Big Data Processing

COS 418: Distributed Systems
Lecture 19

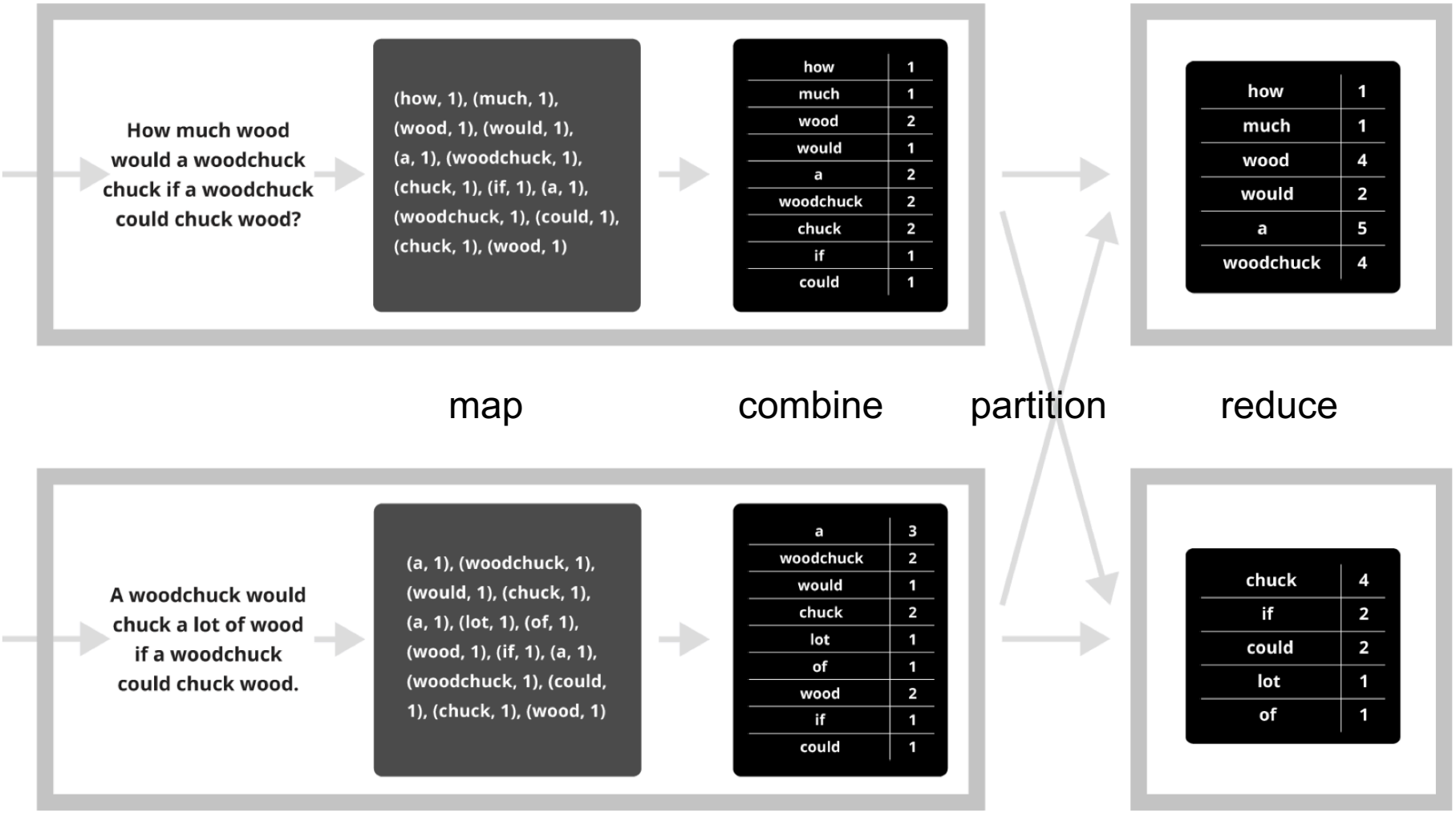
Wyatt Lloyd

Map Reduce Review

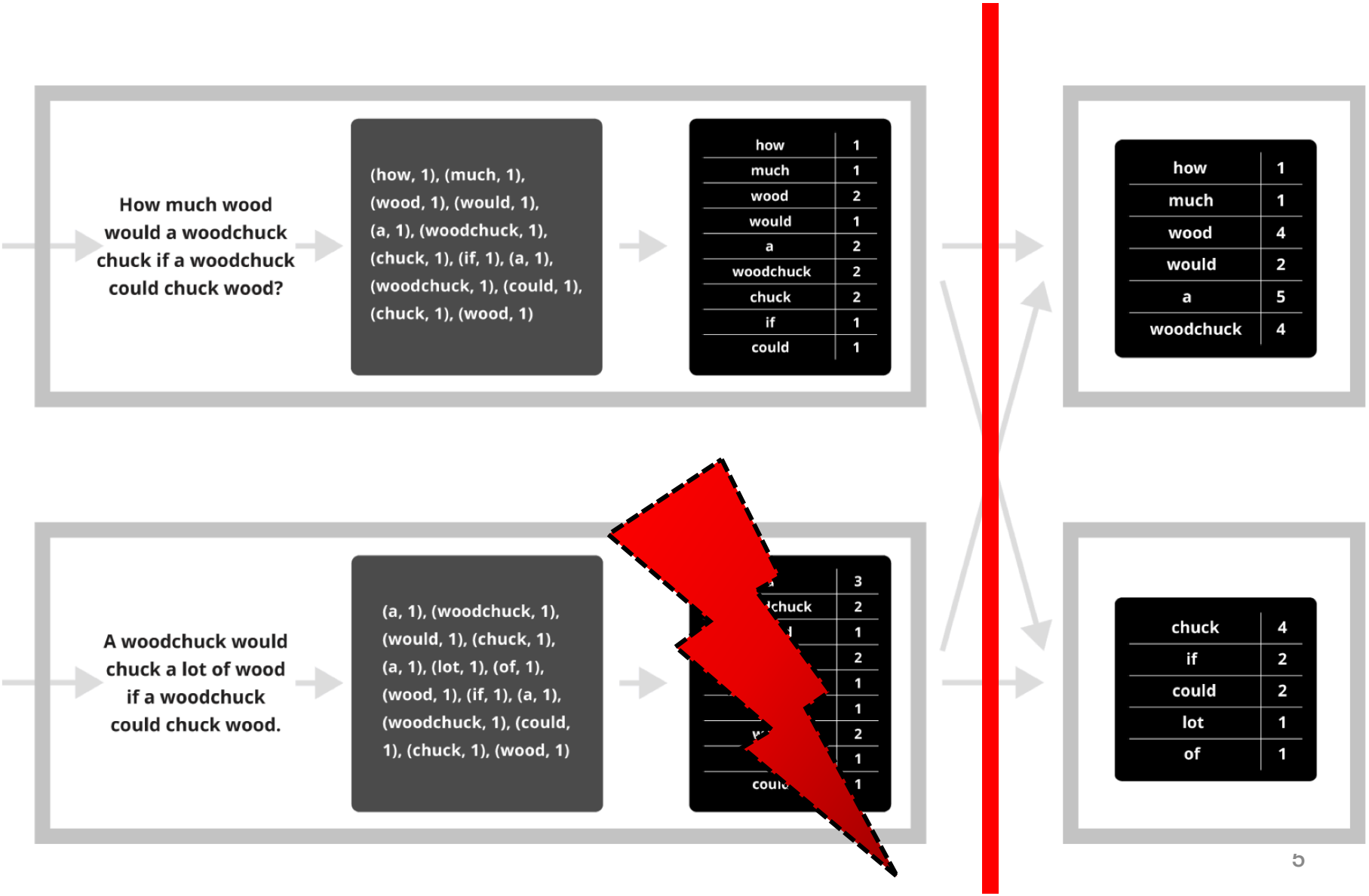
Ex: Word count using partial aggregation

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

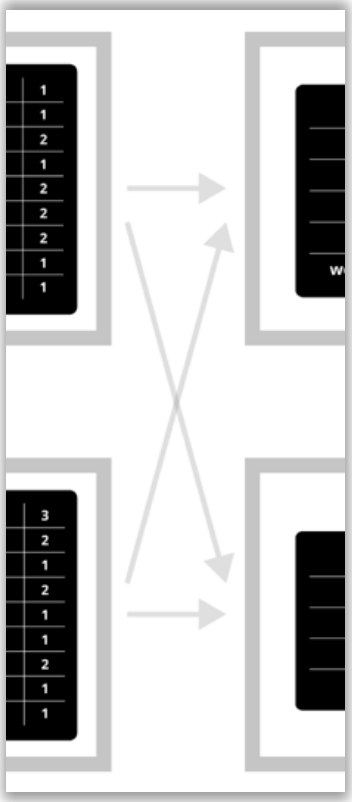
Putting it together…



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

Generality vs Specialization

General Systems

- Can be used for many different applications
- Jack of all trades, master of none
  - Pay a generality penalty
- Once a specific application, or class of applications becomes sufficiently important, time to build specialized systems

MapReduce is a General System

- Can express large computations on large data; enables fault tolerant, parallel computation
- Fault tolerance is an inefficient fit for many applications
- Parallel programming model (map, reduce) within synchronous rounds is an inefficient fit for many applications

MapReduce for Google's Index

- Flagship application in original MapReduce paper

- *Q: What is inefficient about MapReduce for computing web indexes?*
  - "MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency."

- Index moved to Percolator in ~2010 [OSDI '10]
  - Incrementally process updates to index
  - Uses OCC to apply updates
  - 50% reduction in average age of documents

MapReduce for Iterative Computations

- Iterative computations: compute on the same data as we update it
  - e.g., PageRank
  - e.g., Logistic regression

- *Q: What is inefficient about MapReduce for these?*
  - Writing data to disk between all iterations is slow

- Many systems designed for iterative computations, most notable is Apache Spark
  - Key idea 1: Keep data in memory once loaded
  - Key idea 2: Provide fault tolerance via *lineage*:
    - Save data to disks occasionally, remember computation that created later version of data. Use lineage to recompute data that is lost due to failure.

MapReduce for Stream Processing

- Stream processing: Continuously process an infinite stream of incoming events
  - e.g., estimating traffic conditions from GPS data
  - e.g., identify trending hashtags on twitter
  - e.g., detect fraudulent ad-clicks

- *Q: What is inefficient about MapReduce for these?*

Stream Processing Systems

- Many stream processing systems as well, typical structure:
  - Definite computation ahead of time
  - Setup machines to run specific parts of computation and pass data around (topology)
  - Stream data into topology
  - Repeat forever
  - Trickiest part: fault tolerance!

- Notably systems and their fault tolerance
  - Apache/Twitter Storm: Record acknowledgment
  - Spark Streaming: Micro-batches
  - Google Cloud dataflow: transactional updates
  - Apache Flink: Distributed snapshot

- Specialization is much faster, e.g., click-fraud detection at Microsoft
  - Batch-processing system: 6 hours
  - w/ StreamScope[NSDI '16]: 20 minute average

MapReduce for Machine Learning

- Machine learning training often iteratively updates parameters of a model until it converges
  - All workers need to know models of the parameter
- General iterative systems like Spark still slow because they coalesce and then broadcast parameter updates to all workers
- Specialize even further for ML!
  - Many such systems, e.g., TensorFlow [OSDI '16]
- Think ML+Systems is interesting?
  - COS 598G offered in the spring

DeepDive on Distributed Video Processing at Facebook