Design a key value database(like amazonDB, memCached, redis, etc.)

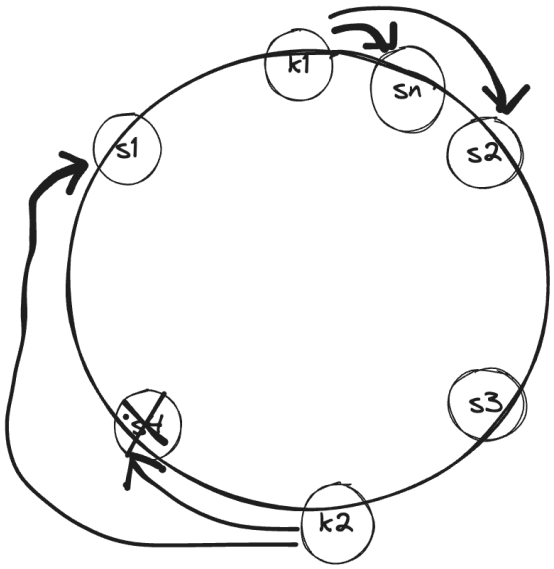
**Problem And Design Scope:**
1. Design should support two main operation get(key) and put(key, value)
2. What would be the size of each key-value pair?
    - less than 10 KB.
3. Ability to store big data.

**Non functional:**
1. As this is No-SQL DB so it should be an AP(Availability and partition tolerance) system.
2. Ability to store big data.
3. High availability: The system responds quickly, even during failures.
4. High scalability: The system can be scaled to support large data set.
5. Automatic scaling: The addition/deletion of servers should be automatic based on traffic.
6. Tunable consistency.
7. Low latency.

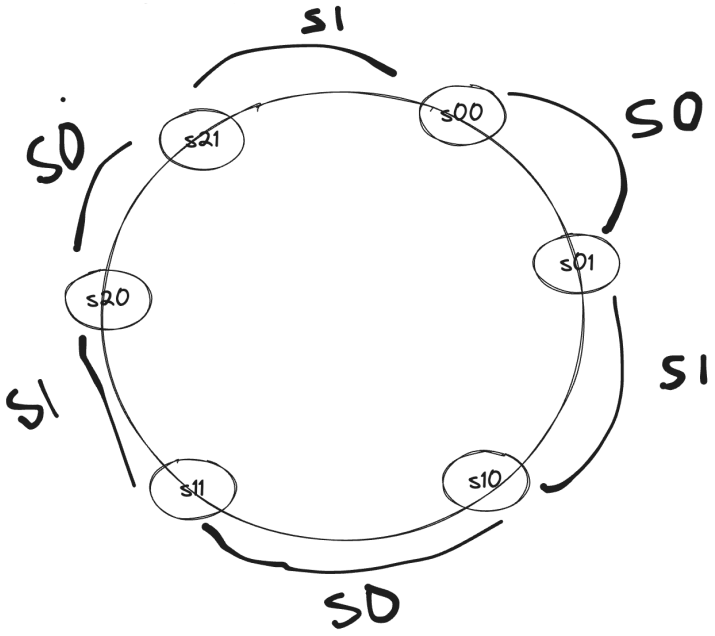
**System Components:**
1. Data Partition
2. Data Replication
3. Consistency
4. Inconsistency resolution
5. Handling failures
6. System Architecture Diagram
7. Write path
8. Read path

**Data Partition:**
For large applications, it is infeasible to fit the complete data set in a single server.
So we can split the data into small partitions and store them into multiple servers. There are two challenges while partitioning the data:
1. When a server goes down the read/write operations coming to that server will fail to update.
2. Distribute data evenly over different servers.

The 1st point can be solved using consistent hashing. By distributing the requests over a hash ring and whenever a server is removed(s4) all requests(k2) pointing to that server will point to the next server(s1) in the hash ring clockwise.
If a server(sn) is added all requests(k1) between the previous server and new server hash will point to new server.




For evenly distributed data we can add virtual nodes. Greater the virtual nodes greater the likelihood of evenly distribution.



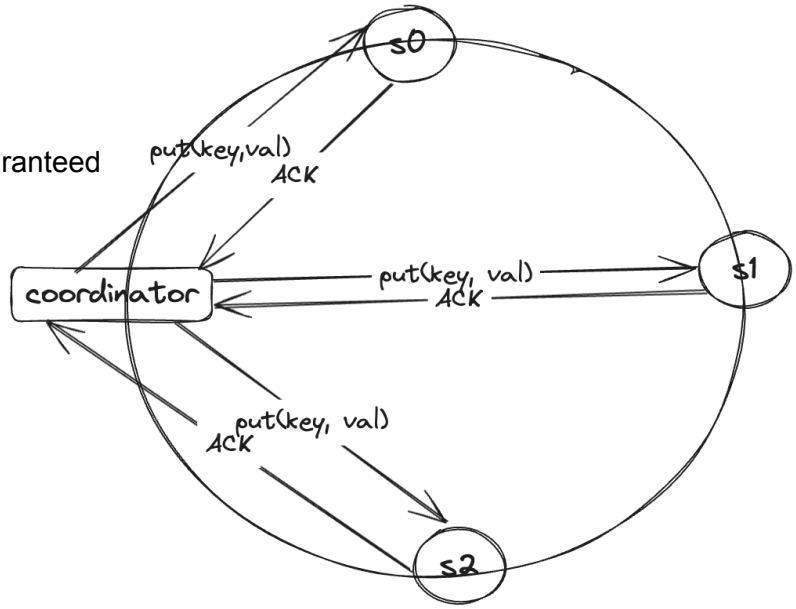

**Data Replication:**
To achieve high availability and reliability, data must be replicated asynchronous over N servers. For better availability, replicas are placed in distinct data centers, and these data centers are connected through high speed networks.

**Consistency:**
Since data is replicated at multiple nodes, it needs to be synchronized across replicas. **Quorum census** can guarantee consistency for both read and write operations.

N= number of replicas
R= read quorum, number of successful read operations required from R number of replicas to consider successful read.
W= write quorum, number of successful write operations required from W number of replicas to consider successful write.
The value of R , W and N is a tradeoff with consistency and latency.

Consider the following for W=3.

If R=1, W= N, then the system is fast read
If R=N, W= 1, then the system is fast write
If W+R > N, then strong consistency is guaranteed
If W+R<=N, then strong consistency is
Not Guaranteed




There are 3 types of consistency:
1. Strong consistency(any read operation will see the most updated data)
2. Weak consistency(subsequent read operation may not see the most updated data)
3. Eventual consistency(given enough time the data in all replicas will be consistent)

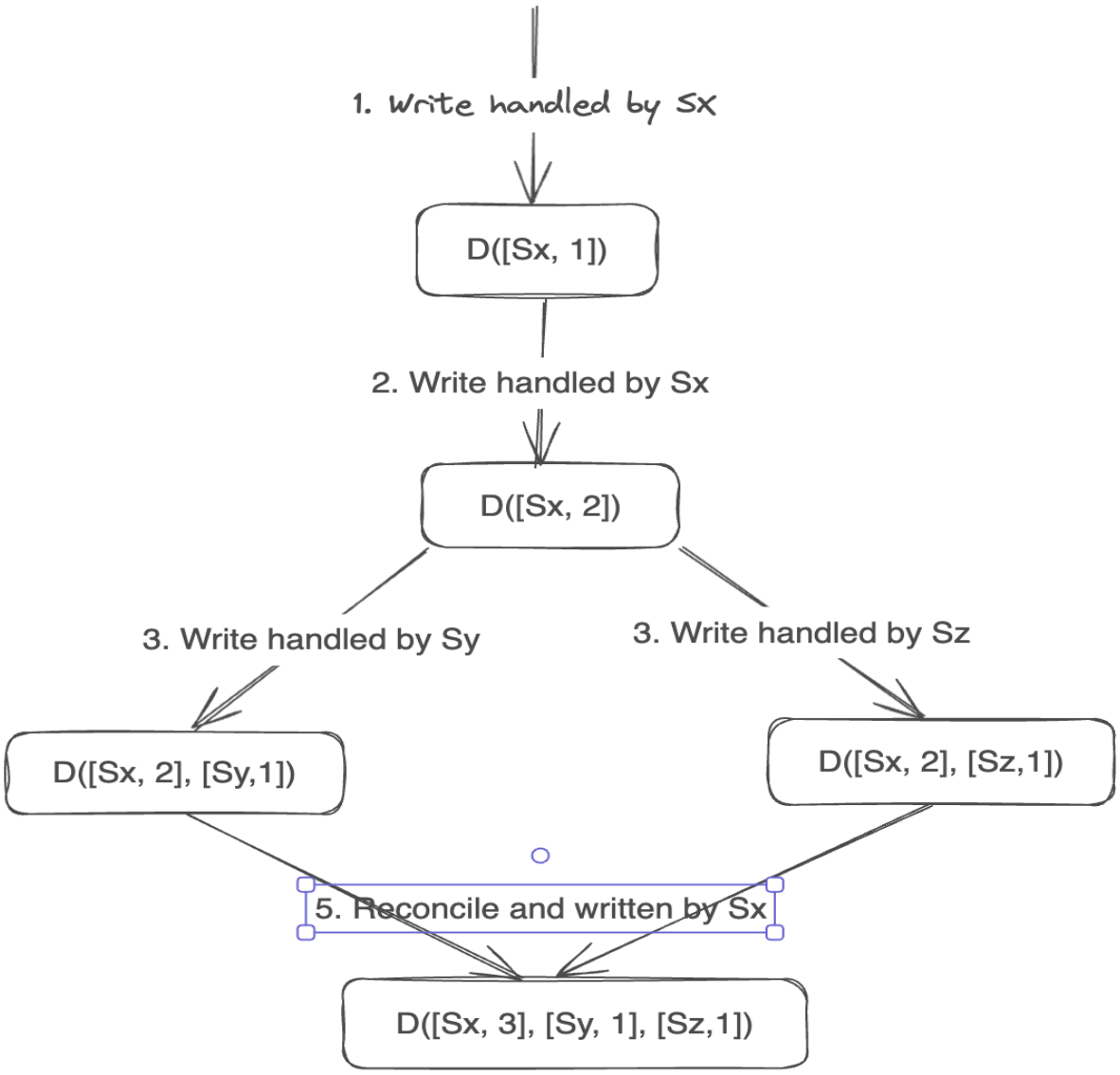
Strong consistency can be achieved by forcing a replica not to take any operation until it is updated to the most recent data. But this is not ideal for the systems which require high availability.

From concurrent writes, eventual consistency allows inconsistent values to enter the system and force the client to read the values to reconcile

### Inconsistency resolution(versioning):

To resolve the issue of inconsistencies among replicas, we can use versioning and vector locks. Versioning means treating each value as an immutable version of data.

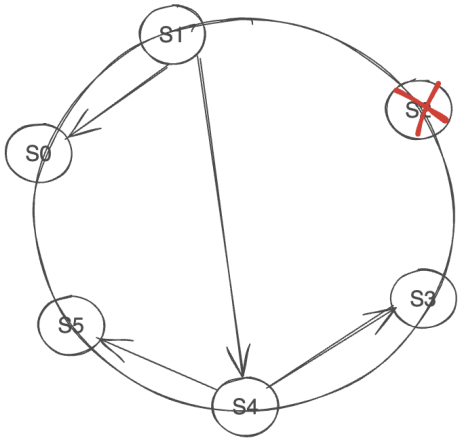
This can be implemented using a **vector clock**, Here is how it works:




But there are two problems with this:
1. Increase in complexity due to conflict resolution
2. The pairs of [Server : version] can grow rapidly. One solution is to keep a threshold and remove the old ones.

**Handling failures:**
Using decentralized failure detection methods like **gossip protocol**.
Gossip protocol works as follows:
1. Every node maintains a membership list, which contains memberId and heartbeat counters
2. Each node periodically increments its heartbeat counter
3. Each node sends heartbeats to a set of nodes which again propagates to some other nodes.
4. Once nodes receive heartbeats, members list is updated with latest info.

5. If the heartbeat has not increased for some predefined threshold time, then that server is considered offline.




| member ID | heartbeat | time |
|---|---|---|
| 0 | 10232 | 12:00:01 |
| 1 | 10224 | 12:00:10 |
| 2 | 9987 | 11:58:01 |
| 3 | 10237 | 12:00:15 |
| 4 | 10234 | 12:00:22 |
| 5 | 10226 | 12:00:20 |

**Handling Temporary Failures:**
After failure is detected, the system needs to deploy some mechanisms to ensure availability. As per quorum approach, read and write will be blocked.

So we can use a technique called **sloppy quorum** to improve availability. Instead of enforcing the quorum requirement, the system chooses the first W healthy servers for write and first R healthy servers for reads in the has ring. Offline servers are ignored.

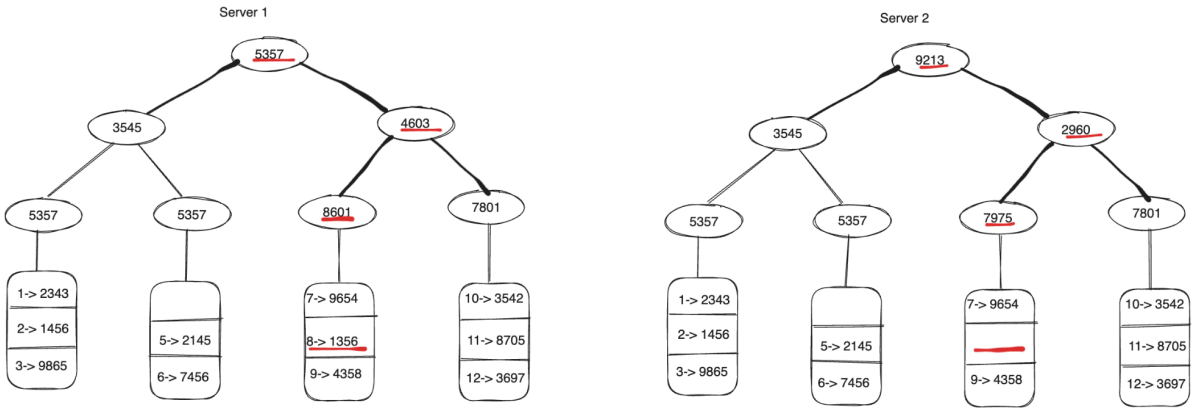
If a server is unavailable due to network or server failures, another server will process requests temporarily. When the down server is up, changes will be pushed back to achieve data consistency. This process is called **hinted handoff**.

**Handling permanent Failures:**
To handle such a situation, we use anti-entropy to keep replicas in sync.
**Anti-entropy** involves comparing each data piece on a replica and updating each replica to the newest version.
A **merkle tree (hash tree)** is used for detecting inconsistency and minimizing the amount of data transferred.

A hash tree or merkle tree is a tree in which every non-leaf node is labeled with the hash or values of its child nodes.

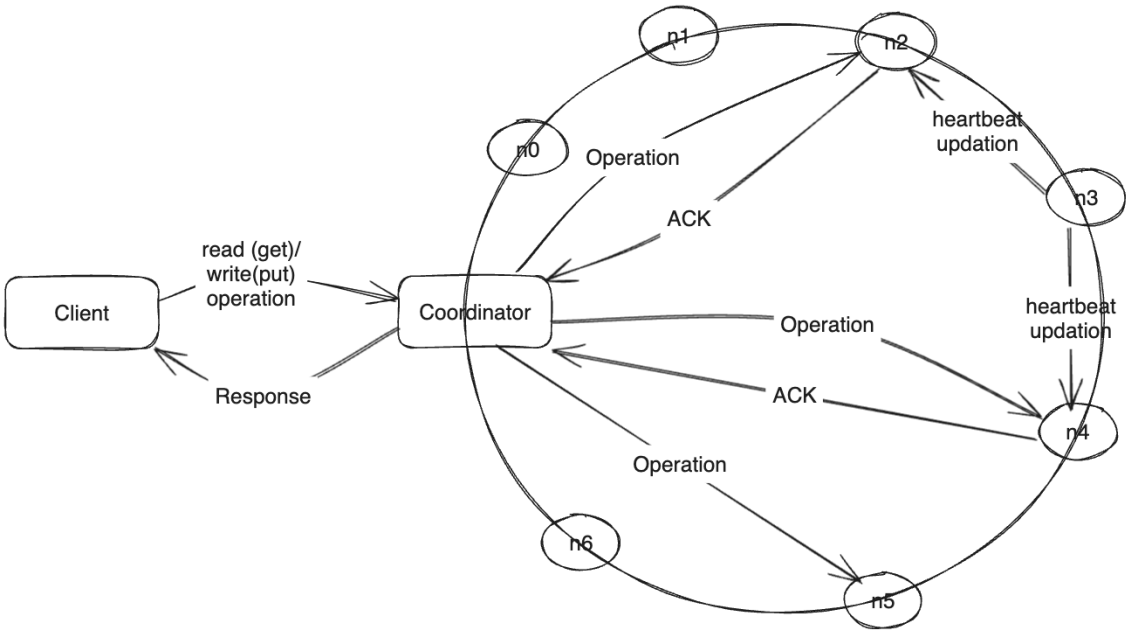
How to build a merkle tree:
1. Divide key space into buckets. A bucket is used as the root level node to maintain a limited depth of the tree.
2. Once the buckets are created, hash each key in a bucket using a uniform hashing method.
3. Create a single hash node per bucket.
4. Build the tree upwards till root by calculating hashes of children.

To compare two merkle trees, by comparing root hashes. If root hashes match both servers have the same data. If root hashes disagree, then the left child hashes are compared followed by right child hashes and so on.
You can traverse the tree to find which buckets are not synchronized and synchronize those buckets only.

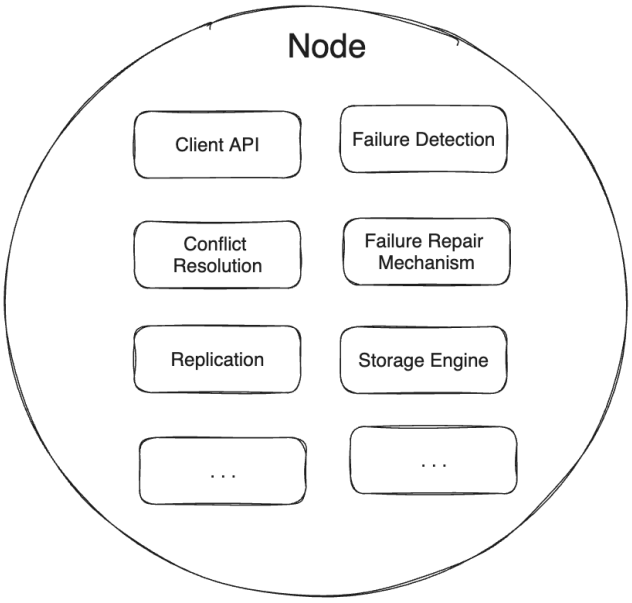



**Handling Data Center outage:**
To build a system capable of handling data center outage, it is important to replicate data across multiple data centers.

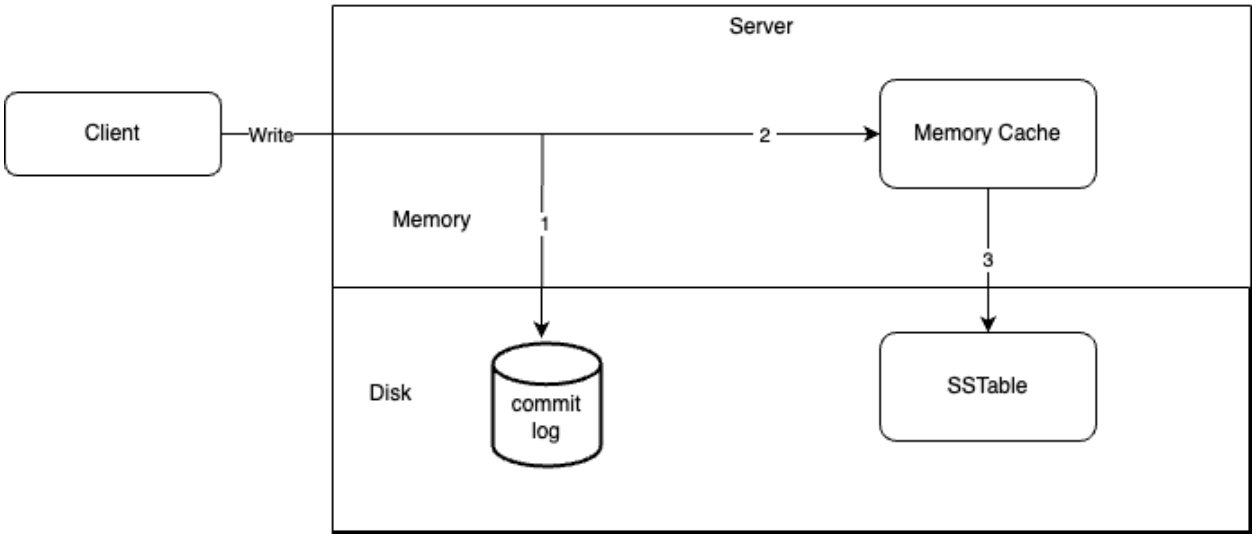
**System Architecture Diagram:**



Various decentralized operations taken care by each node:

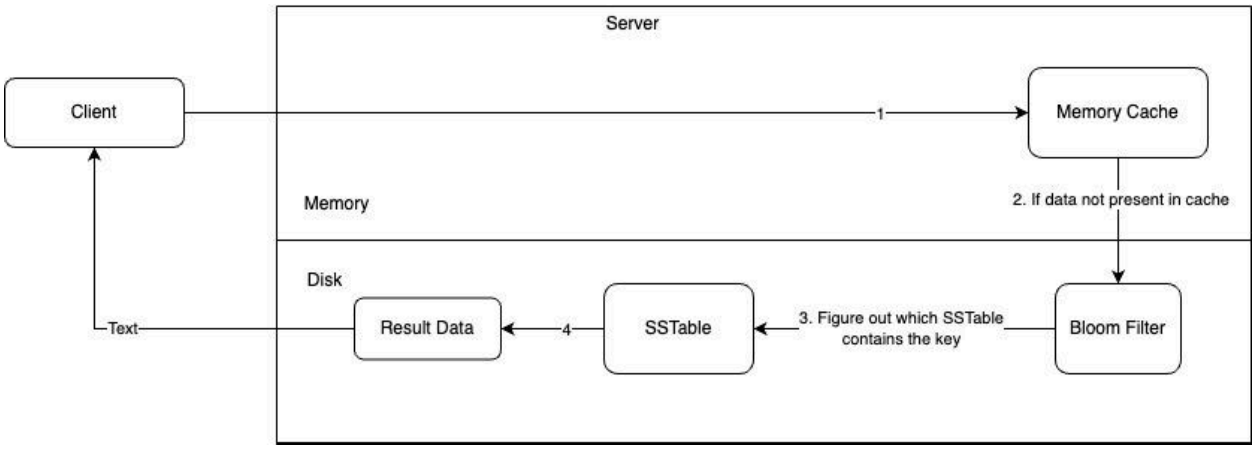



**Write Path:**




A sorted-string Table(SStable) is a sorted list on <key, value> pair. Data is flushed to SSTable when data in the memory cache reaches a certain threshold.

**Read Path:**


Server
Client
1
Memory Cache
Memory
2. If data not present in cache
Disk
Text
Result Data
4
SSTable
3. Figure out which SSTable contains the key
Bloom Filter