Design a Key-Value Store

- A key-value store, also referred to as a key-value database, is a non-relational database. Each unique identified as a key with its associated value. This data-pairing is known as 'key-value' pair.
- The key must be unique. Keys can be plain text or hashed values.

Key	value
145	john
147	bob
160	Julia

- Design a key-value store that supports the following operations:
  - put (key, value)
  - get (key)

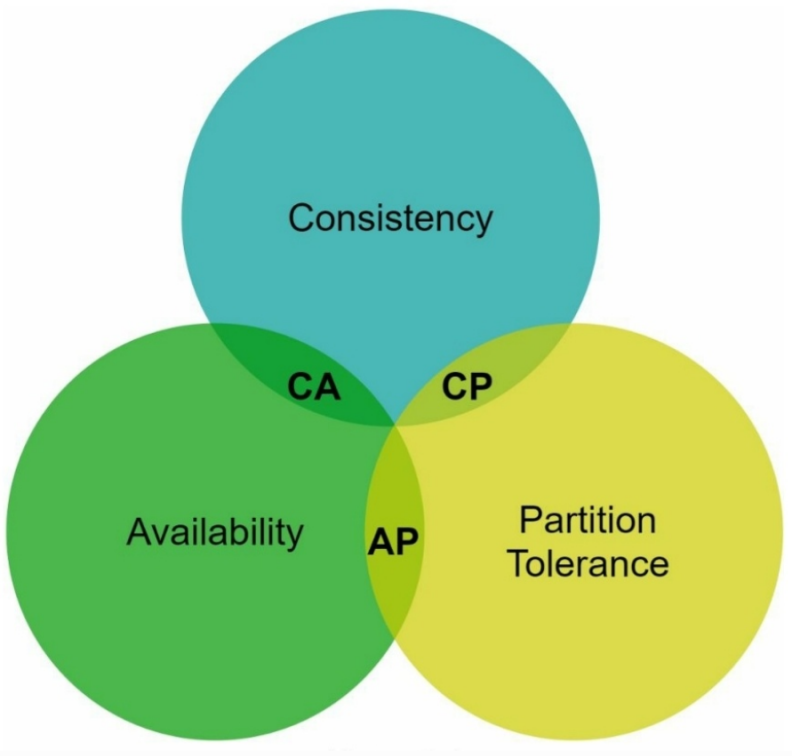
- Understand the problem and establish design scope
  - The size of key-value pair is small: less than 10 KB.
  - Ability to store big data.
  - High Availability: The system responds quickly, even during failures.
  - High Scalability: The system can be scaled to support large dataset.
  - Automatic Scaling: The addition / deletion of servers should be automatic based on traffic.
  - Tunable consistency.
  - Low Latency.

- Single Server Key-Value Store
  - Store key-value pairs in a hash table, which keeps everything in memory.
  - Even though memory access is fast, fitting everything in memory may be impossible due to the space constraint.
  - 2 optimizations can be done to fit more data in a single server:
    - Data Compression
    - Store only frequently used data in memory and the rest on disk.
  - Even with these optimizations, a single server can reach its capacity very quickly. A distributed key-value store is required to support big data.

- Distributed Key-Value Store
  - A distributed key-value store is also called a distributed hash table, which distributes key-value pairs across many servers.
  - When designing a distributed system, it is important to understand CAP (Consistency, Availability, Partition Tolerance) theorem.

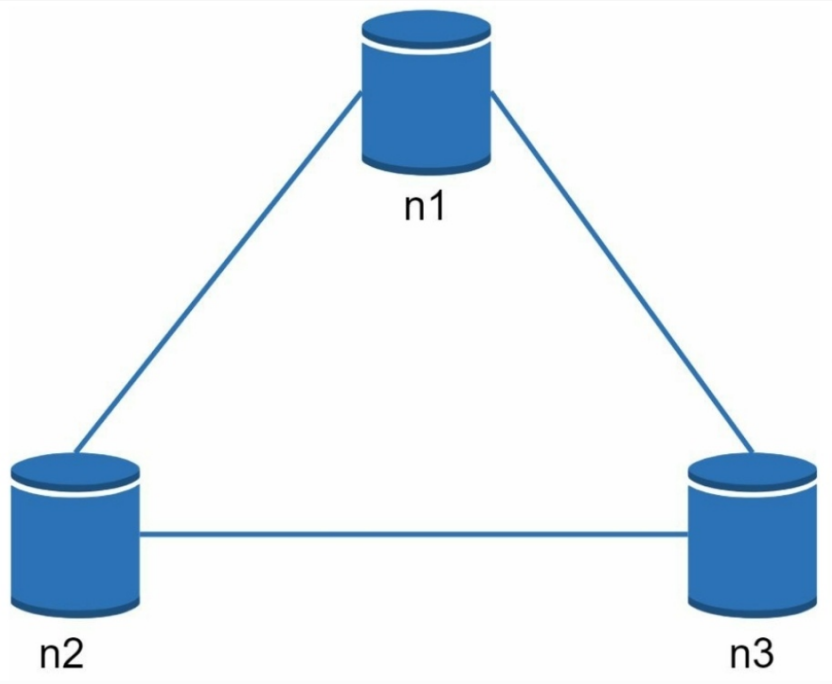
  - CAP Theorem
    - CAP theorem states it is impossible for a distributed system to simultaneously provide more than 2 of these 3 guarantees: consistency, availability and partition tolerance.
    - Consistency means all clients see the same data at the same time no matter which node they connect to.
    - Availability means any client which requests data gets a response even if some of the nodes are down.

- Partition Tolerance: A partition indicates a communication break between 2 nodes. Partition tolerance means the system continues to operate despite network partitions.
- CAP theorem states that one of the three properties must be sacrificed to support 2 of the 3 properties.

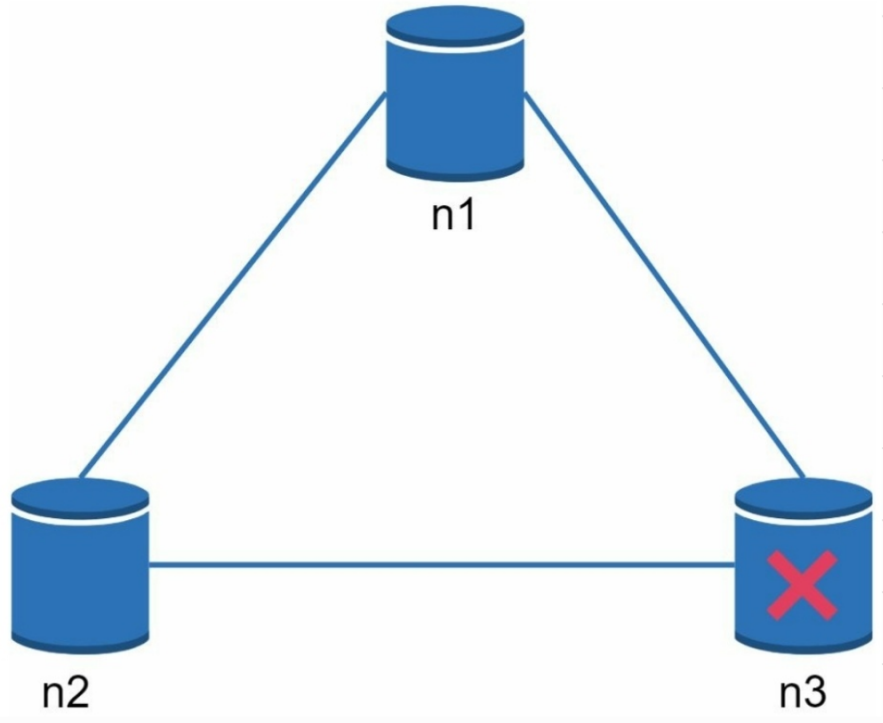


- Key-Value stores are classified based on the two CAP characteristics they support:
  - CP (Consistency and Partition Tolerance) systems: a CP key value store supports consistency and partition tolerance while sacrificing availability.
  - AP (Availability and Partition Tolerance) systems: a AP key-value store supports availability and partition tolerance while sacrificing consistency.
  - CA systems: a CA key-value supports consistency and availability while sacrificing partition tolerance. Since network failure is unavoidable, a distributed system must tolerate network partition. Thus, a CA system cannot exist in real-world applications.

- Ideal Situation
  - In the ideal world, network partition never occurs. Data written to n1 is automatically replicated to n2 and n3. Both consistency and availability is achieved.



- Real-World Distributed Systems
  - In a distributed system, partitions cannot be avoided, and when a partition occurs, we must chose between consistency and availability.



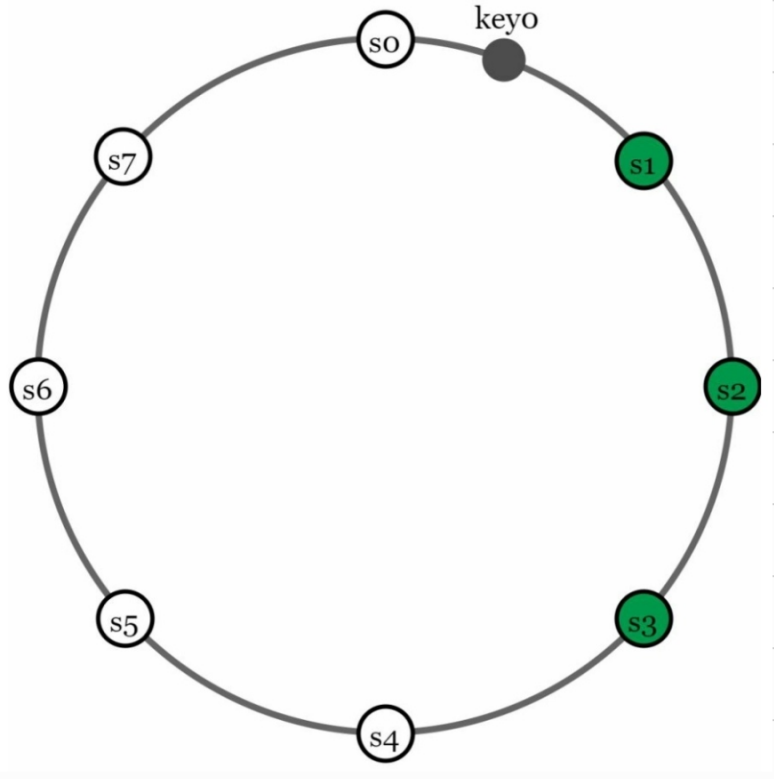
- If n3 goes down and cannot communicate with n1 and n2. If clients write data to n1 or n2, data cannot be propagated to n3. If data is written to n3 but not propagated to n1 or n2 yet, n1 and n2 would have stale data.
- If we chose consistency over availability (CP system), all write operations to n1 and n2 are blocked to avoid data inconsistency among the servers, which makes the system unavailable.
- If we chose availability over consistency (AP system), the system keeps accepting reads, even though it might return stale data. For writes n1 and n2 will keep accepting writes, and data will be synced up to n3 when network partition is resolved.

- System Components
  - Following core components and techniques are used to build a key-value store:
    - Data Partition
    - Data Replication
    - Consistency
    - Inconsistency Resolution
    - Handling Failures
    - System Architecture Diagram
    - Write Path
    - Read Path

- 3 popular key-value store system:
  - Dynamo
  - Cassandra
  - BigTable

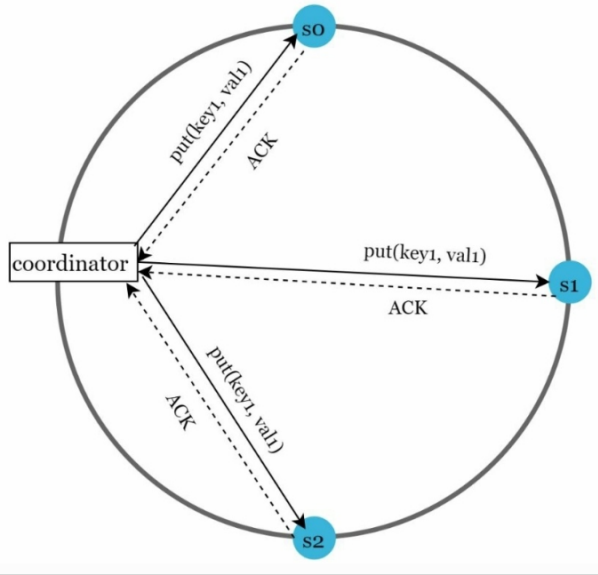
- Data Partition
  - For large applications, it is to fit the complete data set in a single server. The simplest way to accomplish this is to split the data into smaller partitions and store them in multiple servers. Challenges while partitioning the data:
    - Distribute data across multiple servers evenly.
    - Minimize data movement when nodes are added or removed.
  - Consistent hashing is use to solve the problem.

- Data Replication
  - To achieve high availability and reliability, data must be replicated asynchronously over N servers, where N is a configurable parameter.
  - These N servers are chosen using the following logic: after a key is mapped to a position on the hash ring, walk clockwise from that position on the hash ring, walk clockwise from that position and choose the first N servers on the ring to store data copies.



- With virtual nodes, the first N nodes on the ring may be owned by fewer than N physical servers. To avoid this issue, we only choose unique servers while performing the clockwise walk logic.
- Nodes in the same data center often fail at the same time due to power outage, network issue etc. For better reliability, replicas are placed in distinct data centers.

- Consistency
  - Since data is replicated at multiple nodes, it must be synchronized across replicas.
  - Quorum consensus can guarantee consistency for both read and write operations.
  - Few Definitions:
    - N = The number of replicas
    - W = A write quorum of size W. For a write operation to be considered as successful, write operation must be acknowledged from W replicas.
    - R = A read quorum of size W. For a read operation to be considered as successful, read operation must be acknowledged from W replicas.

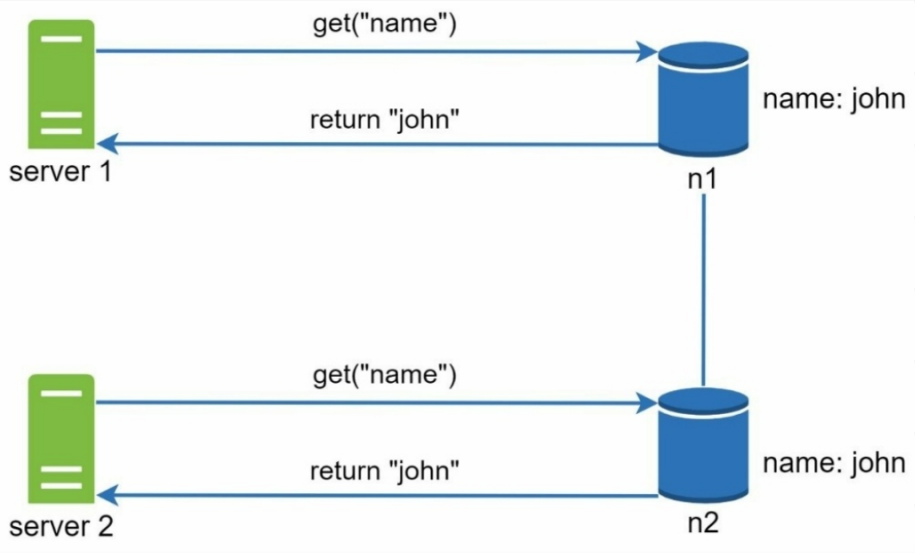


- W=1 means that the coordinator must recieve atleast 1 acknowledge-ment before the write operation is considered as successful.
- The configuration of W, R and N is a typical tradeoff between latency and consistency. If W=1 or R=1, an operation is returned quickly. If W or R>1, the system offers better consistency however query will be slower.
- If R+W>N, strong consistency is guaranteed because there must be atleast 1 overlapping node that has the latest data to ensure consistency.
- If R=1 and W=N, the system is optimized for fast reads.
- If W=1 and R=N, the system is optimized for fast write.
- If W+R>N, strong consistency is guaranteed.
- If W+R ≤ N, strong consistency is not guaranteed.

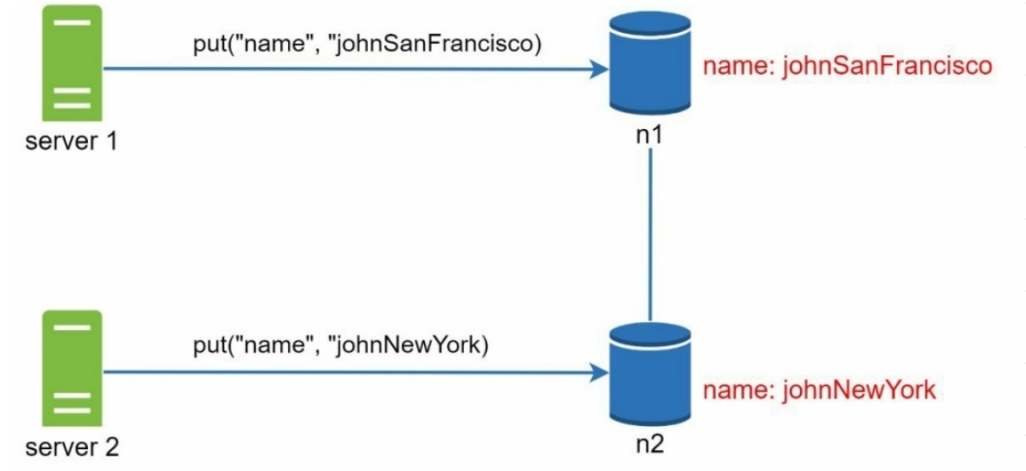
- Consistency Models
  - A consistency model defines the degree of data consistency, and a wide spectrum of possible consistency model exists:
    - Strong Consistency: any read operation returns a value corresponding to the result of the most updated write data item. A client never sees out-of-date data.
    - Weak Consistency: Subsequent read operations may not see the most updated value.
    - Eventual Consistency: this is a specific form a weak consistency. Given enough time, all updates are propagated, and all replicas are consistent.
  - Eventual consistency allows inconsistent values to enter the system and force the client to read the values to reconcile.

- Inconsistency Resolution : Versioning

- Replication gives high availability but causes inconsistencies among replicas. Versioning and vector clocks are used to solve inconsistency problems.

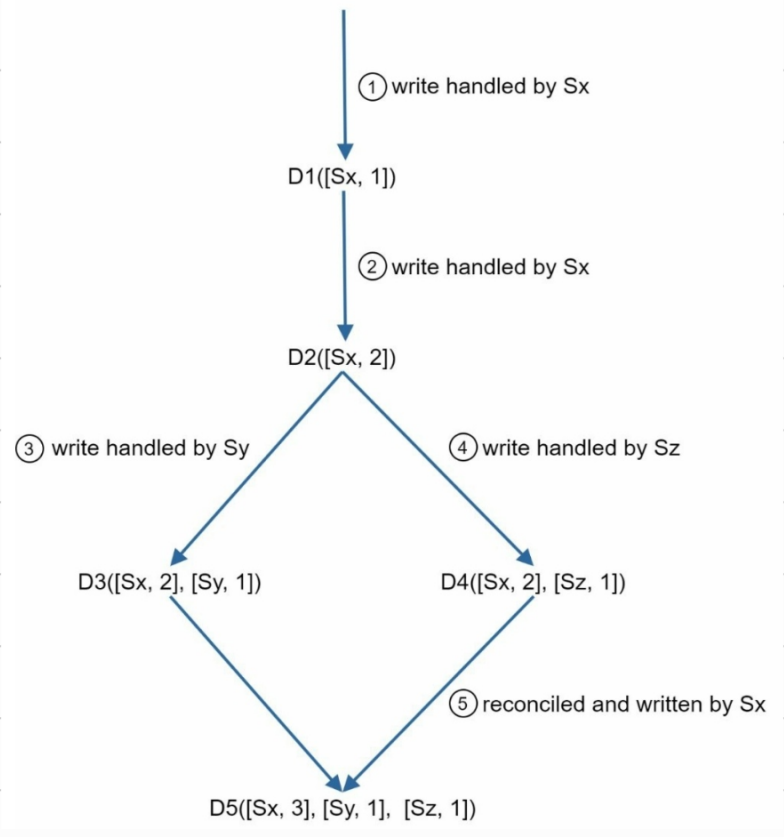


- Both replica nodes n1 and n2 have the same value. Server 1 and 2 get the same value for get("name") operation.



- Next server 1 changes the name to "johnSanFrancisco" and server 2 to "johnNewYork". These 2 changes are performed simultaneously, causing a conflict called versions V_1 and V_2.
- To resolve this issue, we need a versioning system that can detect conflicts and reconcile conflicts.
- A vector clock is a common technique to solve this problem.

- A vector clock is a [server, version] pair associated with a data item. It can be used to check if one version precedes, succeeds or in conflict with others.
- Assume a vector clock is represented by $D([S_1, v_1], [S_2, v_2] \ldots, [S_n, v_n])$, where D is a data item, v_i is a version counter, and S_i is a server number.
- If data D is written to server S_i, the system must perform one of the following tasks.
  - Increment v_i if $[S_i, v_i]$ exists.
  - Otherwise, create a new entry $[S_i, 1]$.

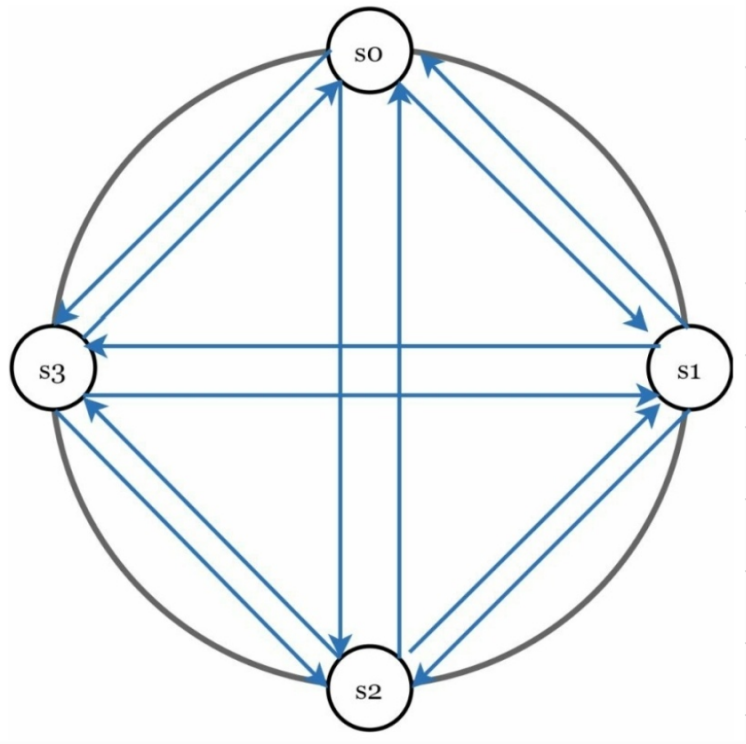


- Using vector clocks, it is easy to tell that a version is an ancestor of another. Eg. - $D([S_0, 1], [S_1, 1])$ is an ancestor of $D([S_0, 1], [S_1, 2])$. (No Conflict Case).
- Similarly, a version can be identified as a sibling of another (Conflict). Eg. - $D([S_0, 1], [S_1, 2])$ and $D([S_0, 2], [S_1, 1])$.

- 2 downside -
  - Vector clocks add complexity to the client because it needs to implement conflict resolution logic.
  - Vector clock pair could grow rapidly. To fix the problem, a threshold is set for a length, and if exceeded, the oldest pair is removed.

- Handling Failures
  - Failures are not only inevitable but common.
  - Handling failure scenarios is very important.

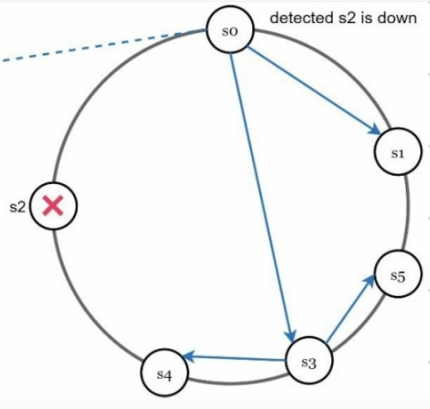
- Failure Detection
  - In a distributed system, it is insufficient to believe that a server is down because another server says so. Usually, it requires atleast two independent sources of information to mark a server down.
  - All-to-all multicasting is a straightforward solution. However, this is inefficient when many servers are in the system.



- A better solution is to use decentralized failure detection methods like gossip protocol. Gossip Protocol works as follows:
  - Each nodes maintains a node membership list, which contains member IDs and heartbeat counter.
  - Each node periodically increments its heartbeat counter.
  - Each node periodically sends heartbeats to a set of random nodes, which in turn propagate to another set of nodes.
  - Once nodes receive heartbeats, membership list is updated to the latest info.
  - If the heartbeat has not increased for more than predefined period, the member is marked offline.

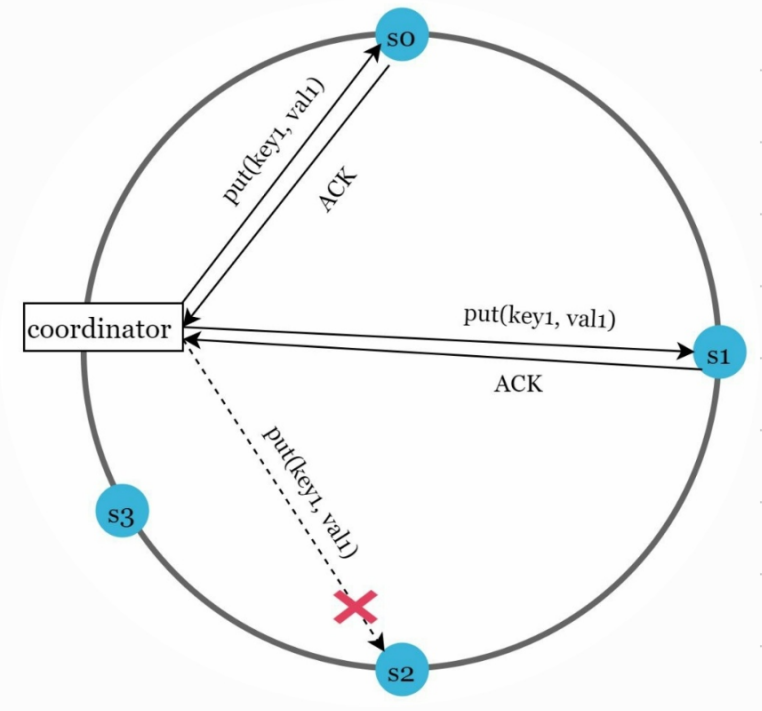
s0's membership list

Member ID	Heartbeat counter	Time
0	10232	12:00:01
1	10224	12:00:10
2	**9908**	**11:58:02**
3	10237	12:00:20
4	10234	12:00:34



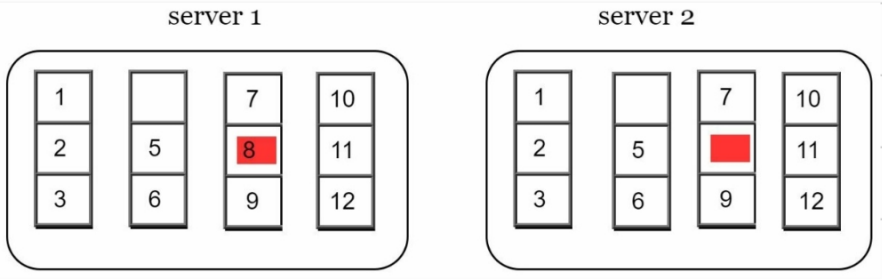
- Node S_0 maintains a node membership list shown on the left side.
- Node S_0 notices that S_2's heartbeat counter has not increased for a long time.
- Node S_0 sends heartbeats that include S_2's info to a set of random nodes. Once other nodes confirm that S_2's heartbeat counter has not been updated for a long time, node S_2 is marked down, and this information is propagated to other nodes.

- Handling Temporary Failures
  - A technique called "sloppy quorum" is used to improve availability. Instead of enforcing the quorum requirement, the system chooses the first W healthy servers for writes and first R healthy servers for reads on the hash 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. The process is called hinted handoff.

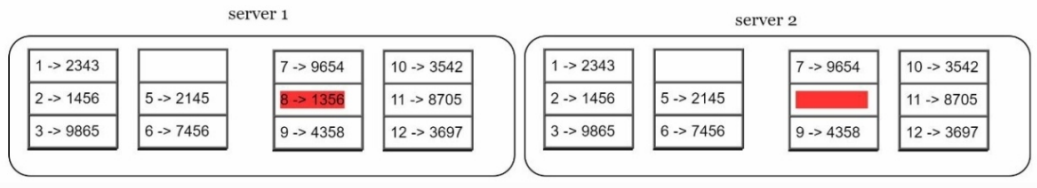


- Handling Permanent Failures
  - Anti-entropy involves comparing each piece of data on replicas and updating each replica to the newest version.
  - A Merkle Tree is used for inconsistency detection and minimizing the amount of data transferred.

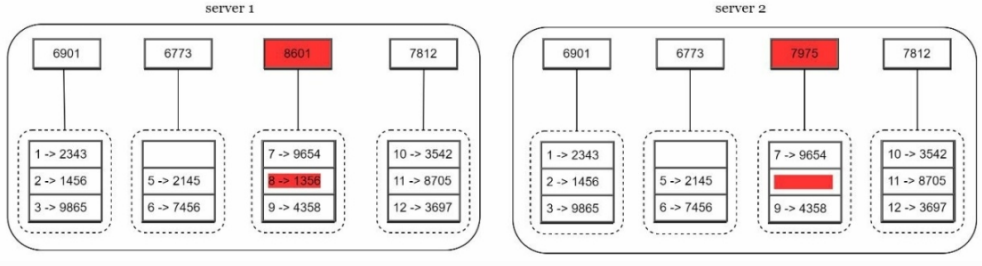
- Assuming key space is from 1 to 12, the following steps show how to build a Merkle Tree:
  - Step 1: Divide key space into buckets. A bucket is used as the root level node to maintain a limited depth of the tree.



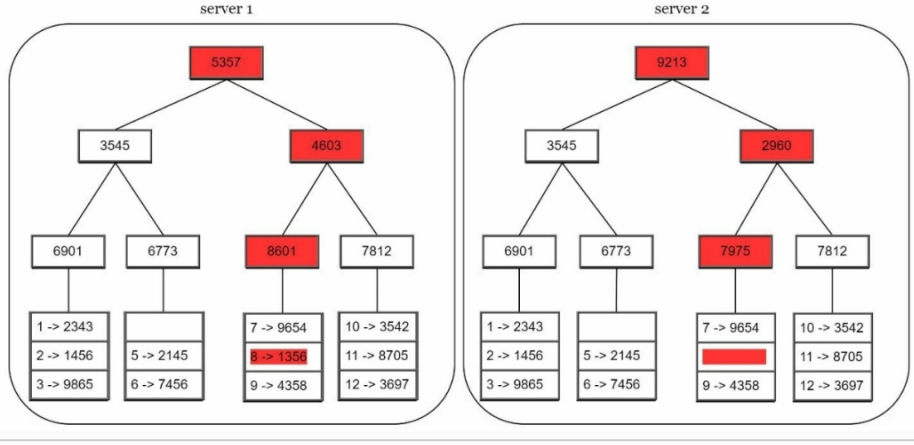
  - Step 2: Once the buckets are created, hash each key in a bucket using a uniform hashing method.



  - Step 3: Create a single hash node per bucket.



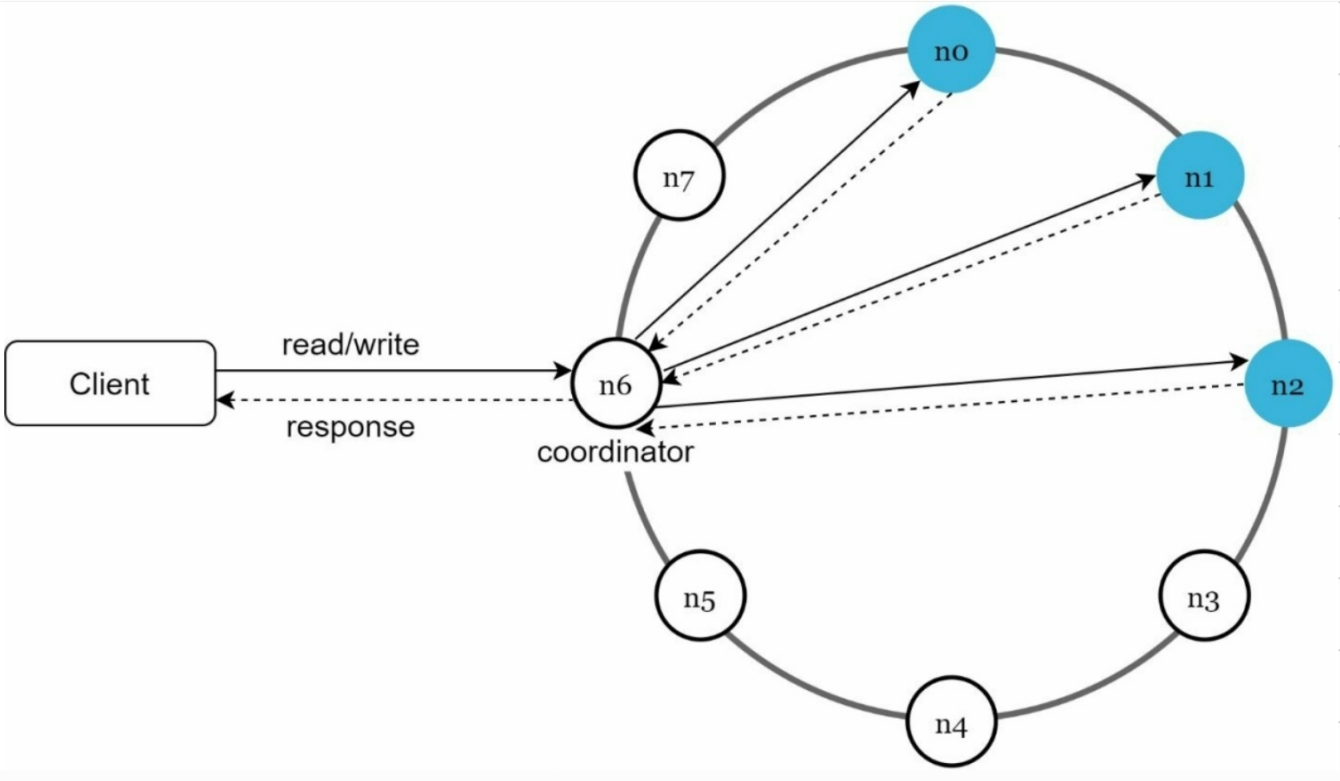
  - Step 4: Build the tree upwards till root by calculating hashes of children



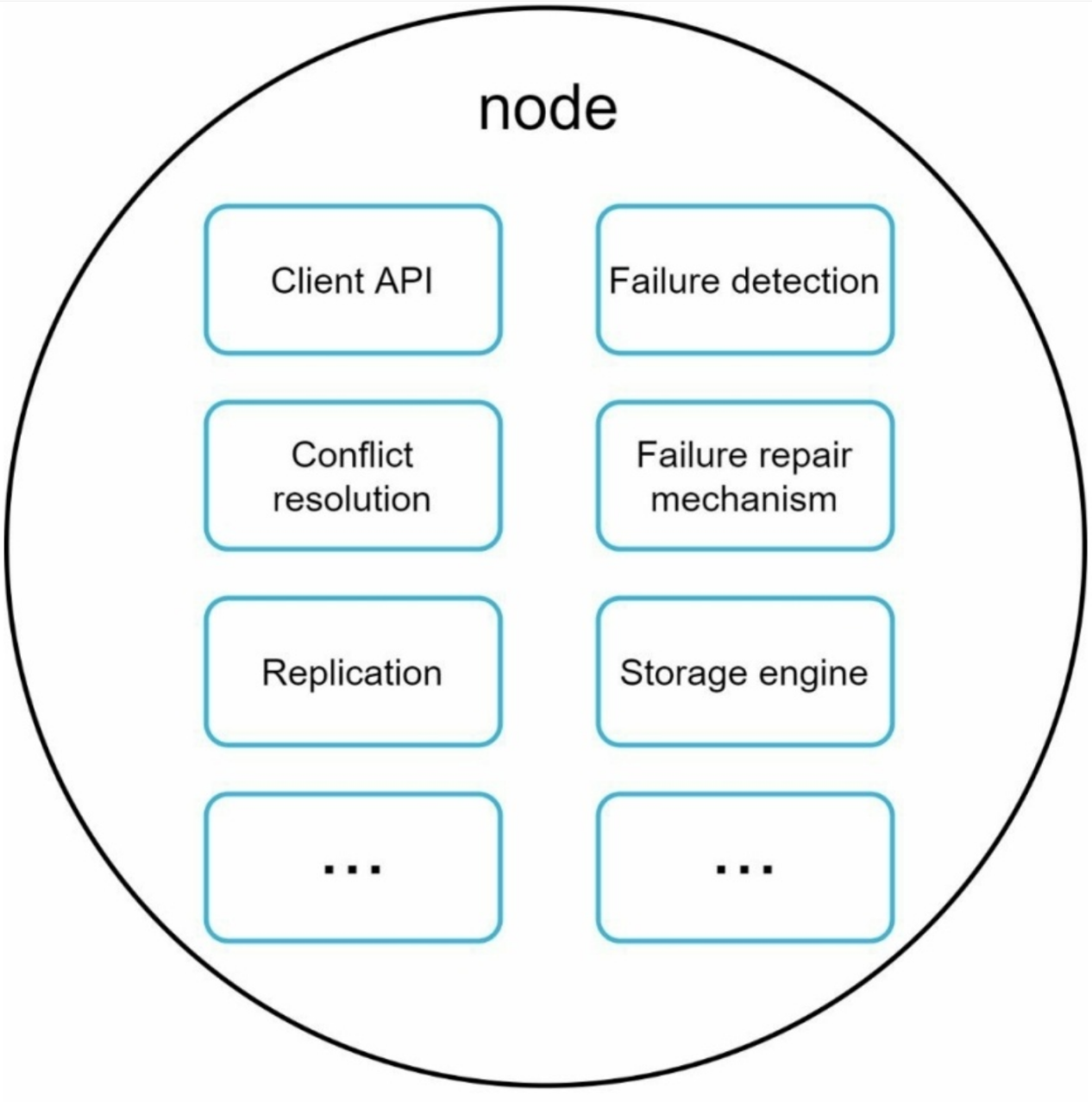
- To compare two Merkle trees, start by comparing the root hashes. If root hashes match, both servers have the same data. If root hashes disagree, then the left child hashes are compared followed by right child hashes. The tree can be traversed to find which buckets are not synchronized.
- Using Merkle trees, the amount of data needed to be synchronized in proportional to the differences between the 2 replicas, and not the amount of data they contain.

- Handling Data Center Outage
  - To build a system capable of handling data center outage, it is important to replicate data across multiple data centers.
  - Even if a data center is completely offline, users can still access data through the other data centers.

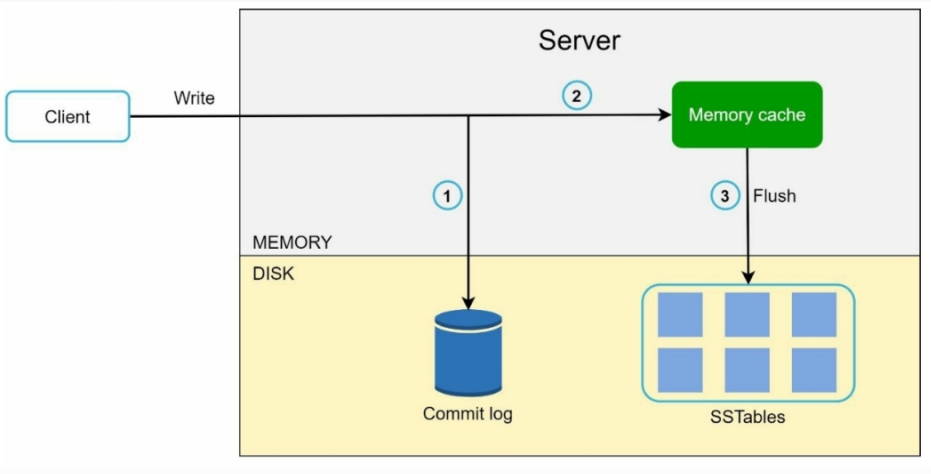
- System Architecture Diagram



- Main Features -
  - Clients communicate with the key-value store through simple APIs: get(key), put(key, value).
  - A coordinator is a node that acts as a proxy between the client and the key-value store.
  - Nodes are distributed on a ring using consistent hashing.
  - The system is completely decentralized so adding and moving nodes can be automatic.
  - Data is replicated at multiple nodes.
  - There is no single point of failure as every node has the same set of responsibility.

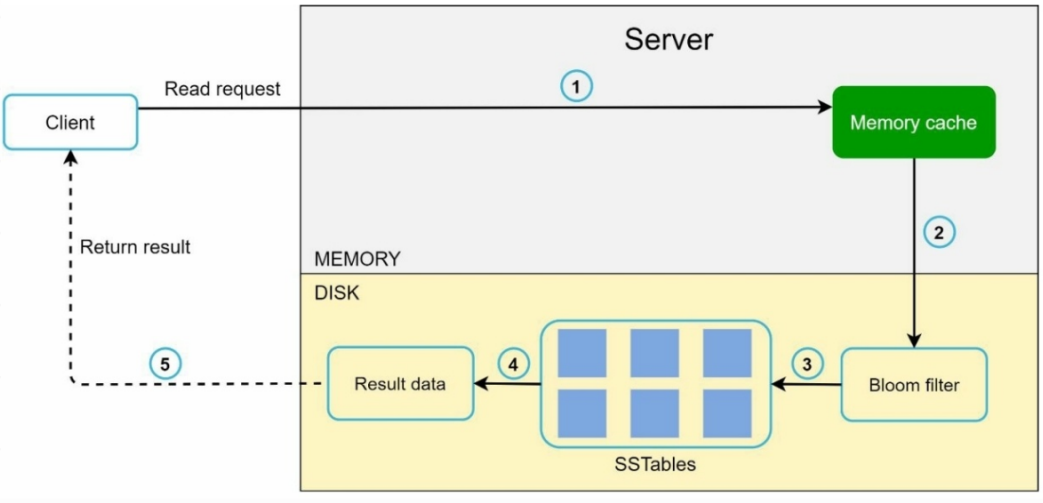


- Write Path



1. The write request is persisted on a commit log file.
2. Data is saved in a memory cache.
3. When the memory cache is full or reaches a predefined threshold, data is flushed to SSTable on disk.

- Read Path



- Data is first checked in memory. If it is present, it is returned to the client.
- If data is not stored in memory -
  1. System checks the bloom list
  2. Bloom filter is use to figure out which SSTable might contain the key.
  3. SSTable returns the result of the dataset.

4. The result of the dataset is returned to the client.

- Summary

  - Ability to store big data : Use consistent hashing to spread load across servers.
  - High availability reads : Data Replication, Multi-Datacenter setup.
  - High available writes : Versioning and conflict resolution with vector clocks.
  - Dataset Partition : Consisted Hashing
  - Incremental Scaling : Consisted Hashing
  - Heterogeneity : Consisted Hashing
  - Tunable Consistency : Quorum Consensus
  - Handling Temporary Failures : Sloppy Quorum and hinted handoff.
  - Handling Permanent Failures : Merkle Tree
  - Handling Data Center Outage : Cross-datacenter Replication