Distributed Systems

Christian J. Rudder

January 2025

Contents

Contents 1

1 Introduction 5
1.1 MapReduce 6

Bibliography 9

This page is left intentionally blank.

Big thanks to **Prof. Anna Arpaci-Dusseau**
and **Prof. Ioannis Liagouris** for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

— 1 —

Introduction

1.1 MapReduce

It's 2004 and Google is looking for a way to process large amounts of web-crawled data efficiently in parallel. They found that most jobs follow the same pattern, leading to the following model:

Definition 1.1: MapReduce

MapReduce automatically parallelizes and executes client jobs provided they give these two functions:

- **Map**: Takes a set of input key-value pairs and produces a set of intermediate key-value pairs.
- **Reduce**: Takes an intermediate key and a set of values for that key, and merges them into a smaller set of values.

If you are familiar with functional programming, the MapReduce model is similar to the `map` and `reduce` functions as seen in languages like Python or JavaScript.

Example 1.1: Word Count in MapReduce

Input: A collection of documents (each document has a name and contents).
Output: The total frequency of each word across all documents.

Map:

- **Key:** document name
- **Value:** document contents
- **Emit:** for each word w in the document, emit $(w, 1)$, of form (key, value)

Reduce:

- **Key:** a word w
- **Value:** list of counts $\{1, 1, \ldots, 1\}$ from all maps
- **Emit:** $\left(w, \sum \text{counts}\right)$, i.e. the total occurrences of w

Example: $\{(A$, "the dog likes to sit"), $(B$, "the lion does not sit")$\}$ maps to
{(the, 1), (dog, 1), (likes, 1), (to, 1), (sit, 1), (the,1) (lion, 1), (does, 1), (not, 1),(sit,1)}

This is then reduced to,
{(the, 2), (sit, 2), (dog, 1), (likes, 1), (to, 1), (lion, 1), (does, 1), (not, 1)}. ■

We can even stack multiple MapReduce jobs together. For example consider the below figure based on trying to find the set difference in Example (1.1):

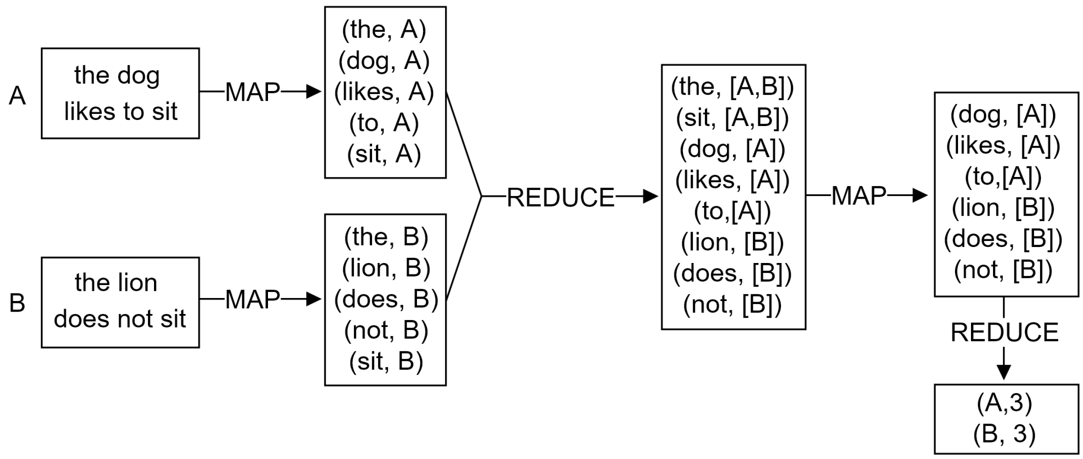

Figure 1.1: Both sets A and B under going two rounds of MapReduce. The first creates a set of words between A and B. The second round discards non-singletons, reducing the set to counts of elements in A and B.

Now to bring this into a distributed system:

Definition 1.2: Implementing MapReduce in a Distributed System

To implement MapReduce with a single coordinator as follows:

- **Split Data**: First take the input data and split it into M chunks.
- **Assign Maps**: The coordinator distributes the M chunks to N worker nodes (may receive multiple chunks).
- **Map Phase**: Each worker processes its chunk and partitions the output into R sections on disk. This is achieved by uniformly hashing the intermediate keys into R buckets.
- **Shuffle Phase**: The coordinator collects the R partitions from all workers and redistributes them back to N workers.
- **Reduce Phase**: Each worker processes its partition and writes the final output to disk.

Additionally, W and R jobs should outnumber N workers to keep them busy (i.e., $W \gg N$ and $R \gg N$).

Now to deal with failures:

Definition 1.3: Fault Tolerance in MapReduce

We deal with hiccups in our system via the following:

- **Map/Shuffle Phase**: If a worker fails to map or goes offline with the intermediate data, the coordinator will reassign the chunk to another worker.
- **Reduce Phase**: If a worker fails to reduce, the coordinator will reassign the partition to another worker.
- **Coordinator Failure**: If the coordinator fails, the system will need to restart the entire MapReduce job. Failures are not recoverable, and are assumed to be rare.

If a worker is slow (**straggler**), the coordinator reassigns its task and reacts accordingly:

- **Map Phase**: The coordinator will only point to the first worker that finishes the map task for intermediate data.
- **Reduce Phase**: It doesn't matter who finishes first, as they write the same data to the same location on disk (e.g, "/filepath/final_data/id"). Moreover, writing is atomic.

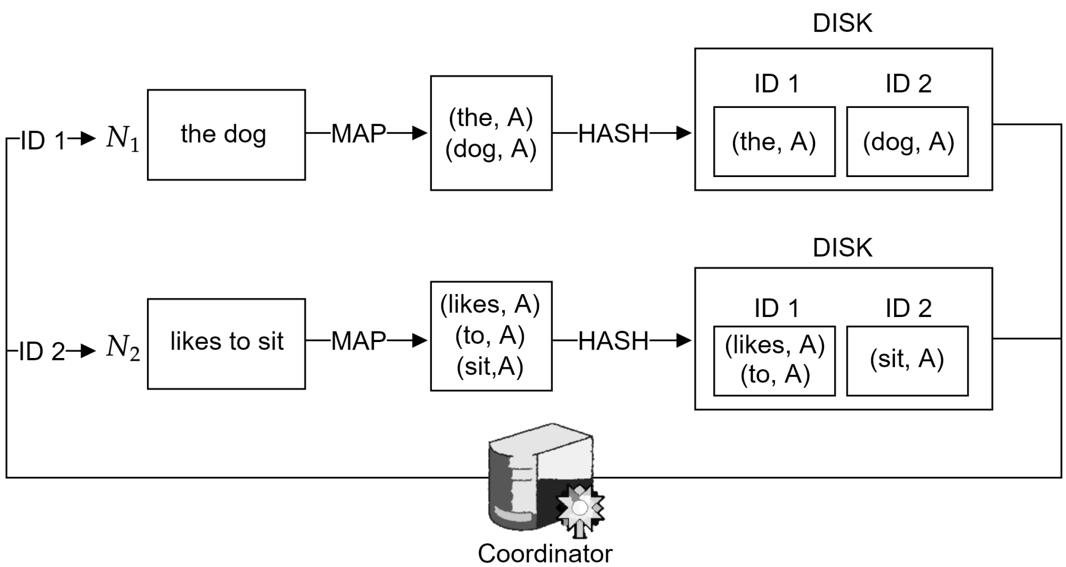

Figure 1.2: In a simplified MapReduce system, two workers receive a partitioned map job of "the dog likes to sit". After N_1 and N_2 finish processing the job, they hash the intermediate data into $R = 2$ buckets. The coordinator then collects the buckets assigning ID 1→ N_1 and ID 2→ N_2, to finish the reduce job.

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.