Distributed Computing

A-13. MapReduce

Big Data

- What does this mean?
  - Web
  - Physics/Astronomy/Finance data…
- 3 Vs:
  - Volume
  - Velocity
  - Variety
- Realization: **data** often **counts more** than the algorithms used

The MapReduce Programming Model

- A programming model inspired by
  - Functional programming
  - Bulk Synchronous Parallelism (BSP)
- Execution frameworks
  - For large-scale data processing
  - Designed to run on "commodity hardware"
    - i.e., medium-range servers

References

- *MapReduce: Simplified Data Processing on Large Clusters*—see paper by Dean and Ghemawat (Google) at USENIX OSDI '04
- The lesson in the course by Pietro Michiardi (EURECOM); and figures thanks to him. Thanks!
- Data-Intensive Text Processing with MapReduce, book by Jimmy Lin and Chris Dyer

Principles

Scale Out, Not Up

- Many “commodity servers” are preferable to few high-end ones
  - Cost grows more than linearly with performance
- In some cases, a big enough server just doesn’t exist
  - Google: estimated at 15 Exabytes in 2013 (15,000,000,000,000,000,000 bytes)
  - Internet Archive: 200 PetaBytes (2021)

Looking at the Bottlenecks

- In many workloads, **disk I/O is the bottleneck**
  - Reading from a HDD: 200-300 MB/s
  - SSDs: 2-13 GB/s
  - Ethernet: up to 40 Gbps (i.e., 5 GB/s)
  - RAM: ~20 GB/s (DDR4), 32-64GB/s (DDR5)
- We want to **read from several disks at the same time**

Avoiding Synchronization

- **Shared Nothing** architecture:
  - Independent entities, with **no common state**
  - Synchronization introduces latencies, and it is difficult to implement without bugs
  - A goal is to **minimize sharing**, so that we synchronize **only when we need to**

Failures Are the Norm

- When you have a cluster that's big enough, failures are **the norm, not the exception**
  - Hardware, software, network, electricity, cooling, natural disasters, attacks...
  - Cascading failures when the failure of a service makes another unavailable
  - Most failures are transient (data can be eventually recovered)

Move Processing to the Data

- High-Performance Computing (HPC):
  - Distinction between performance & storage nodes
  - **CPU-intensive** tasks: computation is the bottleneck
- **Data-Intensive** workloads:
  - Network (if not the disks) is generally the bottleneck
  - We want to process the data **close to the disks where it resides**
  - **Distributed filesystems** are necessary, and they need to **enable local processing**

Process Data Sequentially

- Data is **too large to fit in memory**, so it's **on disks**
- We've seen 200MB/s for a HDD—that's for **sequential reads**
  - Disk seeks for random disk access make everything **much slower**
- Consider a 1TB DB with 10^{10} 100-byte records
  - Updating 1% of the records with random access will require around **a month** (seek latency ~30ms)
  - Rewriting all records will require around **3 hours** (at 200 MB/s)
- There's a big advantage in **organizing computation for sequential reads**

What MapReduce is For

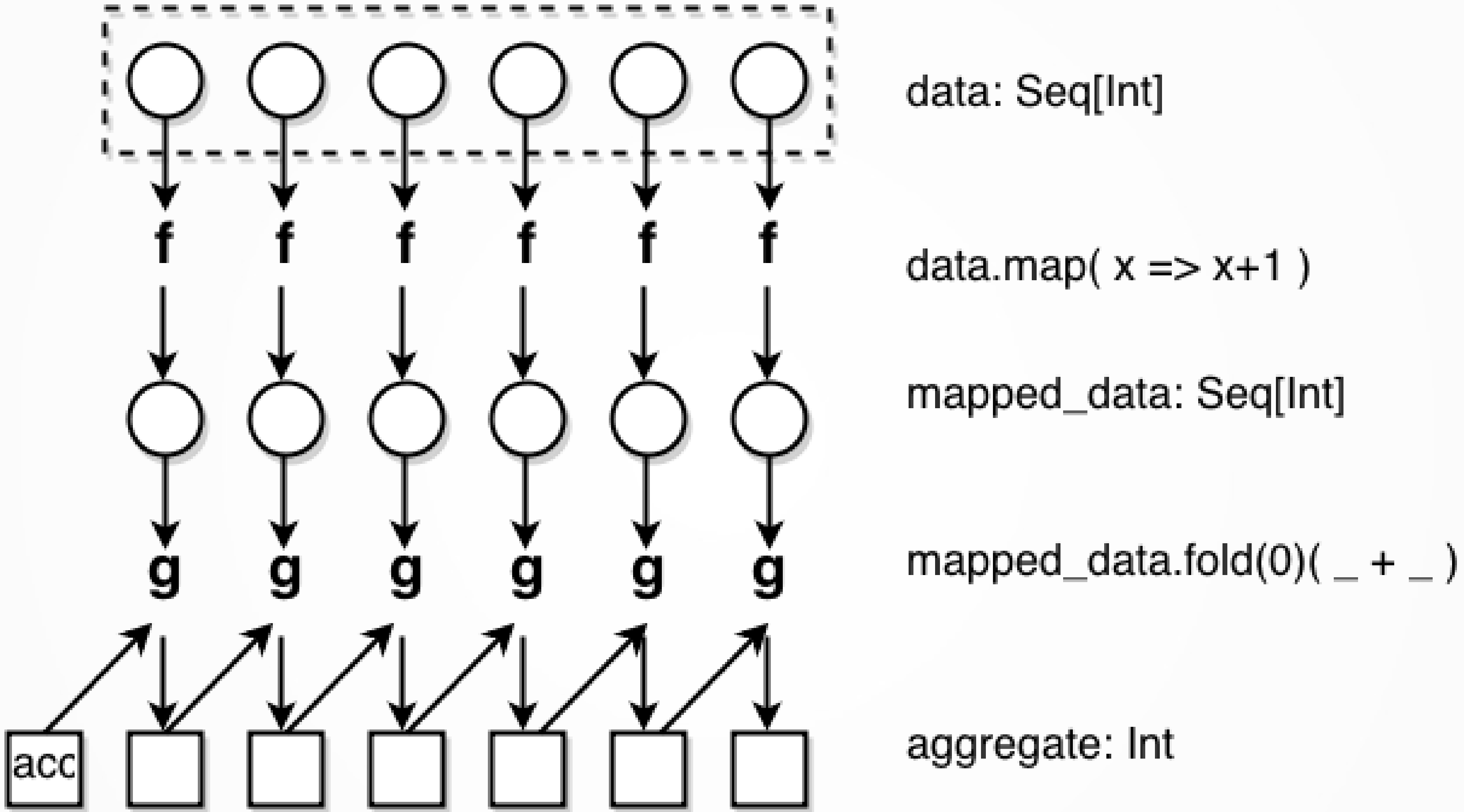
- Batch processing involving (mostly) **full scans** of a dataset
- Data collected elsewhere and copied to a distributed filesystem
- Examples:
  - Compute PageRank, a score for the “reputation” of each page on the Web
  - Process a very large social graph
  - Train a large machine learning system
  - Log analysis

Scalability Goals

- In two dimensions:
  - **Data**: if we double the data size, the same algorithm should ideally take around **twice** as much the time
  - **Resources**: if we double the cluster size, the same algorithm should ideally run in around **half** the time
- **Embarassingly parallel** problems: shared-nothing computations that can be done separately on fragments of the dataset
  - E.g., convert data items between formats, filter, etc.
- Exploit having **embarassingly parallel sub-problems**

Programming Model

Functional Programming Roots



- **Map** and **reduce** (or **fold**): higher-order functions
  - Accepting functions as arguments

Functional Map

- Takes a sequence as input
- Apply a single function ***m*** to each element of your dataset
- Produce a new sequence as output
- Example: map(neg, [4, -1, 3]) = [-4, 1, -3]

Functional Reduce

- Given a list *l* with *n* elements, an intial value v_0 and a function ***r***, the output we can compute
  - $v_1 = r(v_0, l_0)$
  - $v_2 = r(v_1, l_1)$

    ...
  - $v_n = reduce(r, v_0, l) = r(v_{n-1}, l_{n-1})$
- For example, *sum(l)=reduce(add, 0, l)*
- Can be seen as an **aggregation operation**

The MapReduce Model

- Dean and Ghemawat, engineers at Google, discovered that their scalable algorithms followed this pattern
  - A “map” part where original data is transformed, on the machines that were originally holding the data
  - A “reduce” part where the first results are aggregated
- The MapReduce framework facilitated writing programs with this style
- Implemented as free/opensource in Apache Hadoop MapReduce (originally developed at Yahoo!)

MapReduce Map Phase

- Processes data **where it's read**
  - Filter what's not needed so you don't **waste network bandwidth sending it**
  - **Transform data** (e.g., convert to the format that's best for your computation)
  - Unlike the functional *map*, this always creates key/value pairs
  - **Embarassingly parallel**: each "fragment" of input determines its own output, alone

Shuffle Phase

- Data gets **grouped by key**, so that we get a sequences of all values **mapped to the same key**
  - Handled by the execution framework (Hadoop, Google MapReduce), so programmers don’t have to do anything
  - Yet, there are optimizations possible visible to the users
  - Data gets moved on the network
  - If data is well distributed along keys, work is well distributed between machines

Reduce Phase

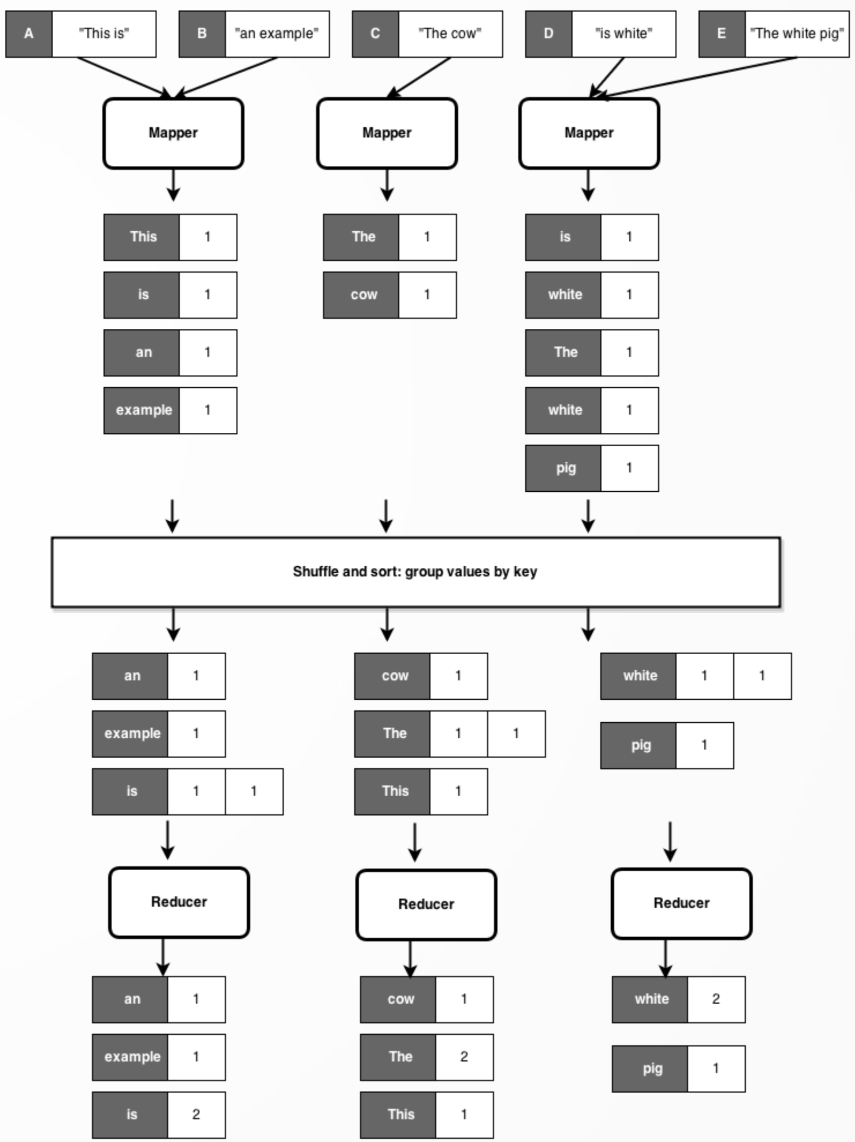
- **Reduce phase**: an **aggregation operation**, defined by the user, is performed on all elements having the same key
- The output is written on the distributed filesystem
- This output can be an **input to a further map-reduce step**

WordCount: MapReduce's Hello World

```
def map(text):
    for word in text:
        emit word, 1

def reduce(word, counts):
    emit word, sum(counts)
```

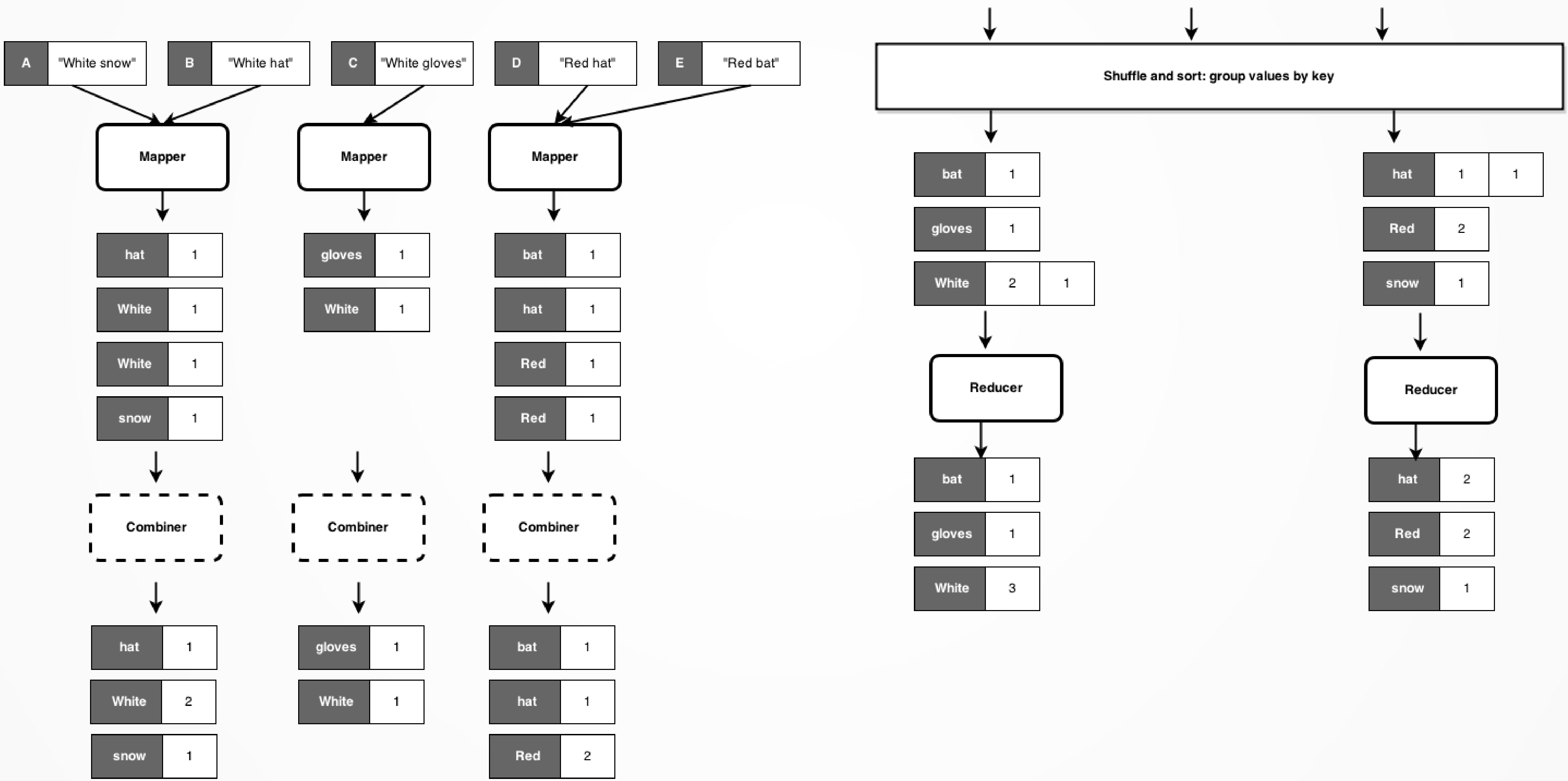
- We'll implement it in a simulated framework
- Run it on the Moby Dick text



Combiners

- A way to reduce the amount of data before sending it over the network
- They are “mini-reducers”, run on mapper machines to pre-aggregate data
- In Hadoop they’re **not guaranteed to be run**, so the algorithm must be correct without them
- We’ll write a combiner for our WordCount

Combiners in WordCount



Exercise

- Write a “MapReduce program” for computing the per-team mean “overall” stat of players in FIFA 21
- Then, try adding a combiner
  - Note: *mean(1,2,3,4,5)* **is not** *mean(mean(1,2,3), mean(4,5))*
  - Hence, the combiner cannot output partial means
  - The algorithm should work without combiners
  - Hint: try outputting *(k, (sum, n_items))* from mappers and combiners

What Can We Do With MapReduce?

- "Everything"
  - Trivially, we could send everything to a single reduce function and compute anything there
  - Would scale terribly, of course
- The question becomes: what can we do **efficiently**?
  - It's about finding scalable solutions
  - This is non-trivial! It's about optimizing computation, communication, and sharing costs well
- Many algorithms require **multiple rounds** of MapReduce

Patterns

Co-Occurrence Matrices

- Problem: building a co-occurrence matrix
  - *M*, a square $n \times n$ matrix, where n is the number of words
  - A cell m_{ij} contains the number of times word w_i occurs **in the same context** of w_j (e.g., appear in the same sliding window of k words)
- A building block for more complex manipulations
  - E.g., Natural language processing (NLP)
  - Similar problem: recommender systems
    - *"Customers who buy X often also buy Y"*

Is the matrix too large?

- *M* has size n^2: it can become very big quickly
- English: hundreds of thousands of words
  - i.e., tens of billions of cells
- Other use cases: billions
  - i.e., forget about it
- Most of those cells will anyway have a value of 0
  - Let’s just compute the nonzero values!

The Pairs Approach

- Use **complex keys**: when the mapper encounters w_1 close to w_2, it will emit the $((w_1,w_2), 1)$ pair, meaning "I've found the (w_1,w_2) pair once"
- From there on, it really looks similar to WordCount :)
- Let's do it as an exercise!

The Stripes Approach

- Say we have words *[b, c, b, d]* in the context around word *a*
- The mapper will return *(a, {b: 2, c: 1, d: 1})*: we associate to the key *a* a mapping to all the words corresponding non-empty columns in a matrix row
- The combiner and the reducer will aggregate each of the stripes