

# MapReduce case study



جامعة الملك عبد الله  
للعلوم والتقنية  
King Abdullah University of  
Science and Technology

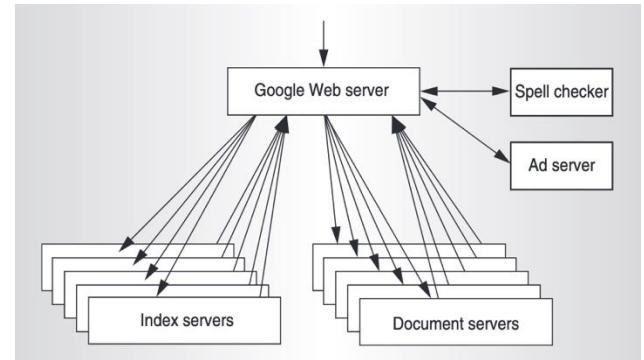
---

CS 240: *Computing Systems and Concurrency*  
Lecture 1.1

Marco Canini

# Why scalable analytics?

- Google's index was perhaps one of the first "big data" problems
  - Crawler fetched 100s of millions of web pages
  - Needed to create giant indices from keywords
  - Too much work for any individual machine  
→ needed to be spread across many machines
- Soon they also needed to compute various statistics on this data
  - For instance, how many documents contained a given keyword?
- This led to the development of the MapReduce framework



# Key challenges

---

- Data is spread across (many) computers
  - What do we do if related data is on different computers, but we need all of it to perform some computation?
- Communication is expensive
  - Need to be smart about where data is stored, and when it is moved
- Coordination is key
  - The computation needs to be carefully orchestrated to get the correct result
  - ... especially if there are failures, heterogeneous machines, etc.

# Case Study: MapReduce

(Data-parallel programming at scale)

# What is MapReduce?

---

- MapReduce is a famous distributed programming model
  - Invented at Google; paper published in 2004
  - At that time, it was used for the production indexing system
- Closed source, but open-source reimplementations exist
  - Example: Apache Hadoop
- Originally ran on GFS (The Google FileSystem)
  - GFS is designed for sequential reads and appends
  - This is the workload that MapReduce would produce!

# Application: Word Count

---

```
SELECT count(word) FROM data  
GROUP BY word
```

```
cat data.txt
```

```
| tr -s '[:punct:][:space:]' '\n'
```

```
| sort | uniq -c
```

# Using partial aggregation

---

1. Compute word counts from individual files
2. Then merge intermediate output
3. Compute word count on merged outputs

# Using partial aggregation: data flow

---

1. In parallel, send to worker:
  - Compute word counts from individual files
  - Collect result, wait until all finished
2. Then merge intermediate output
3. Compute word count on merged intermediates



# I don't want to deal with all this!

---

- Wouldn't it be nice if there were some system that took care of all these details for you?
  - But every task is different!
  - Or is it? The detailed are different (what to compute, etc.), but the **data flow** is often same!
  - Maybe we can have a 'generic' solution?
- Ideally, just tell the system what needs to be done
- This is what frameworks like MapReduce (and Apache Spark and Apache Flink) do!

# MapReduce: Programming Interface

---

`map(key, value) -> list(<k', v'>)`

- Apply function to (key, value) pair and produces set of intermediate pairs

`reduce(key, list<value>) -> <k', v'>`

- Applies aggregation function to values collected by key
- Outputs result

# MapReduce example: Word Count

---

```
map(String key, String value):  
    // key: document name; value: document line  
    for each word w in value:  
        EmitIntermediate(w, "1");
```

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

# MapReduce: Optimizations

---

`combine(list<key, value>) -> list<k, v>`

- Perform partial aggregation on mapper node:

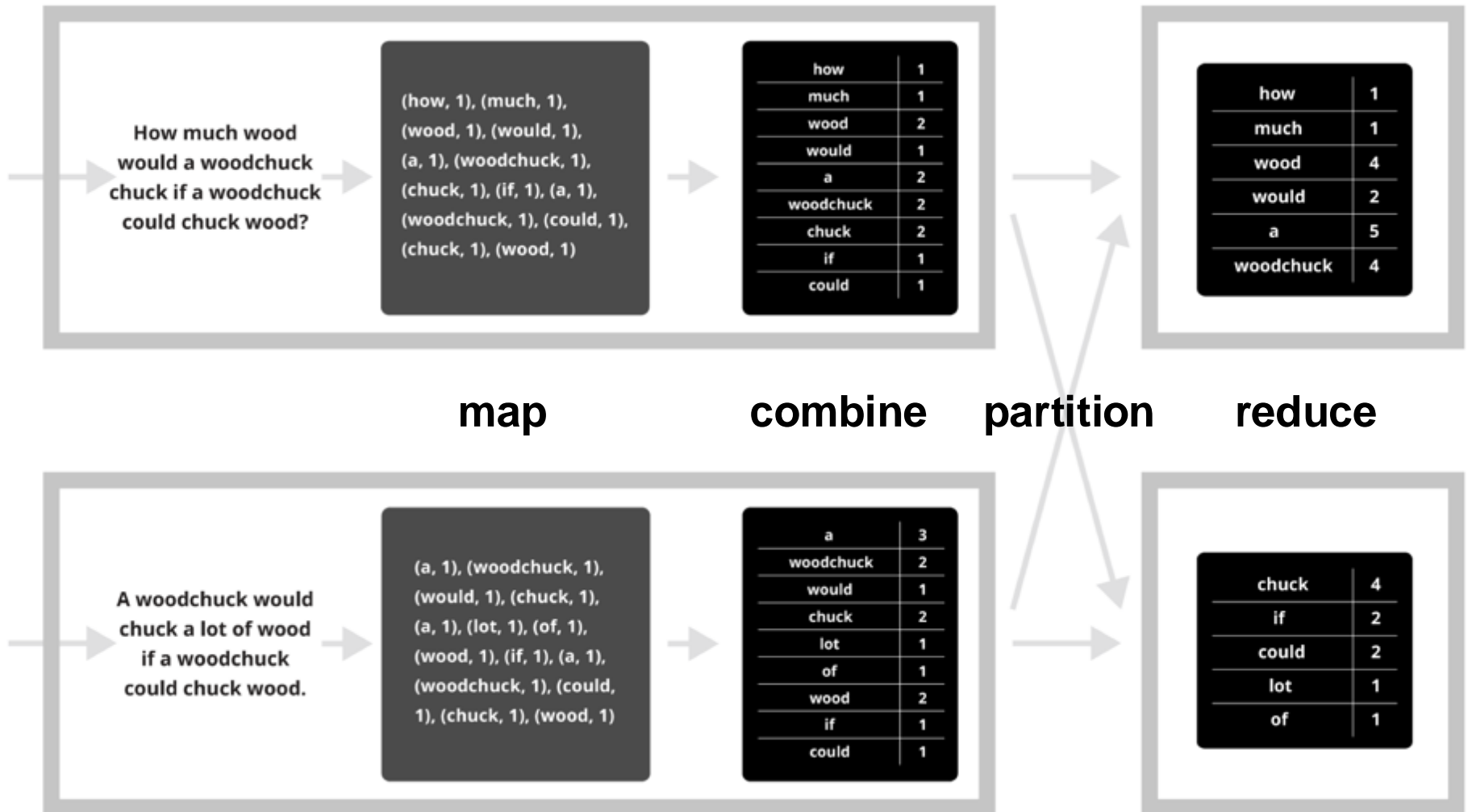
`<the, 1>, <the, 1>, <the, 1> → <the, 3>`

- `combine()` should be commutative and associative

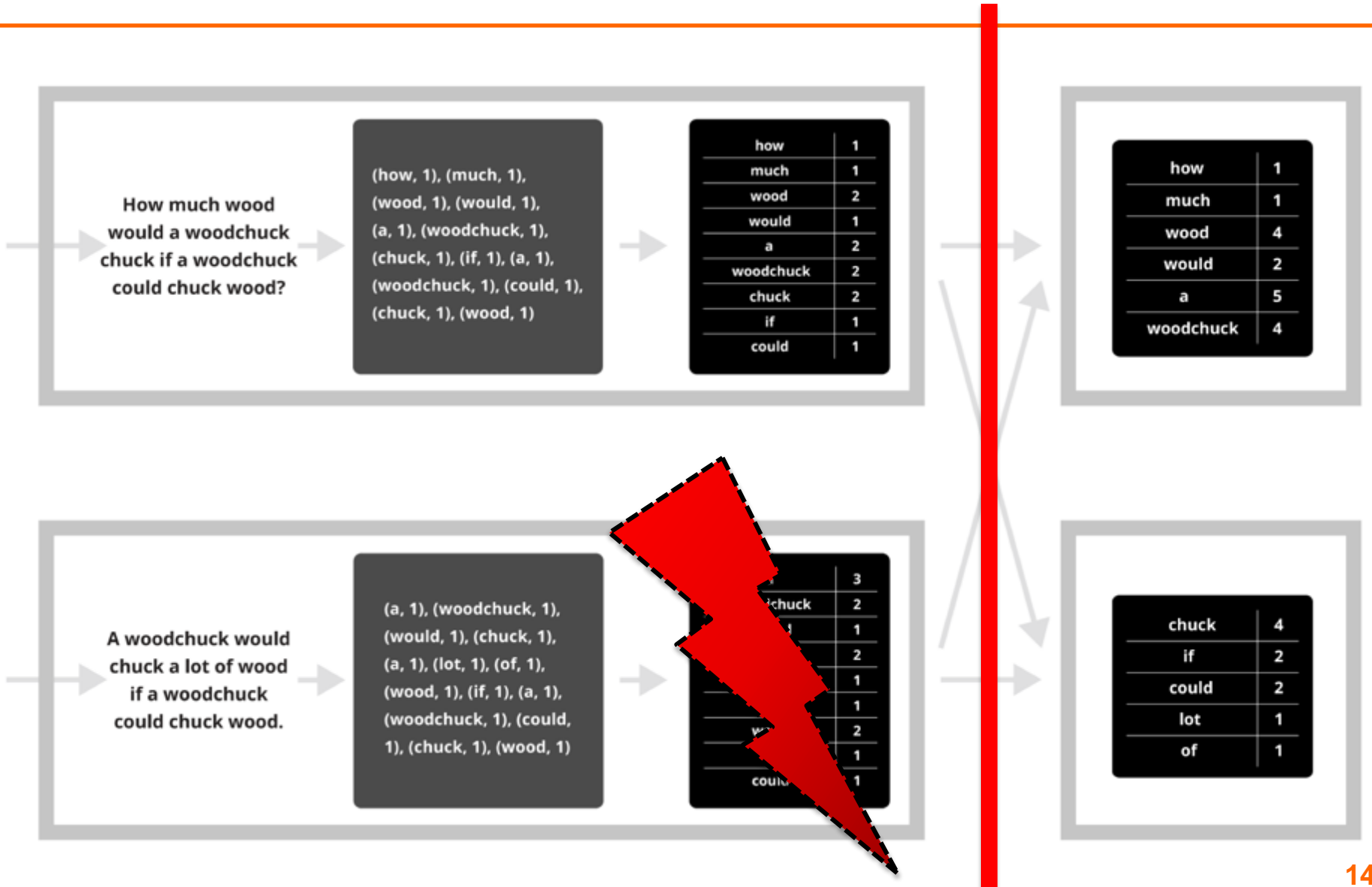
`partition(key, int) -> int`

- Need to aggregate intermediate vals with same key
- Given  $n$  partitions, map key to partition  $0 \leq i < n$
- Typically via `hash(key) mod n`

# Putting it together...

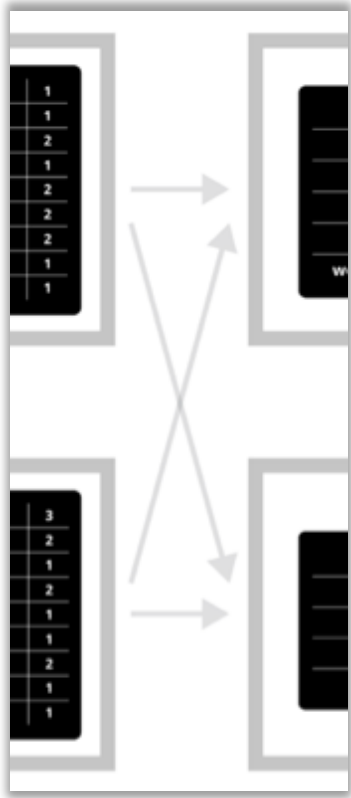


# Synchronization Barrier



# Fault Tolerance in MapReduce

---



- Map worker writes intermediate output to local disk, separated by partitioning. Once completed, tells master node.
- Reduce worker told of location of map task outputs, pulls their partition's data from each mapper, execute function across data
- Note:
  - “All-to-all” shuffle b/w mappers and reducers
  - Written to disk (“materialized”) b/w *each* stage

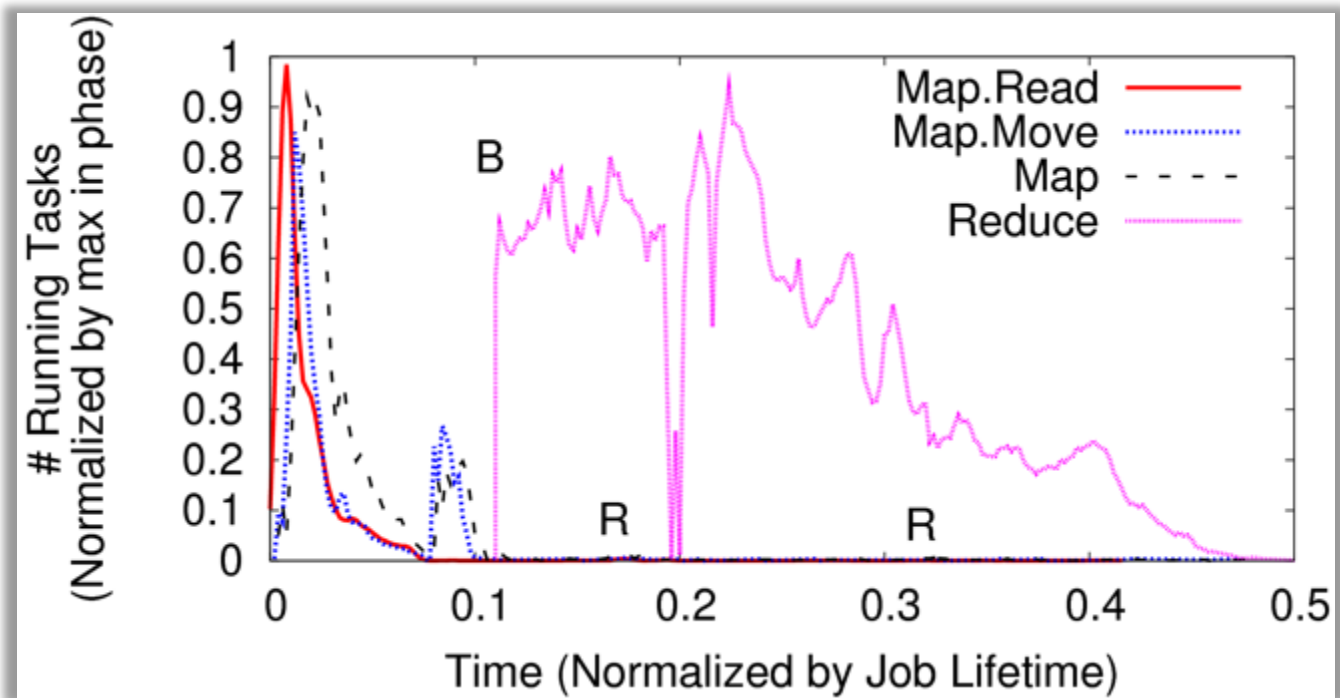
# Fault Tolerance in MapReduce

---

- Master node monitors state of system
  - If master failures, job aborts and client notified
- Map worker failure
  - Both in-progress/completed tasks marked as idle
  - Reduce workers notified when map task is re-executed on another map worker
- Reducer worker failure
  - In-progress tasks are reset to idle (and re-executed)
  - Completed tasks had been written to global file system



# Straggler Mitigation in MapReduce



- Tail latency means some workers finish late
- For slow map tasks, execute in parallel on second map worker as “backup”, race to complete task

# MapReduce: Limitations

---

- MapReduce worked very well for Google's initial use cases, and lots of others besides
  - No data dependencies within map/reduce phases → Good scalability
- But it does have some important limitations:
  - Complex operations have to be rewritten into 'map' and 'reduce' operations (possibly with several rounds of mapping and reducing)
  - Dataflows always read from and write to disk (why?) → limited speed

# You'll build (simplified) MapReduce!

---

- Assignment 1: Sequential MapReduce
  - Learn to program in Go!
- Assignment 2: Distributed MapReduce
  - Learn Go's concurrency, network I/O, and RPCs