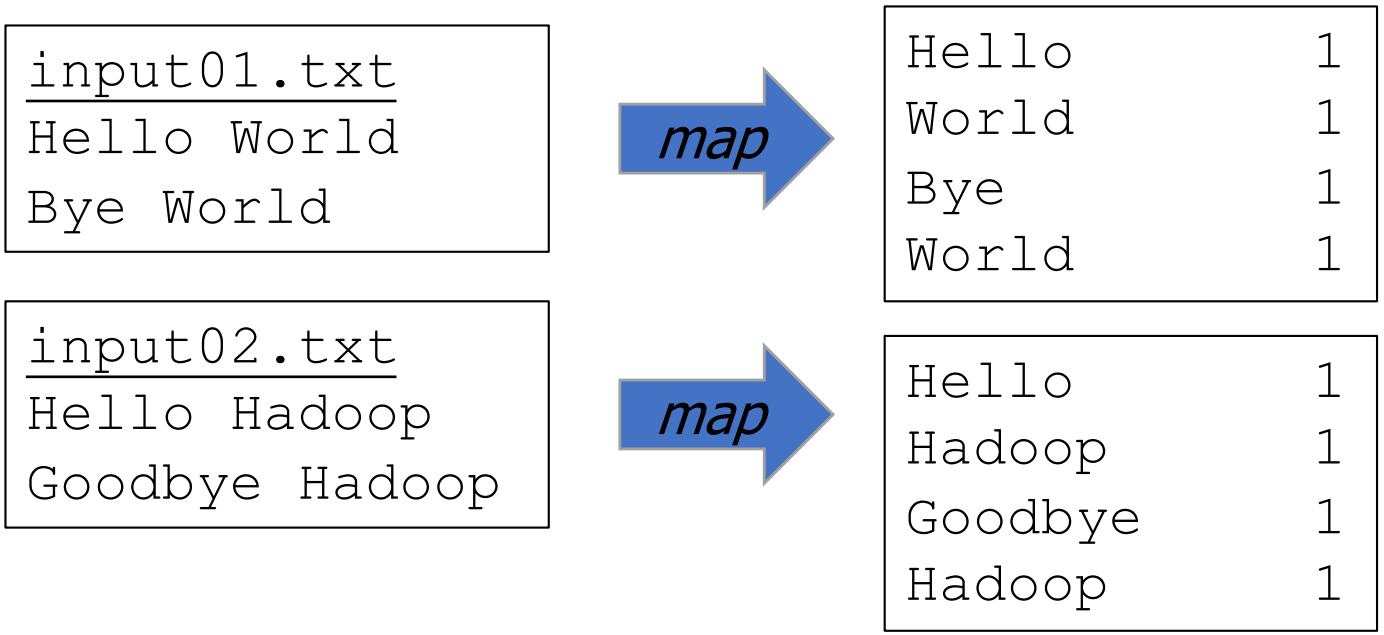
```
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print(word + "\t" + "1")
```

- Mappers don't count words, it just splits them up



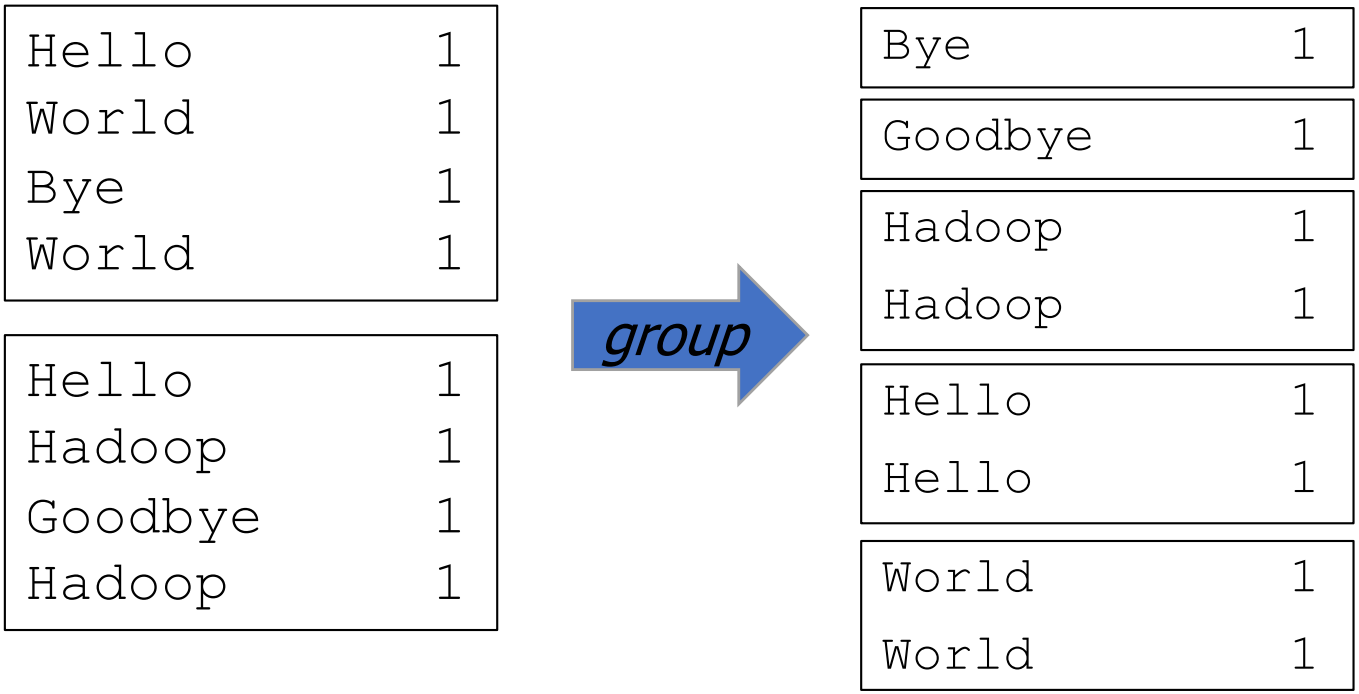
Intermediate key value pairs

- Map output is in the form of *key value pairs*
- Separated by a TAB in the Hadoop streaming interface
- What to put here is up to the programmer



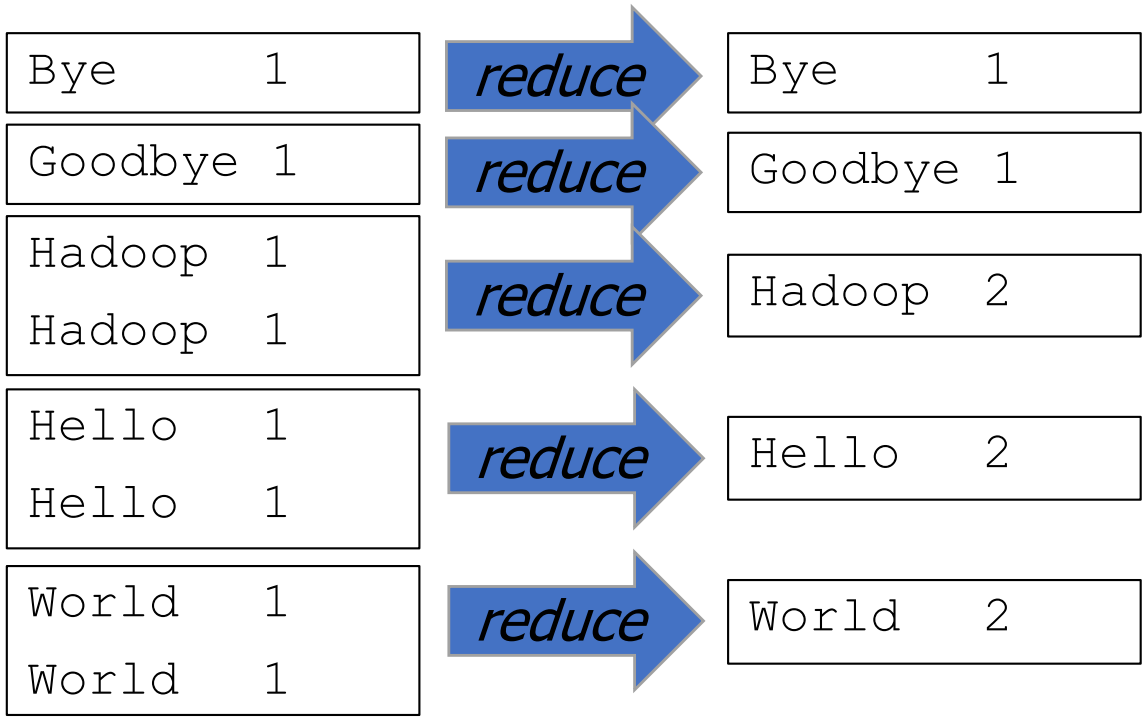
Word count group

- Group is provided by the MapReduce framework
- Group by key



Word count reduce

```
counts = collections.defaultdict(int)

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

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



Intermediate data size

- Isn't the intermediate data size large?
- Option 1: a "local reducer" called a Combiner at each map
  - Compresses data between map and reduce
- Option 2: a different map implementation

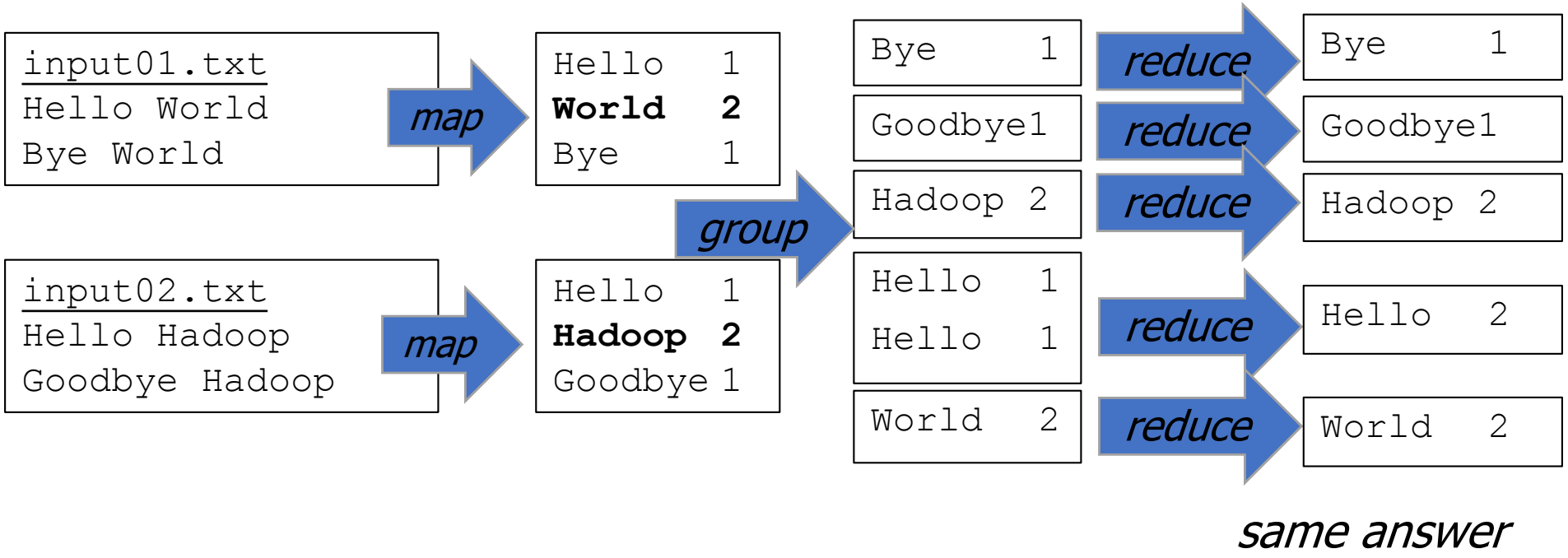
Word count

- Option 2: map sums the words in each doc
  - This is our previous implementation

```
import collections
import sys

word_count = collections.defaultdict(int)

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
    word_count[word] += 1

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

Word count

```
word_count = collections.defaultdict(int)

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
    word_count[word] += 1

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



same answer

Agenda

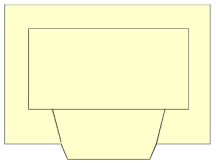
- Distributed systems introduction
  - MapReduce motivation
- MapReduce programming model
- **MapReduce execution model**
- Example
- Fault tolerance
- Summary

MapReduce execution model

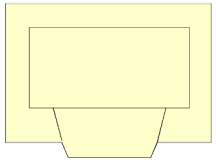
- Segment input
- Map stage
- Group stage
- Reduce stage

Job processing

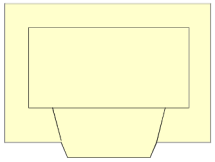
Main



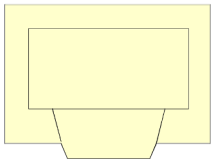
Worker 0



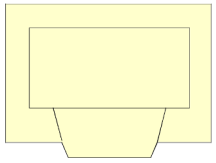
Worker 1



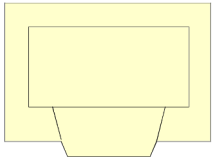
Worker 2



Worker 3

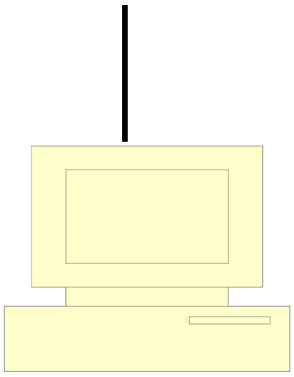


Worker 4



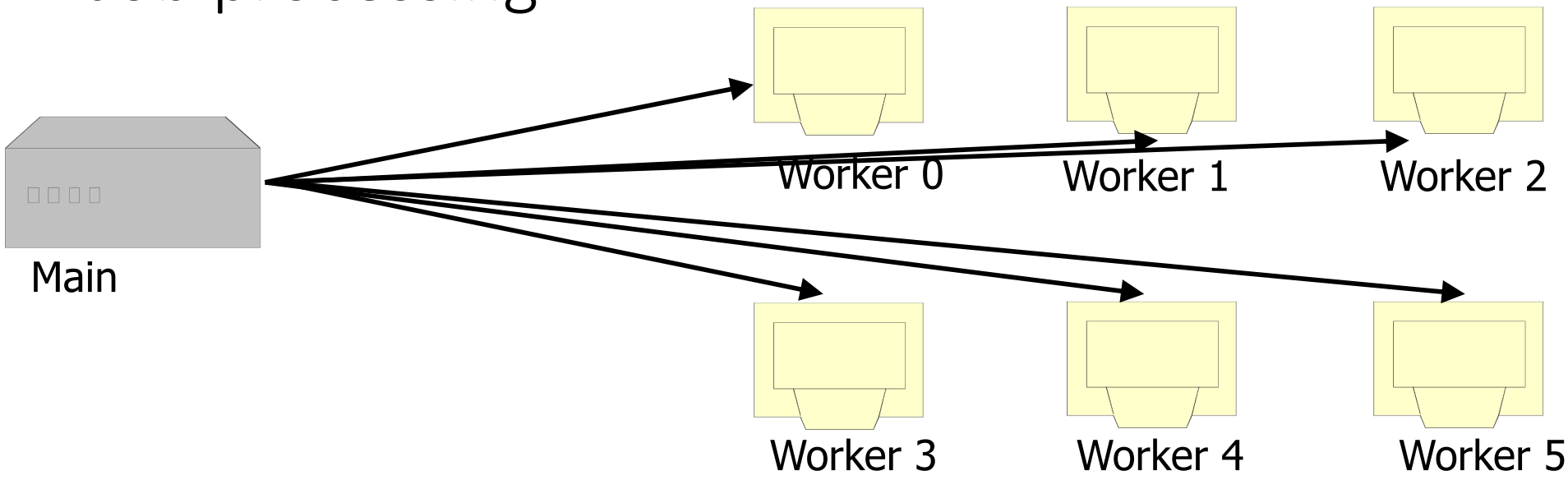
Worker 5

```
{
   "mapper": "wc_map.sh",
   "reducer": "wc_reduce.sh"
}
```

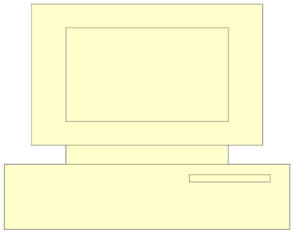


1. Client submits word count job, indicating map code, reduce code and input files

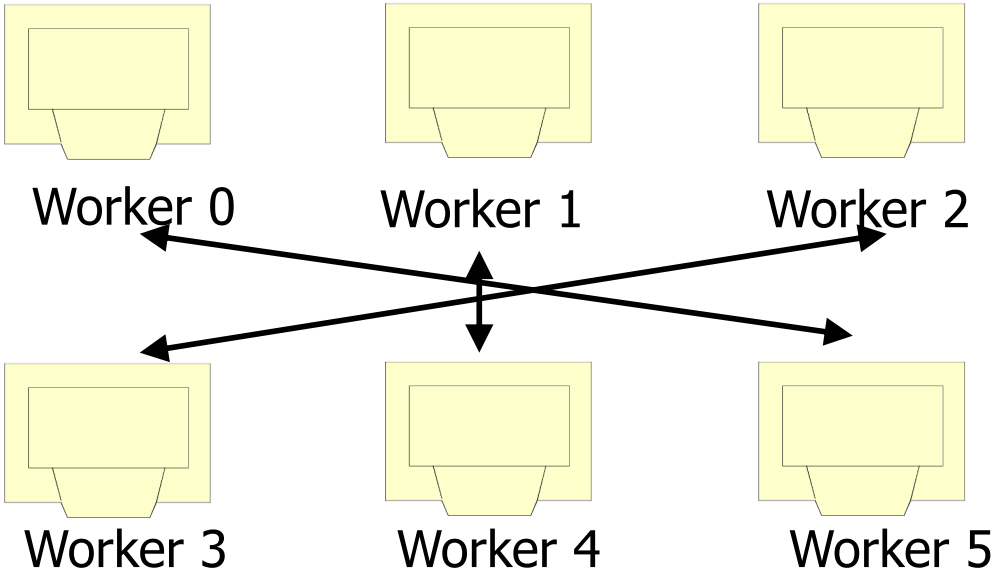
Job processing



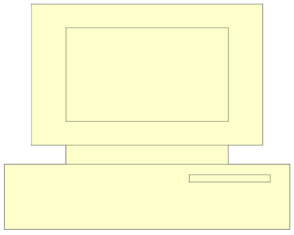
1. Client submits word count job, indicating map code, reduce code and input files
2. Main breaks input file into *k* chunks, (in this case 6). Assigns work to workers.



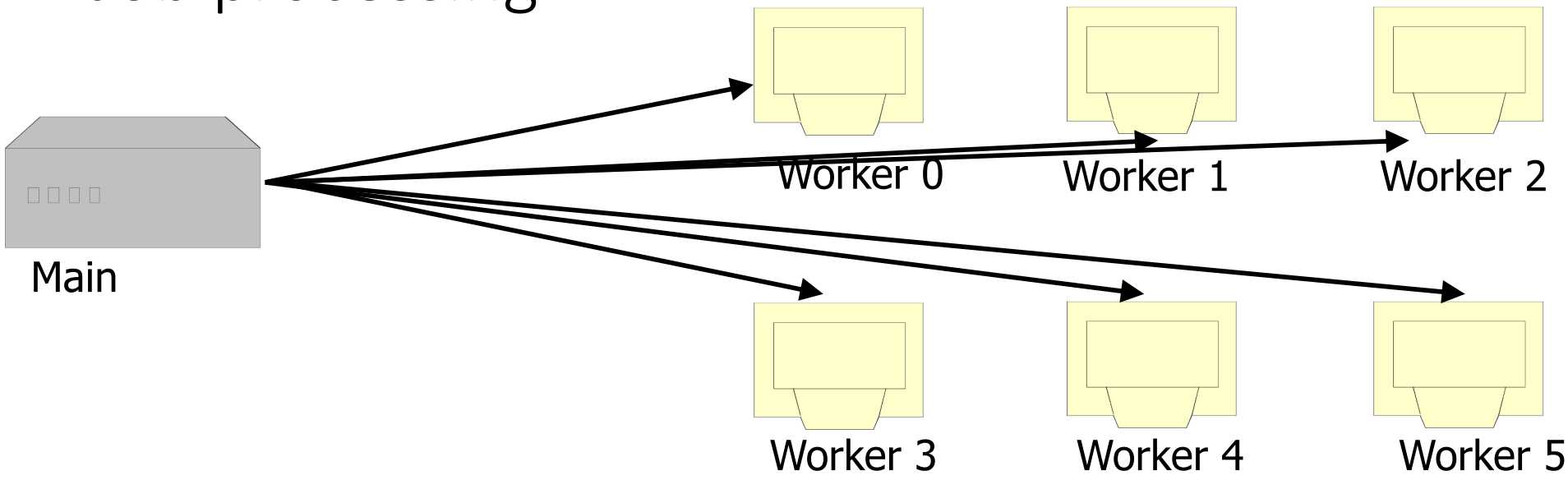
Job processing



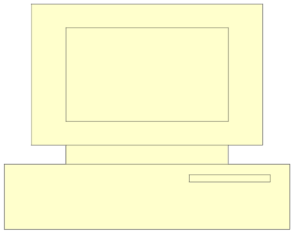
1. Client submits word count job, indicating map code, reduce code and input files
2. Main breaks input file into *k* chunks, (in this case 6). Assigns work to workers.
3. After map(), workers exchange map-output to produce groups



Job processing

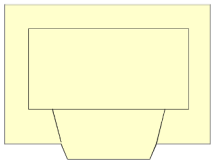


1. Client submits word count job, indicating map code, reduce code and input files
2. Main breaks input file into *k* chunks, (in this case 6). Assigns work to workers.
3. After map(), workers exchange map-output to produce groups
4. Main breaks reduce() keyspace into *m* chunks (in this case 6). Assigns work.

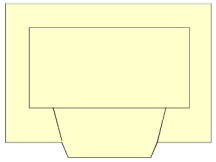


Job processing

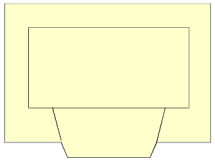
Main



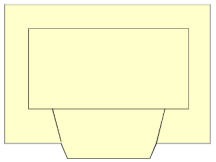
Worker 0



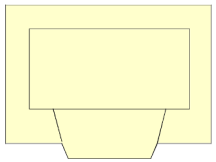
Worker 1



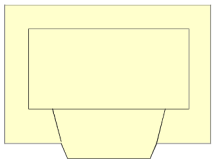
Worker 2



Worker 3

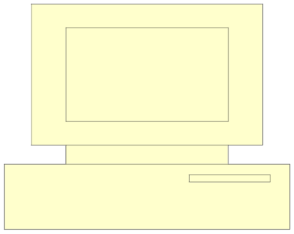


Worker 4



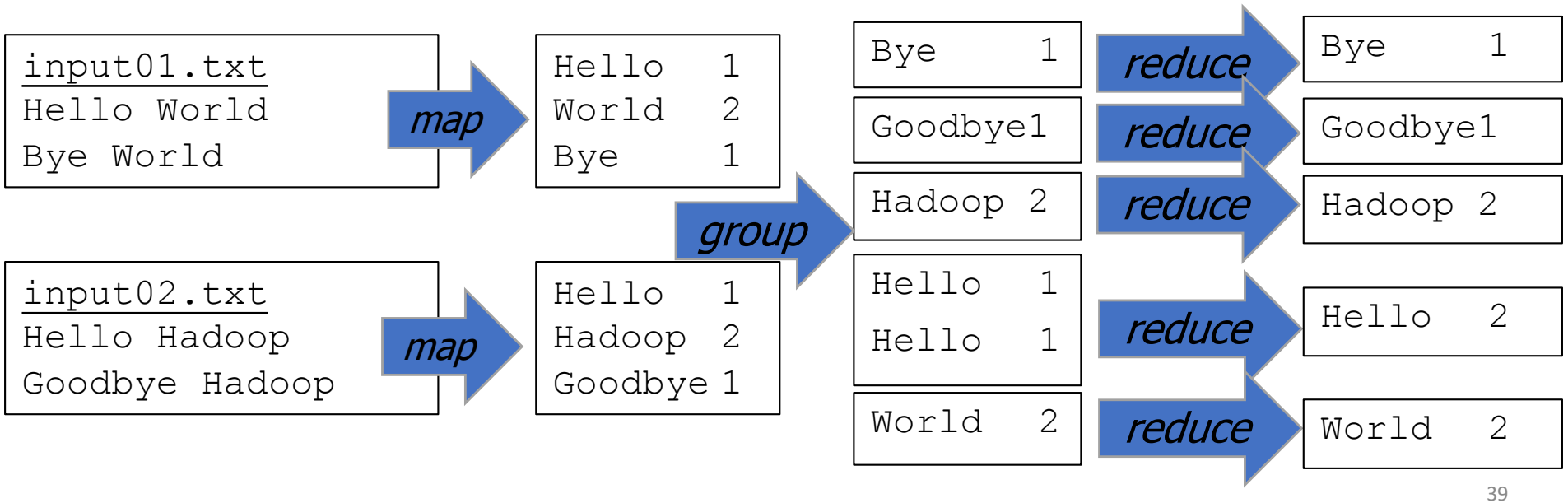
Worker 5

1. Client submits word count job, indicating map code, reduce code and input files
2. Main breaks input file into *k* chunks, (in this case 6). Assigns work to workers.
3. After map(), workers exchange map-output to produce groups
4. Main breaks reduce() keyspace into *m* chunks (in this case 6). Assigns work.
5. reduce() output may go to shared fs



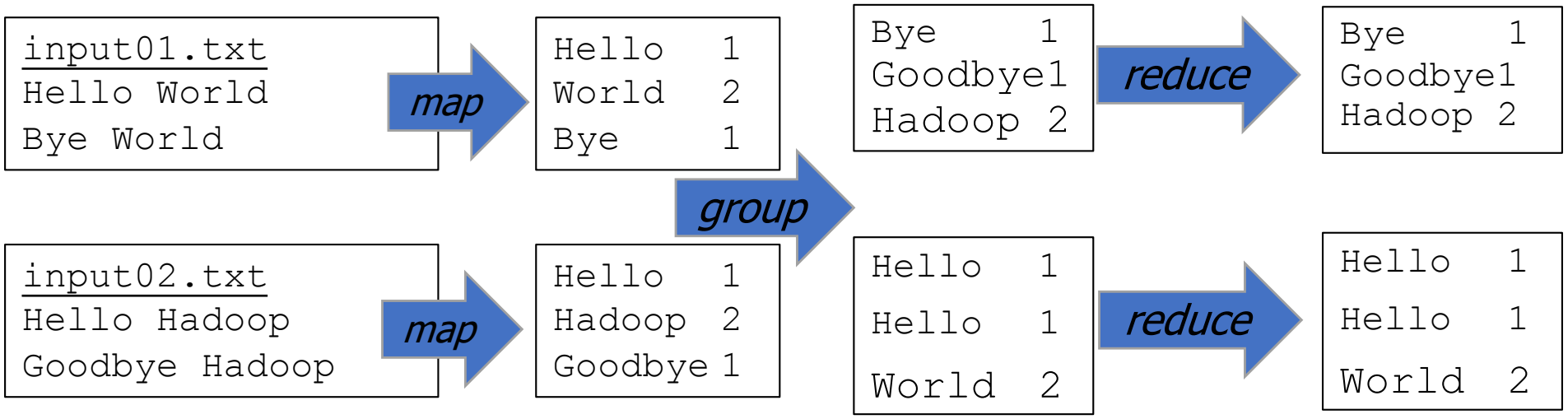
Same key, same group

- MapReduce framework provides grouping functionality
- All values for the **same key** *must* be in the **same group**



Same key, same group

- Could a group have more than 1 key? Yes.
  - In the Hadoop streaming interface



Project 4

- You'll write a MapReduce framework in EECS 485 Project 4

- Segment input
- Map stage
- Group stage
- Reduce stage

- We'll provide the MapReduce programs

Agenda

- Distributed systems introduction
  - MapReduce motivation
- MapReduce programming model
- MapReduce execution model
- **Example**
- Fault tolerance
- Summary

Grep

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

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

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

```
1 Hello World
```

```
1 Hello Hadoop
```

Grep

- Map

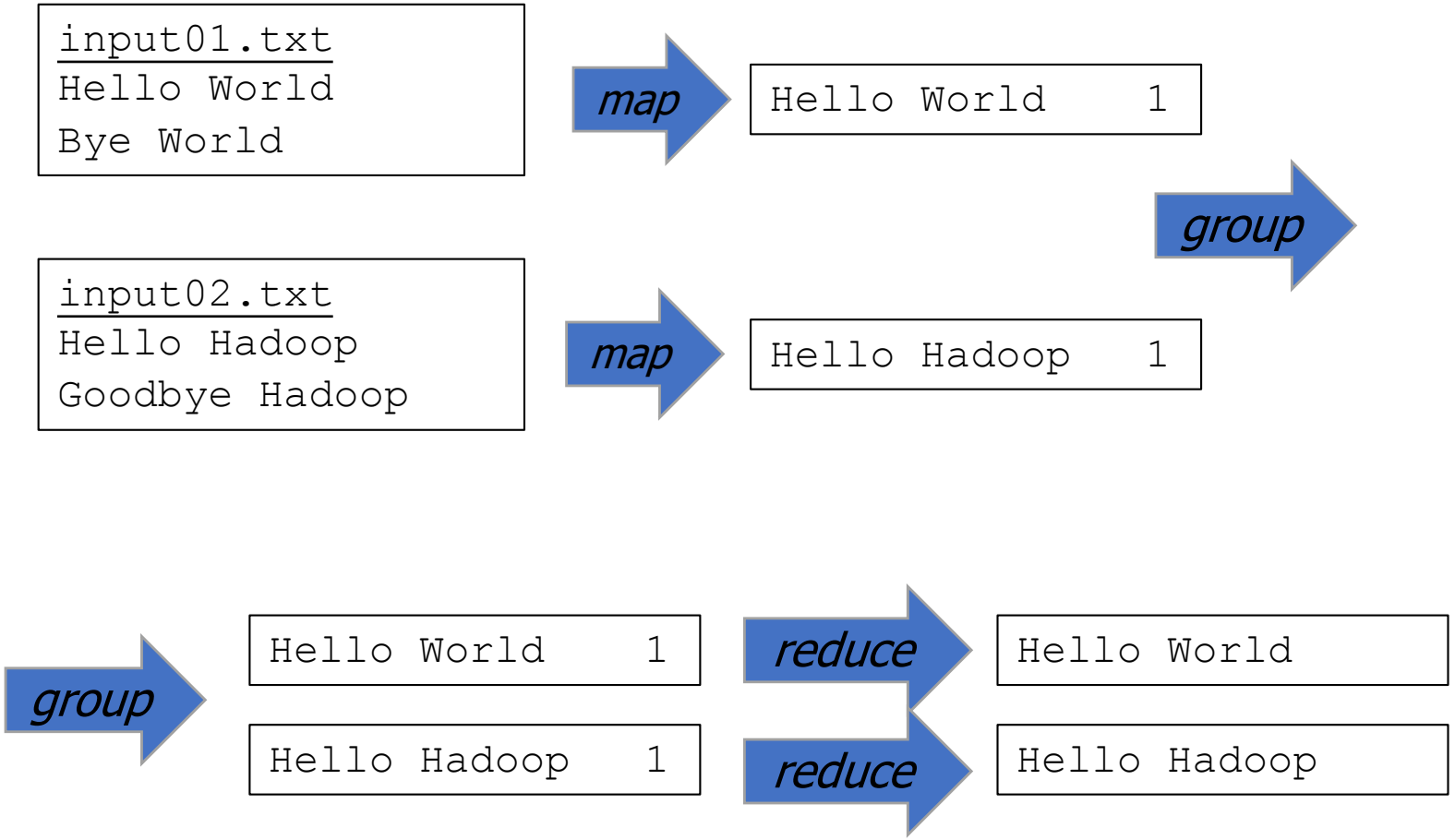
```
import sys
for line in sys.stdin:
    line = line.strip()
    if "Hello" in line:
        print(line + "\t" + "1")
```

- Reduce

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

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

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



Agenda

- Distributed systems introduction
  - MapReduce motivation
- MapReduce programming model
- MapReduce execution model
- Example
- **Fault tolerance**
- Summary

Fault tolerance

- How do we know if a machine goes down?

- Workers send periodic heartbeat messages to Main
- Main 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

Fault tolerance nice side effects

- What about slow, not dead, machines?
- Speculative execution for stragglers
- Kill the 2nd-place finisher

Agenda

- Distributed systems introduction
  - MapReduce motivation
- MapReduce programming model
- MapReduce execution model
- Example
- Fault tolerance
- **Summary**

Key observations

- Scalability and fault-tolerance achieved by optimizing the execution engine once
  - Use it many times by writing different map and reduce functions for different applications
- Stateless mapper
- Stateless reducer

Key observations

- Map and reduce functions inspired by functions of the same name in Lisp programming language
- Functional programming
  - Computation as the evaluation of mathematical functions
- Functions have no side effects
  - AKA "pure" functions
  - AKA stateless
  - Does not change state outside itself
- Easy to parallelize!

Implementations

- Examples of MapReduce implementations
  - The framework that executes MapReduce programs

- Apache Hadoop
- Disco Project
- Riak
- EECS 485 Project 4

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