Key-Value Store Design: Addressing Scalability, Availability, and Consistency:

A single server Key-Value store presents a single point of failure (SPOF). While rehashing is a simple approach to distribute data, it requires remapping *all* keys when servers are added or removed, causing significant disruption. Consistent hashing mitigates this by minimizing the impact of server changes. When a server is added or removed, only a fraction (approximately *k/n*, where *k* is the number of replicas and *n* is the number of servers) of keys need to be remapped.

Data Distribution and Virtual Nodes:

Consistent hashing distributes data across the ring. However, simple consistent hashing can lead to uneven data distribution and imbalanced load, especially during server additions or removals. Virtual nodes address this. A virtual node represents a real server and is placed at multiple points on the ring. For example, server S1 might have virtual nodes S1_1, S1_2, ..., S1_N. Virtual nodes improve data distribution uniformity and provide a more balanced load distribution, minimizing the impact of server failures.

Replication and Consistency:

To ensure availability and reliability, data is replicated asynchronously (for eventual consistency) or synchronously (for strong consistency) across multiple servers. A common strategy is to replicate each data item on *k* out of *N* servers ($k < N$).

Consistency models represent a trade-off between consistency and availability. Systems must be partition-tolerant (handling network partitions), leaving us to balance availability and consistency. Common consistency levels include:

- **Strong Consistency:** All clients see the same data at the same time.
- **Weak Consistency:** Clients might see different versions of data at different times.
- **Eventual Consistency:** If no further updates occur, all clients will eventually see the same data.

Quorum Consensus:

Quorum consensus is used to achieve the desired consistency level. Parameters *W* (write quorum), *R* (read quorum), and *N* (number of replicas) are tuned to control consistency. *W* represents the minimum number of acknowledgements required for a successful write, and *R* represents the minimum number of acknowledgements for a successful read. Importantly, *W*

and *R* refer to the number of *acknowledgements*, not necessarily the number of servers written to or read from. To guarantee strong consistency, the condition $W + R > N$ must be met.

Conflict Resolution and Vector Clocks:

In distributed systems, data conflicts can arise. Vector clocks are used to track the causal relationships between updates and help resolve conflicts. However, vector clocks can grow large. A common strategy is to truncate them after a certain threshold, discarding older entries. Conflict resolution logic is typically handled by the client.

Failure Handling:

- **Failure Detection:** Gossip protocols are often used for efficient failure detection. They maintain a membership list and propagate information about server status, avoiding the overhead of all-to-all communication.

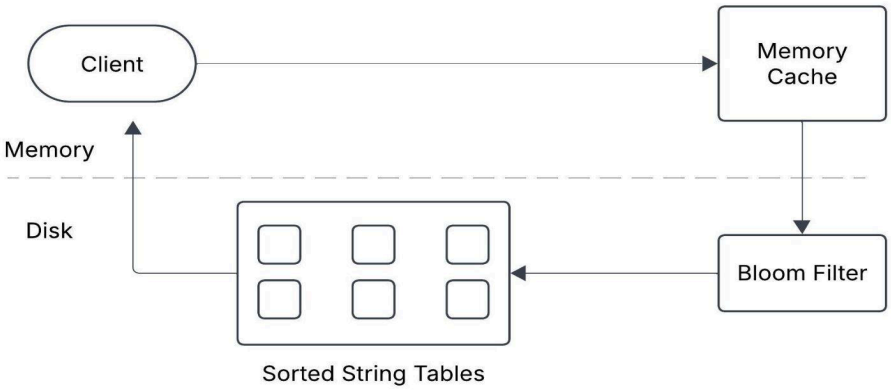
- **Sloppy Quorum:** During temporary failures, *sloppy quorum* relaxes the quorum requirements. Writes are accepted by the first *W* healthy servers, and reads are served by the first *R* healthy servers, even if they are not the "closest" ones according to the consistent hash ring.

- **Hinted Handoff:** When a server is temporarily unavailable, another server accepts writes on its behalf (a "hint"). When the original server recovers, the hinted writes are transferred. This ensures data durability during transient failures.

- **Permanent Failure:** For permanent failures, Merkle trees can be used to efficiently detect and synchronize inconsistencies between replicas, minimizing data transfer during recovery.

This revised description provides a more structured and technically accurate overview of your Key-Value store design. It clarifies the purpose of each component and its role in achieving scalability, availability, and consistency.

Read Path



Write Path