Distributed Systems

Christian J. Rudder

January 2025

Contents

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 Sharding Data & Consistent Hashing

Say we have a 1 TB dataset with only 5 servers each with 200 GB of storage.

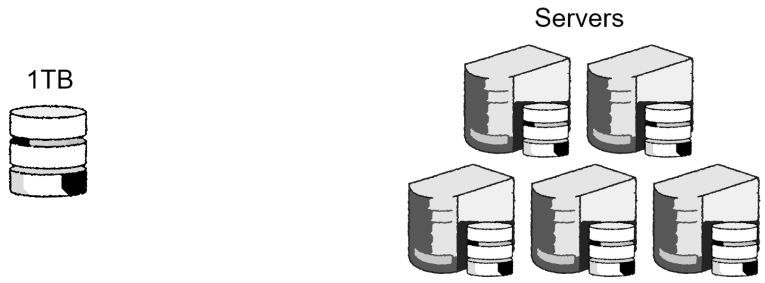

Figure 1.1: 1 TB dataset with 5 servers each with some storage capacity.

Once method is to split up our dataset into 5 partitions, and distribute them across the servers.

Definition 1.1: Sharding

Sharding is the process of splitting up a dataset into smaller, more manageable pieces called **shards**. Each shard is stored on a different server, allowing for parallel processing.

In addition, to strengthen fault tolerance, replication could be used to store multiple copies of a single shard on different servers.

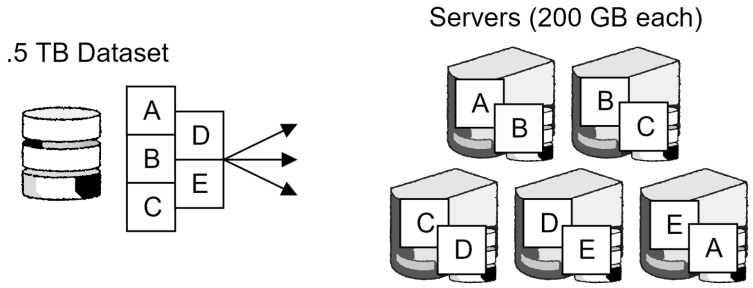

Figure 1.2: Sharding a dataset of .5 TB into 5 partitions, each part replicated twice and distributed evenly across 5 servers.

As shown above, shards could be replicated and housed with other shards on the same server. For instance, this could help in the case that two pieces of data are frequently accessed together.

Now we discuss how to assign shards to servers. There are two naive methods:

- **Randomized**: Though statistically balanced, it requires a lookup procedure to find shards.
- **Alphabetical**: Too deterministic, though we skip the lookup procedure.

Instead we could combine these ideas and use a **hash function** to map the data to a server:

Definition 1.2: Hash Function

A **hash function** is a function that takes an input (or **key**) and returns a fixed-size string of bytes. The output is typically a **digest** that is unique to each unique input. Depending on the hash algorithm, the output may be a number or a string of characters, or even produce collisions (two distinct inputs producing the same output).

Ideally the hash function should be:

- **Deterministic**: The same input should always produce the same output.
- **Uniform**: The output should be uniformly distributed across the range of possible outputs.
- **Fast**: The hash function should be fast to compute.
- **Collision Resistant**: It should be hard to find clashing inputs that produce the same output.

Though this seems fine, it may be costly to deal with migrations:

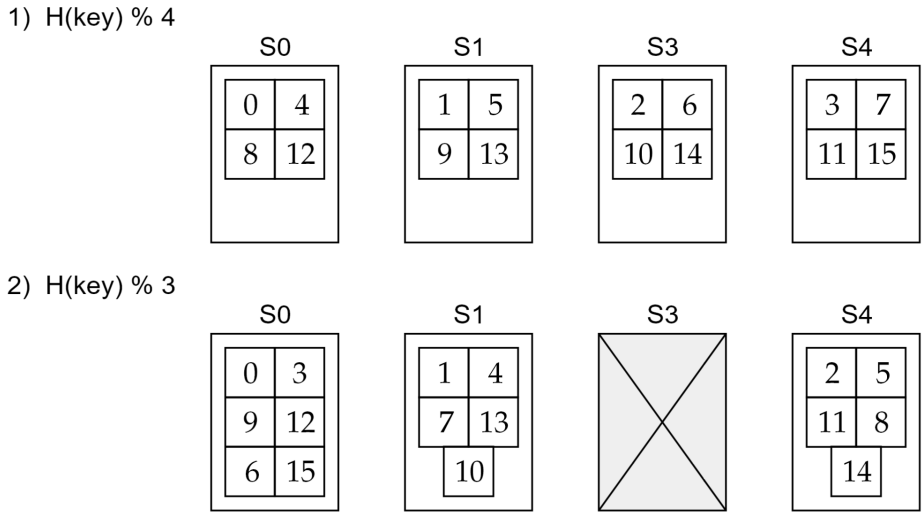

Figure 1.3: Given a hash function H and a shard ID key, where $H(key)$ returns some server ID – 1) There are 4 servers utilizing the hash function, using the size of servers to modulo overflow. 2) Server 3 goes down, forcing a rehash of the data (unnecessary migration).

To fix this migration problem, we build off the idea of the wrapping modulo overflow behavior, as well as to allow severs to handle multiple hash values:

Definition 1.3: Consistent Hashing (Part 1)

Consistent Hashing is a technique used to distribute data across a cluster of servers in a way that minimizes the amount of data that needs to be moved when servers are added or removed.

Here the hash function H maps keys to an m-bit value, which provides a hash-space of 2^m. I.e., we have 2^m possible hash values ($\{0, 1, \ldots, 2^m - 1\}$), which forms a ring, allowing us to wrap around the hash space.

Then servers S_i often called **virtual nodes** or **virtual replicas**, are evenly assigned a portion of the ring to cover. Any hash that falls within this range is assigned to the server. If a server S_i goes down, its range is passed to the next server S_{i+1} (without having to rehash the data).

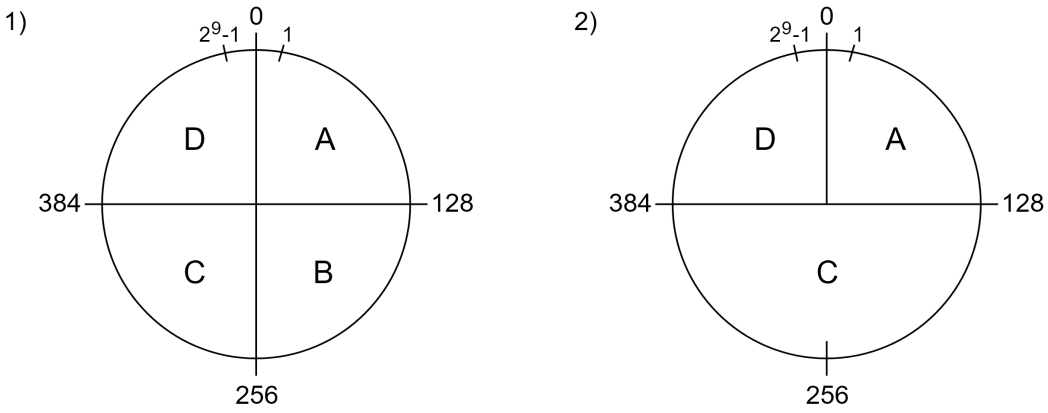

Figure 1.4: A consistent hashing ring of 9-bit values (512 possible values). The ring includes 4 servers A, B, C, D. 2) Server B goes down, transferring its range to C.

In Figure (1.4) in part 2, the data is **no longer evenly distributed** across servers after a lost server. Likewise, the same problem occurs when adding a server. We mitigate this via the following:

Definition 1.4: Consistent Hashing (Part 2)

To achieve an even distribution of virtual nodes, we in essence take server S_i's range and split it into k slices, distributing them evenly across the ring.

In particular, given n servers, and a ring of size 2^m, we create k virtual nodes to each server, requiring k different hash functions for server assignment. Typically, $k \approx \log_2(2^m) = m$. [2]

Below we illustrate how just a basic implementation of this improves our load balancing:

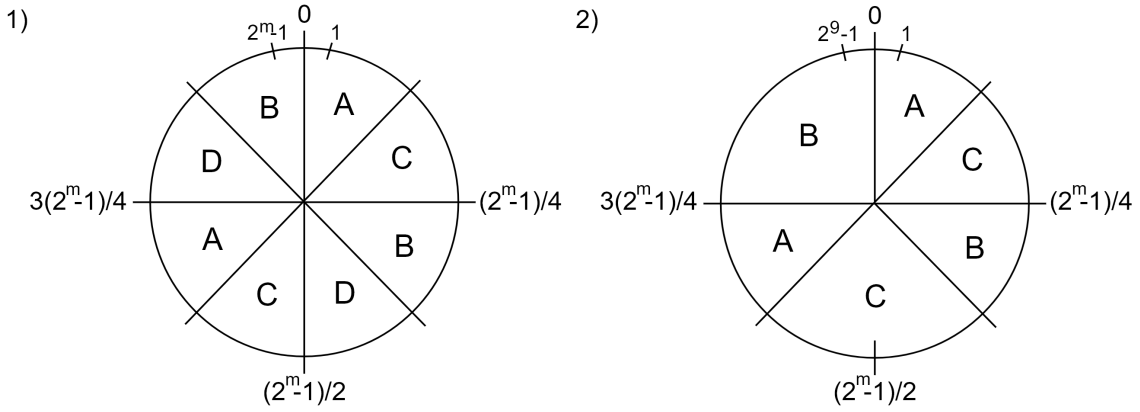

Figure 1.5: A consistent hashing ring of m-bit values, where we choose $k = 2$ splits for $n = 4$ servers across the ring. 1) All 4 servers and their replicated virtual nodes in even distribution. 2) Server D goes down, transferring its range to B and C.

In the above figure, choosing just $k = 2$ makes a huge difference as opposed to our previous solution in Figure (1.5).

Bibliography

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

[2] Tim Roughgarden and Gregory Valiant. CS168: The Modern Algorithmic Toolbox, Lecture #1: Introduction and Consistent Hashing. Lecture notes, Stanford University, April 2024.