07

Design A Key-value Store

A key-value store, also referred to as a key-value database, is a non-relational database. Each unique identifier is stored as a key with its associated value. This data pairing is known as a “key-value” pair.

In a key-value pair, the key must be unique, and the value associated with the key can be accessed through the key. Keys can be plain text or hashed values. For performance reasons, a short key works better. What do keys look like? Here are a few examples:

- Plain text key: “last_logged_in_at”
- Hashed key: 253DDEC4

The value in a key-value pair can be strings, lists, objects, etc. The value is usually treated as an opaque object in key-value stores, such as Amazon dynamo [1], Memcached [2], Redis [3], etc.

Here is a data snippet in a key-value store:

key	**value**
145	john
147	bob
160	julia

Table 1

In this chapter, you are asked to design a key-value store that supports the following operations:

- put(key, value) // insert “value” associated with “key”

- get(key) // get “value” associated with “key”

Understand the problem and establish design scope

There is no perfect design. Each design achieves a specific balance regarding the tradeoffs of the read, write, and memory usage. Another tradeoff has to be made was between consistency and availability. In this chapter, we design a key-value store that comprises of the following characteristics:

- The size of a 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 data set.
- Automatic scaling: The addition/deletion of servers should be automatic based on traffic.
- Tunable consistency.
- Low latency.

Single server key-value store

Developing a key-value store that resides in a single server is easy. An intuitive approach is to 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. Two 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 (**C**onsistency, **A**vailability, **P**artition Tolerance) theorem.

CAP theorem

CAP theorem states it is impossible for a distributed system to simultaneously provide more than two of these three guarantees: consistency, availability, and partition tolerance. Let us establish a few definitions.

Consistency: consistency means all clients see the same data at the same time no matter which node they connect to.

Availability: 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 two 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 as shown in Figure 1.

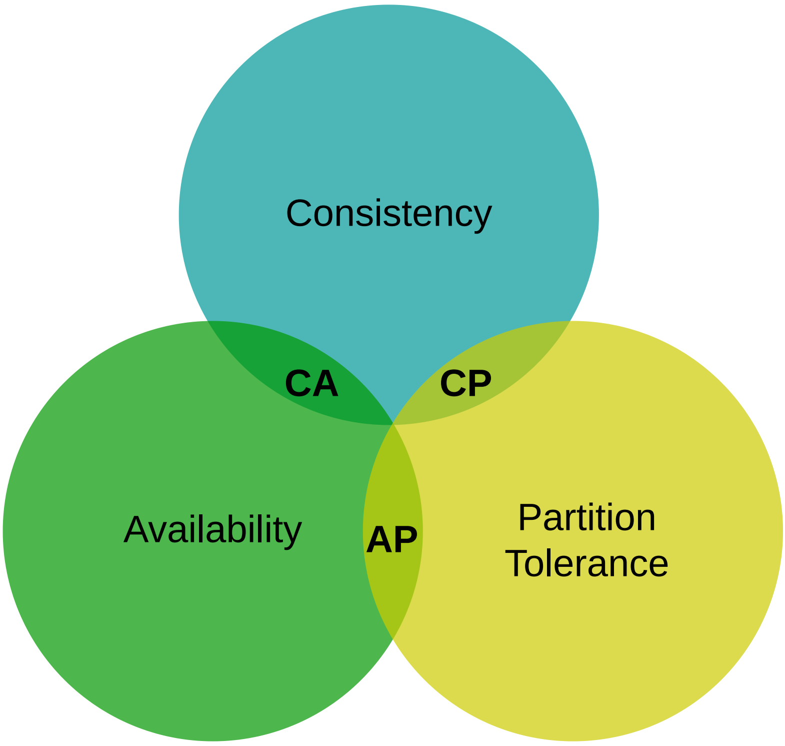

Figure 1

Nowadays, 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: an AP key-value store supports availability and partition tolerance while sacrificing consistency.

CA (consistency and availability) systems: a CA key-value store 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.

What you read above is mostly the definition part. To make it easier to understand, let us take a look at some concrete examples. In distributed systems, data is usually replicated multiple times. Assume data are replicated on three replica nodes, *n1*, *n2* and *n3* as shown in Figure 2.

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 are achieved.

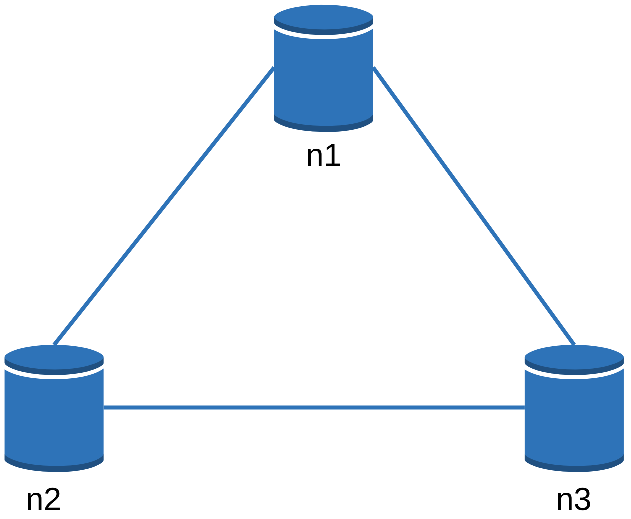

Figure 2

Real-world distributed systems

In a distributed system, partitions cannot be avoided, and when a partition occurs, we must choose between consistency and availability. In Figure 3, *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* and *n2* yet, *n1* and *n2* would have stale data.

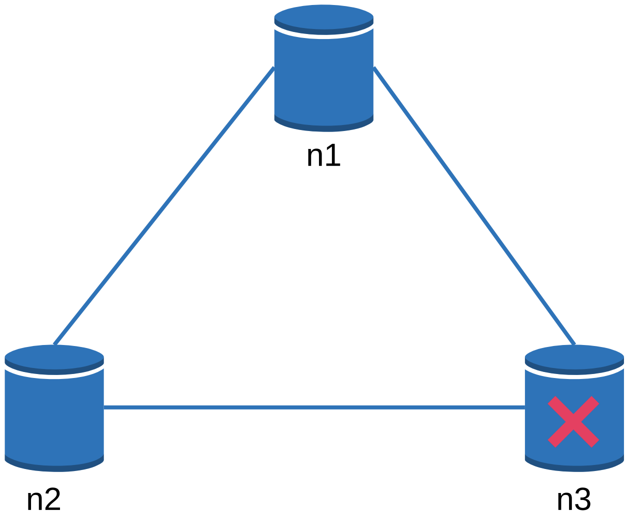

Figure 3

If we choose consistency over availability (CP system), we must block all write operations to *n1* and *n2* to avoid data inconsistency among these three servers, which makes the system unavailable. Bank systems usually have extremely high consistent requirements. For example, it is crucial for a bank system to display the most up-to-date balance info. If inconsistency occurs due to a network partition, the bank system returns an error before the inconsistency is resolved.

However, if we choose 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 to *n3* when the network partition is resolved.

Choosing the right CAP guarantees that fit your use case is an important step in building a distributed key-value store. You can discuss this with your interviewer and design the system accordingly.

System components

In this section, we will discuss the following core components and techniques used to build a key-value store:

- Data partition
- Data replication
- Consistency
- Inconsistency resolution
- Handling failures
- System architecture diagram
- Write path
- Read path

The content below is largely based on three popular key-value store systems: Dynamo [4], Cassandra [5], and BigTable [6].

Data partition

For large applications, it is infeasible 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. There are two challenges while partitioning the data:

- Distribute data across multiple servers evenly.
- Minimize data movement when nodes are added or removed.

Consistent hashing discussed in the previous chapter is a great technique to solve these problems. Let us revisit how consistent hashing works at a high-level.

- First, servers are placed on a hash ring. In Figure 4, eight servers, represented by *s0, s1, …, s7*, are placed on the hash ring.
- Next, a key is hashed onto the same ring, and it is stored on the first server encountered while moving in the clockwise direction. For instance, *key0* is stored in *s1* using this logic.

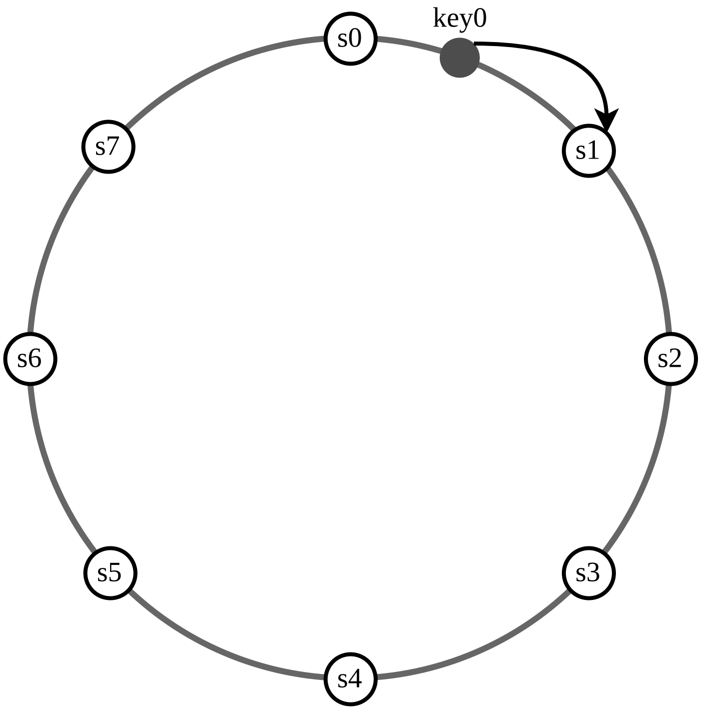

Figure 4

Using consistent hashing to partition data has the following advantages:

Automatic scaling: servers could be added and removed automatically depending on the load.

Heterogeneity: the number of virtual nodes for a server is proportional to the server capacity. For example, servers with higher capacity are assigned with more virtual nodes.

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 and choose the first *N* servers on the ring to store data copies. In Figure 5 (*N = 3*), *key0* is replicated at *s1, s2,* and *s3*.

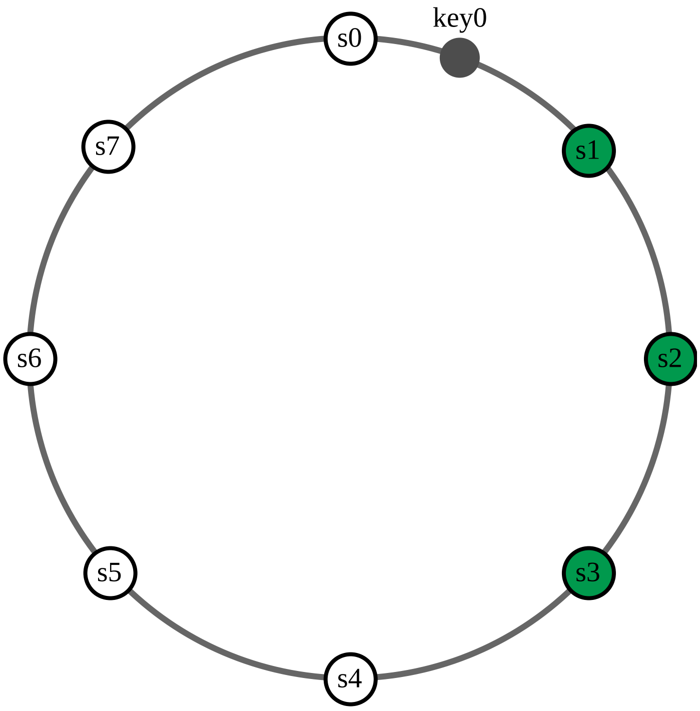

Figure 5

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 outages, network issues, natural disasters, etc. For better reliability, replicas are placed in distinct data centers, and data centers are connected through high-speed networks.

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. Let us establish a few definitions first.

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 *R*. For a read operation to be considered as successful, read operation must wait for responses from at least *R* replicas.

Consider the following example shown in Figure 6 with *N = 3*.

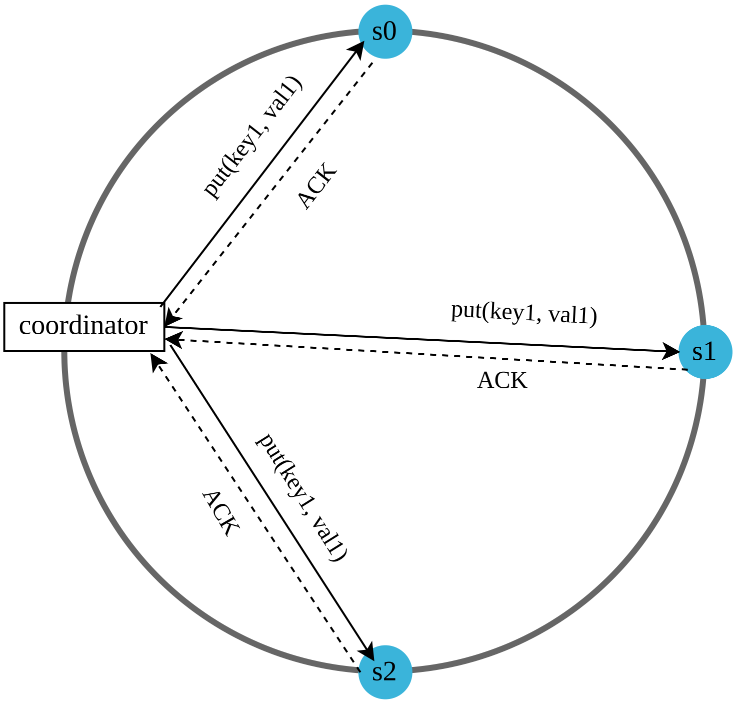

Figure 6 (ACK = acknowledgement)

W = 1 does not mean data is written on one server. For instance, with the configuration in Figure 6, data is replicated at *s0*, *s1,* and *s2*. *W = 1* means that the coordinator must receive at least one acknowledgment before the write operation is considered as successful. For instance, if we get an acknowledgment from *s1*, we no longer need to wait for acknowledgements from *s0* and *s2*. A coordinator acts as a proxy between the client and the nodes.

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 because a coordinator only needs to wait for a response from any of the replicas. If *W* or *R > 1*, the system offers better consistency; however, the query will be slower because the coordinator must wait for the response from the slowest replica.

If *W + R > N*, strong consistency is guaranteed because there must be at least one overlapping node that has the latest data to ensure consistency.

How to configure *N, W*, and *R* to fit our use cases? Here are some of the possible setups:

If *R = 1* and *W = N*, the system is optimized for a fast read.

If *W = 1 and R = N*, the system is optimized for fast write.

If *W + R > N*, strong consistency is guaranteed (Usually *N = 3, W = R = 2*).

If *W + R <= N*, strong consistency is not guaranteed.

Depending on the requirement, we can tune the values of *W, R, N* to achieve the desired level of consistency.

Consistency models

Consistency model is other important factor to consider when designing a key-value store. A consistency model defines the degree of data consistency, and a wide spectrum of possible consistency models exist:

- 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 of weak consistency. Given enough time, all updates are propagated, and all replicas are consistent.

Strong consistency is usually achieved by forcing a replica not to accept new reads/writes until every replica has agreed on current write. This approach is not ideal for highly available systems because it could block new operations. Dynamo and Cassandra adopt eventual consistency, which is our recommended consistency model for our key-value store. From concurrent writes, eventual consistency allows inconsistent values to enter the system and force the client to read the values to reconcile. The next section explains how reconciliation works with versioning.

Inconsistency resolution: versioning

Replication gives high availability but causes inconsistencies among replicas. Versioning and vector locks are used to solve inconsistency problems. Versioning means treating each data modification as a new immutable version of data. Before we talk about versioning, let us use an example to explain how inconsistency happens:

As shown in Figure 7, both replica nodes *n1* and *n2* have the same value. Let us call this value the original *value. Server 1* and *server 2* get the same value for *get("name")* operation.

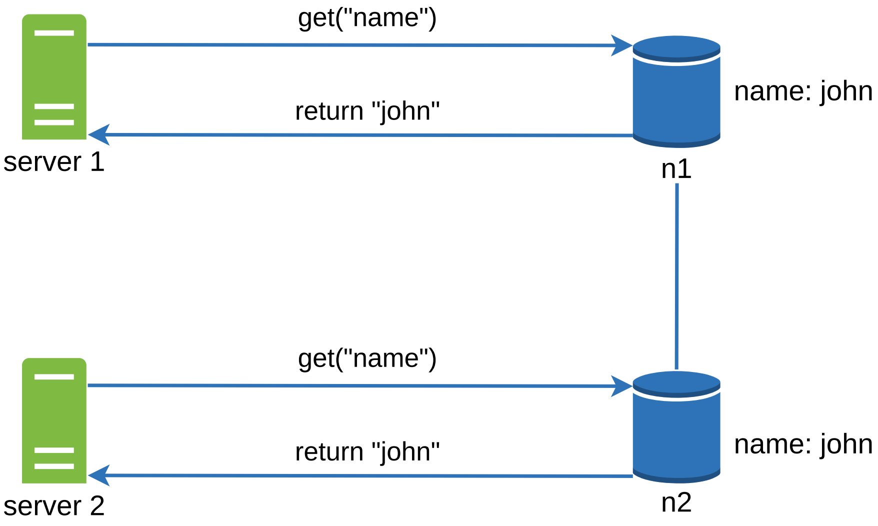

Figure 7

Next, *server 1* changes the name to "johnSanFrancisco", and *server 2* changes the name to "johnNewYork" as shown in Figure 8. These two changes are performed simultaneously. Now, we have conflicting values, called versions *v1* and *v2*.

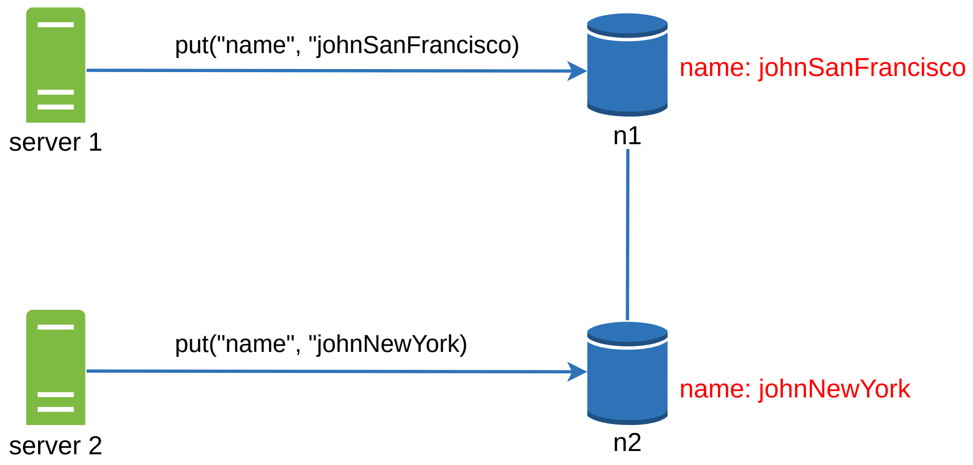

Figure 8

In this example, the original value could be ignored because the modifications were based on it. However, there is no clear way to resolve the conflict of the last two versions. 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. Let us examine how vector clocks work.

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([S1, v1], [S2, v2], …, [Sn, vn])*, where *D* is a data item, *v1* is a version counter, and *s1* is a server number, etc. If data item *D* is written to server *Si*, the system must perform one of the following tasks.

- Increment *vi* if *[Si, vi]* exists.
- Otherwise, create a new entry *[Si, 1]*.

The above abstract logic is explained with a concrete example as shown in Figure 9.

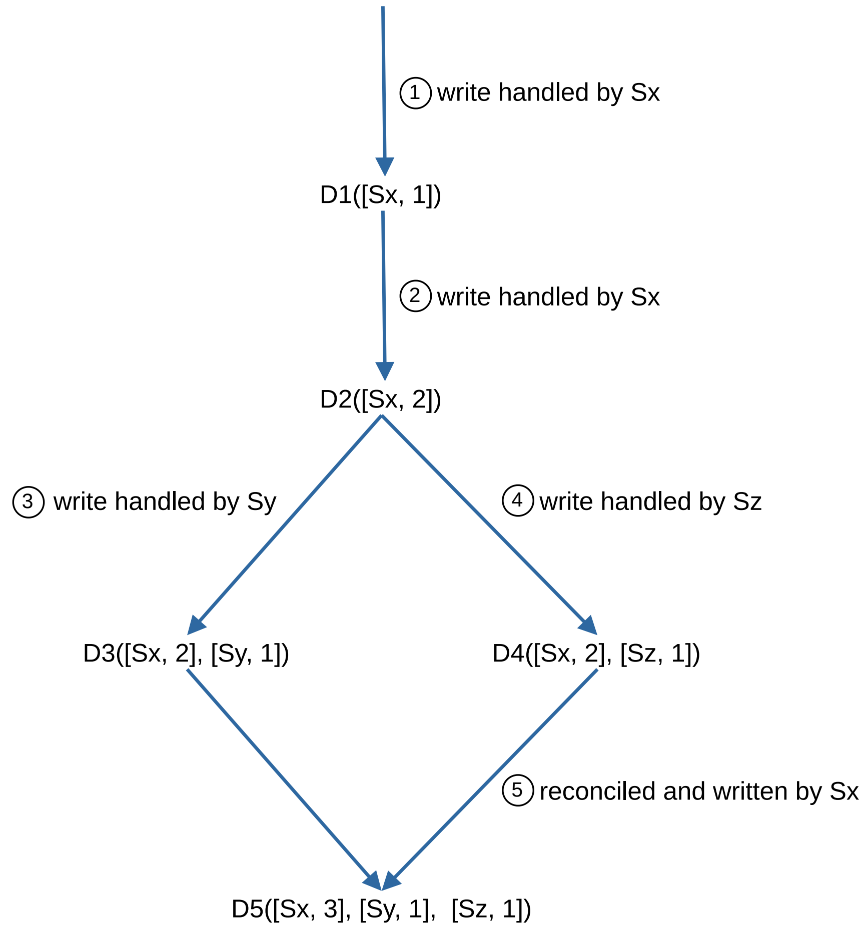

Figure 9

1. A client writes a data item *D1* to the system, and the write is handled by server *Sx*, which now has the vector clock *D1[(Sx, 1)]*.

2. Another client reads the latest *D1*, updates it to *D2*, and writes it back. *D2* descends from *D1* so it overwrites *D1*. Assume the write is handled by the same server *Sx*, which now has vector clock *D2([Sx, 2])*.

3. Another client reads the latest *D2*, updates it to *D3*, and writes it back. Assume the write is handled by server *Sy*, which now has vector clock *D3([Sx, 2], [Sy, 1]))*.

4. Another client reads the latest *D2*, updates it to *D4*, and writes it back. Assume the write is handled by server *Sz*, which now has *D4([Sx, 2], [Sz, 1]))*.

5. When another client reads *D3* and *D4*, it discovers a conflict, which is caused by data item *D2* being modified by both *Sy* and *Sz*. The conflict is resolved by the client and updated data is sent to the server. Assume the write is handled by *Sx*, which now has *D5([Sx, 3], [Sy, 1], [Sz, 1])*. We will explain how to detect conflict shortly.

Using vector clocks, it is easy to tell that a version *X* is an ancestor (i.e. no conflict) of version *Y* if the version counters for each participant in the vector clock of *Y* is greater than or equal to the ones in version *X*. For example, the vector clock *D([s0, 1], [s1, 1])]* is an ancestor of *D([s0, 1], [s1, 2])*. Therefore, no conflict is recorded.

Similarly, you can tell that a version *X* is a sibling (i.e., a conflict exists) of *Y* if there is any participant in *Y*'s vector clock who has a counter that is less than its corresponding counter in *X*. For example, the following two vector clocks indicate there is a conflict: *D([s0, 1], [s1, 2])* and *D([s0, 2], [s1, 1]).*

Even though vector clocks can resolve conflicts, there are two notable downsides. First, vector clocks add complexity to the client because it needs to implement conflict resolution logic.

Second, the *[server: version]* pairs in the vector clock could grow rapidly. To fix this problem, we set a threshold for the length, and if it exceeds the limit, the oldest pairs are removed. This can lead to inefficiencies in reconciliation because the descendant relationship cannot be determined accurately. However, based on Dynamo paper [4], Amazon has not yet encountered this problem in production; therefore, it is probably an acceptable solution for most companies.

Handling failures

As with any large system at scale, failures are not only inevitable but common. Handling failure scenarios is very important. In this section, we first introduce techniques to detect failures. Then, we go over common failure resolution strategies.

Failure detection

In a distributed system, it is insufficient to believe that a server is down because another server says so. Usually, it requires at least two independent sources of information to mark a server down.

As shown in Figure 10, all-to-all multicasting is a straightforward solution. However, this is inefficient when many servers are in the system.

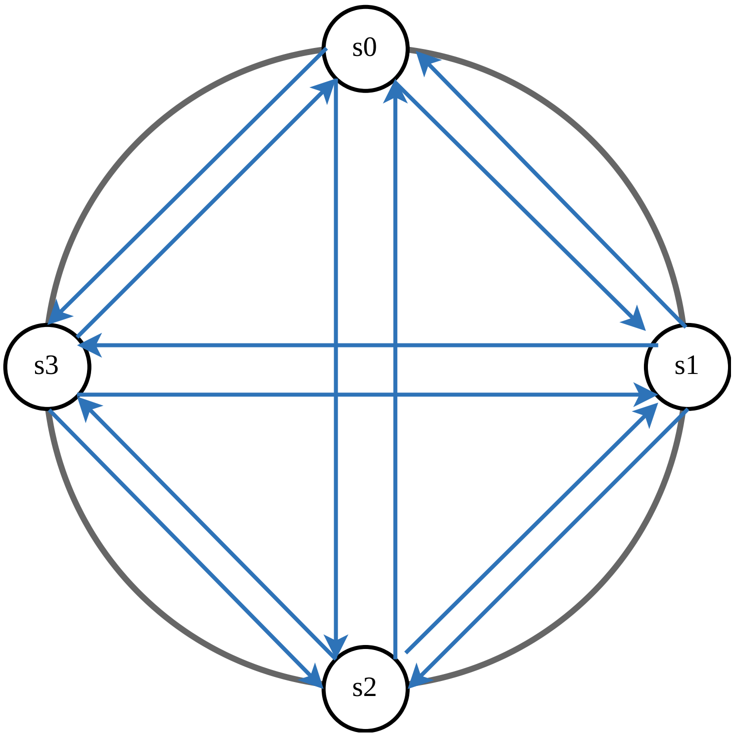

Figure 10

A better solution is to use decentralized failure detection methods like gossip protocol. Gossip protocol works as follows:

- Each node maintains a node membership list, which contains member IDs and heartbeat counters.
- 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 periods, the member is considered as 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

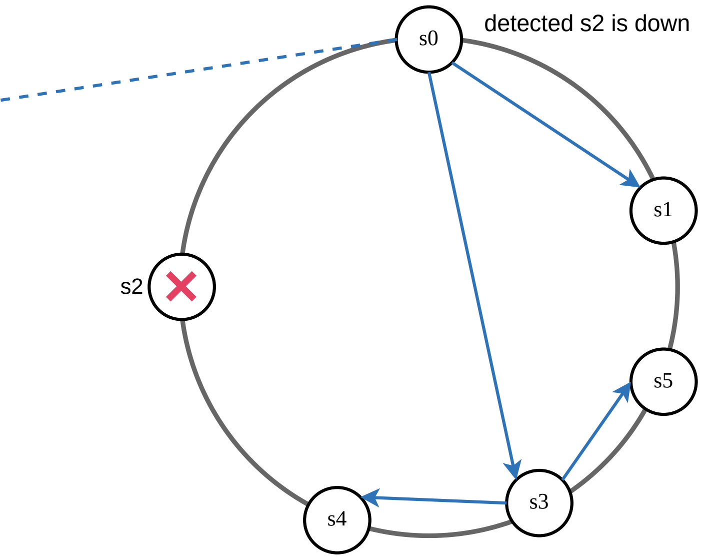

Figure 11

As shown in Figure 11:

- Node *s0* maintains a node membership list shown on the left side.
- Node *s0* notices that node s2’s (member ID = 2) heartbeat counter has not increased for a long time.
- Node *s0* sends heartbeats that include *s2*’s info to a set of random nodes. Once other nodes confirm that *s2*’s heartbeat counter has not been updated for a long time, node *s2* is marked down, and this information is propagated to other nodes.

Handling temporary failures

After failures have been detected through the gossip protocol, the system needs to deploy certain mechanisms to ensure availability. In the strict quorum approach, read and write operations could be blocked as illustrated in the quorum consensus section.

A technique called “sloppy quorum” [4] 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. This process is called hinted handoff. Since *s2* is unavailable in Figure 12, reads and writes will be handled by *s3* temporarily. When *s2* comes back online, *s3* will hand the data back to *s2*.

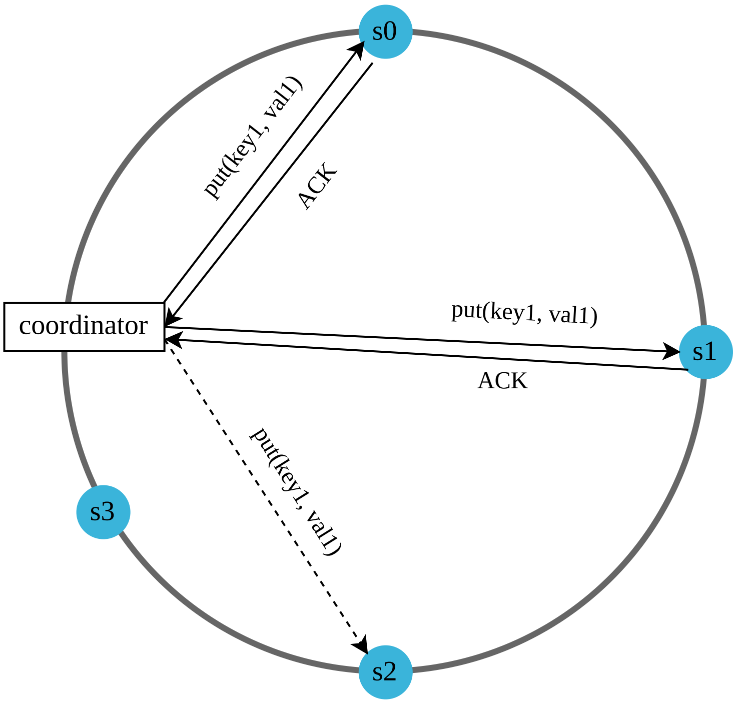

Figure 12

Handling permanent failures

Hinted handoff is used to handle temporary failures. What if a replica is permanently unavailable? To handle such a situation, we implement an anti-entropy protocol to keep replicas in sync. 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.

Quoted from Wikipedia [7]: “A hash tree or Merkle tree is a tree in which every non-leaf node is labeled with the hash of the labels or values (in case of leaves) of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures”.

Assuming key space is from 1 to 12, the following steps show how to build a Merkle tree. Highlighted boxes indicate inconsistency.

Step 1: Divide key space into buckets (4 in our example) as shown in Figure 13. A bucket is used as the root level node to maintain a limited depth of the tree.

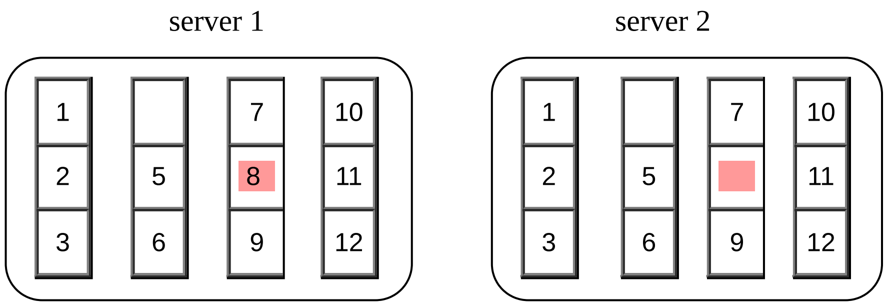

Figure 13

Step 2: Once the buckets are created, hash each key in a bucket using a uniform hashing method (Figure 14).

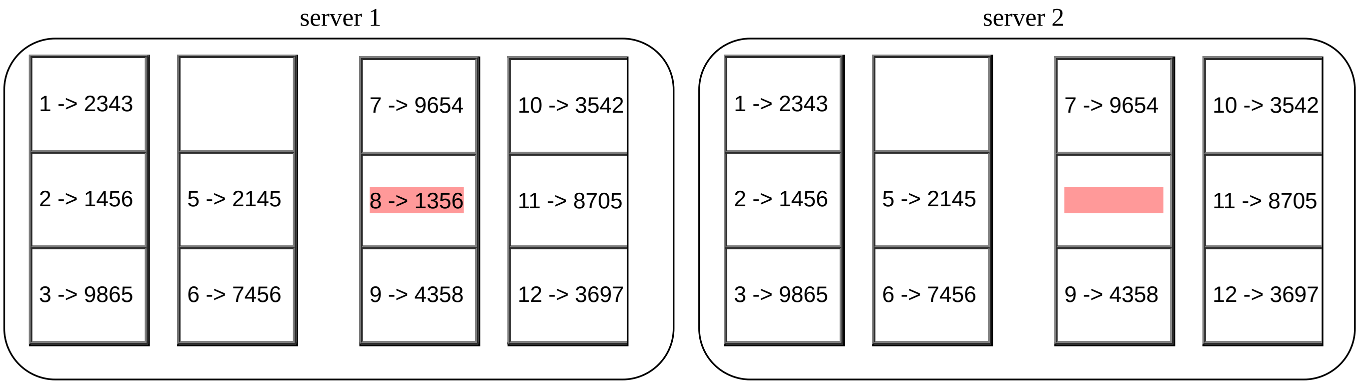

Figure 14

Step 3: Create a single hash node per bucket (Figure 15).

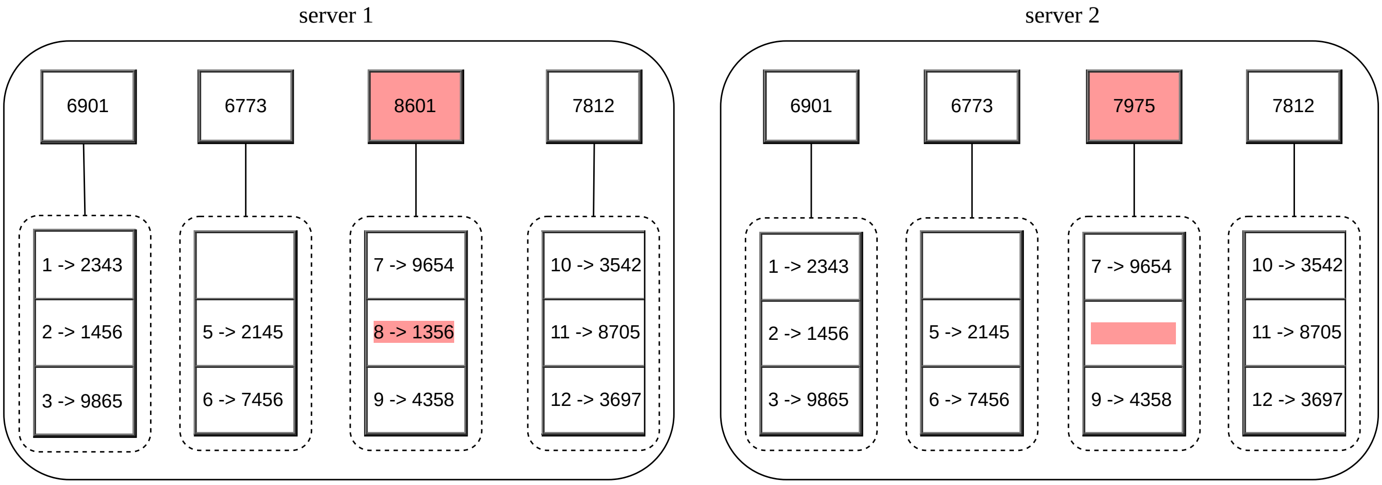

Figure 15

Step 4: Build the tree upwards till root by calculating hashes of children (Figure 16).

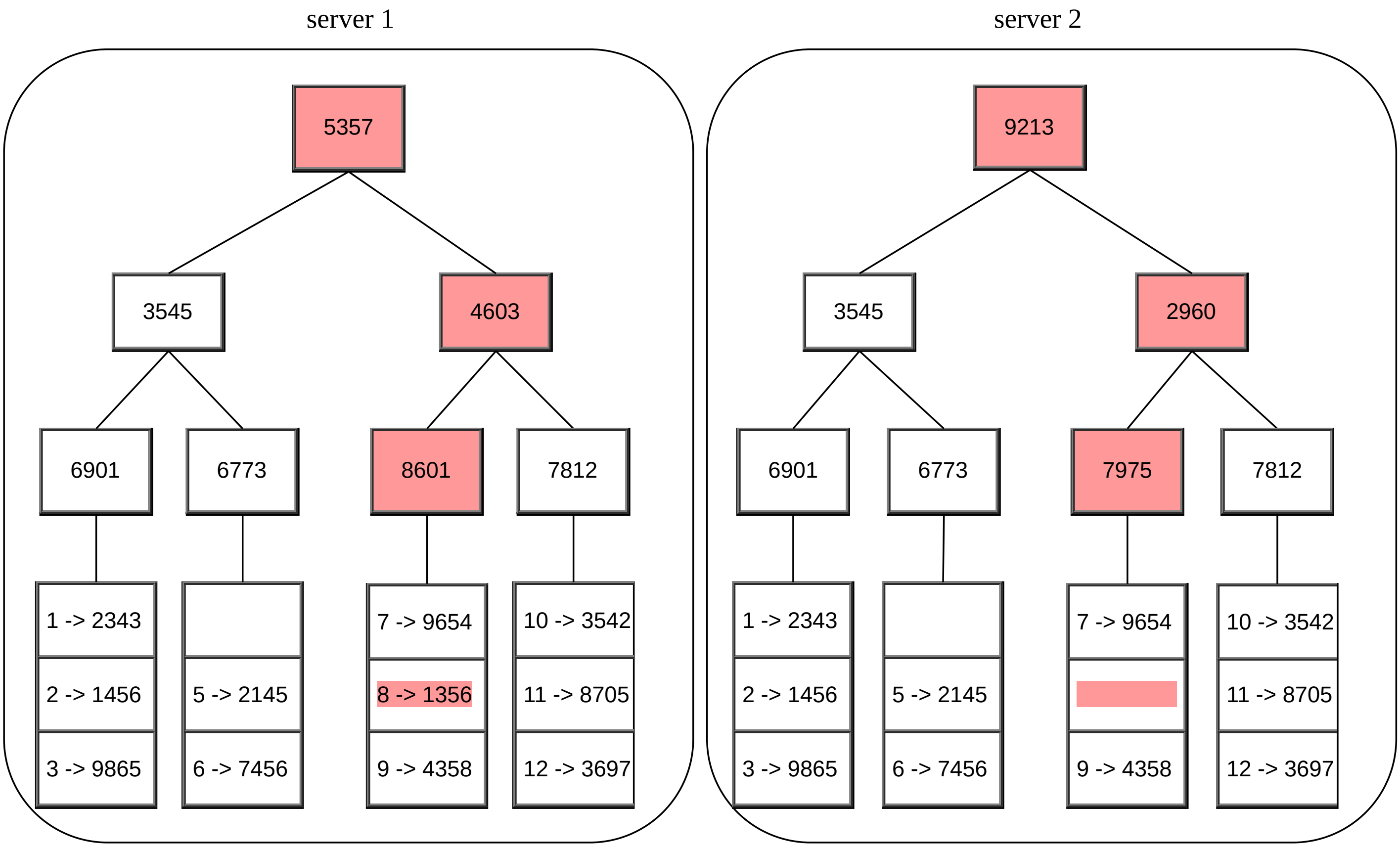

Figure 16

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. You can traverse the tree to find which buckets are not synchronized and synchronize those buckets only.

Using Merkle trees, the amount of data needed to be synchronized is proportional to the differences between the two replicas, and not the amount of data they contain. In real-world systems, the bucket size is quite big. For instance, a possible configuration is one million buckets per one billion keys, so each bucket only contains 1000 keys.

Handling data center outage

Data center outage could happen due to power outage, network outage, natural disaster, etc. 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

Now that we have discussed different technical considerations in designing a key-value store, we can shift our focus on the architecture diagram, shown in Figure 17.

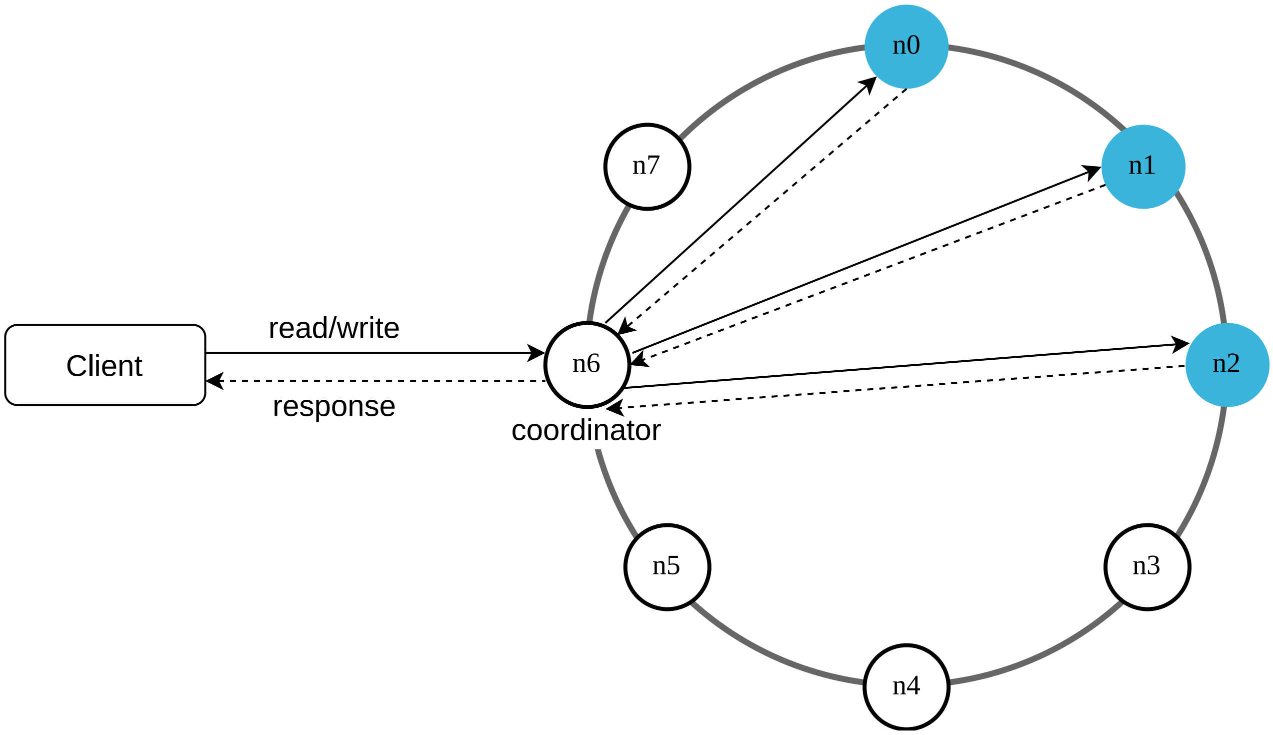

Figure 17

Main features of the architecture are listed as follows:

- Clients communicate with the key-value store through simple APIs: *get(key)* and *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 responsibilities.

As the design is decentralized, each node performs many tasks as presented in Figure 18.

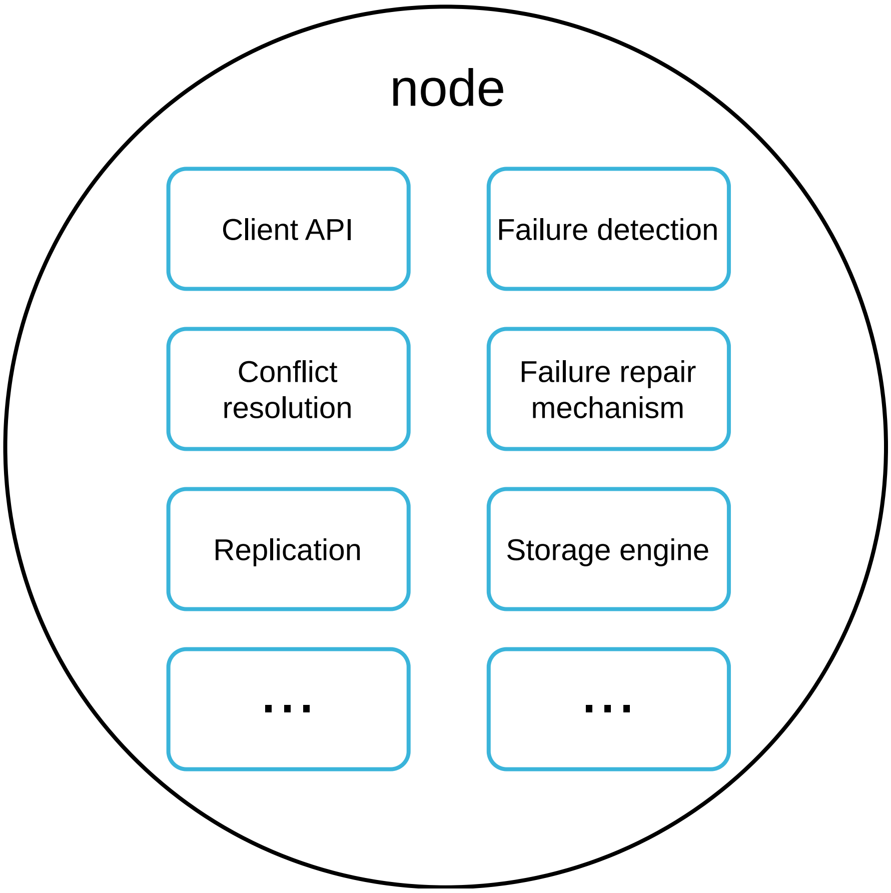

Figure 18

Write path

Figure 19 explains what happens after a write request is directed to a specific node. Please note the proposed designs for write/read paths are primary based on the architecture of Cassandra [8].

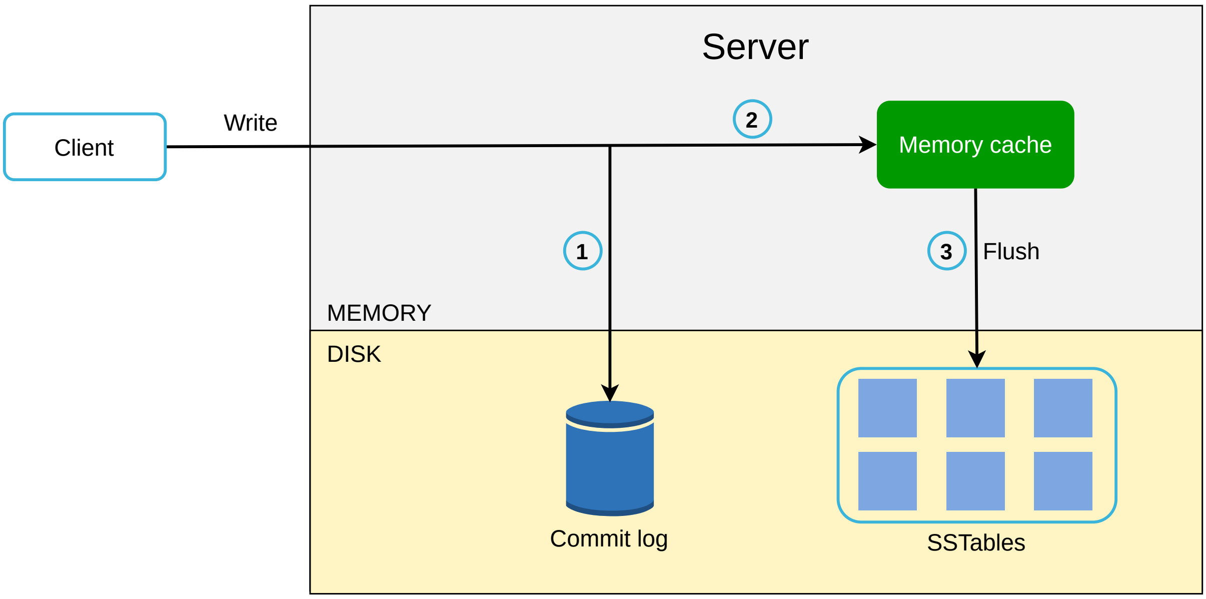

Figure 19

1. The write request is persisted on a commit log file.

2. Data is saved in the memory cache.

3. When the memory cache is full or reaches a predefined threshold, data is flushed to SSTable [9] on disk. Note: A sorted-string table (SSTable) is a sorted list of <key, value> pairs. For readers interested in learning more about SStable, refer to the reference material [9].

Read path

After a read request is directed to a specific node, it first checks if data is in the memory cache. If so, the data is returned to the client as shown in Figure 20.

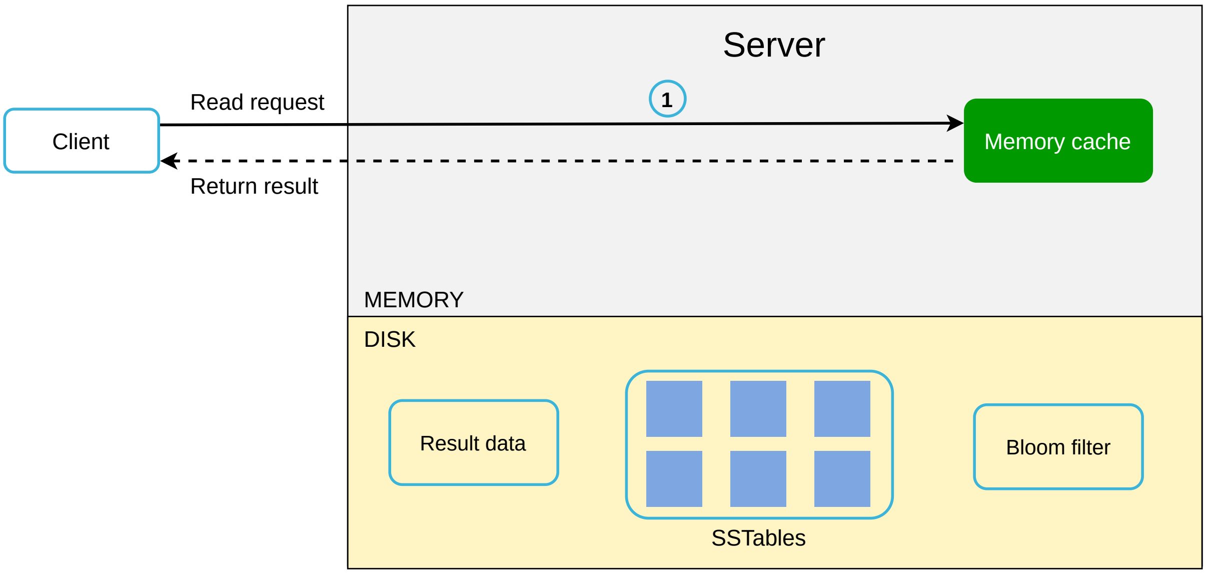

Figure 20

If the data is not in memory, it will be retrieved from the disk instead. We need an efficient way to find out which SSTable contains the key. Bloom filter [10] is commonly used to solve this problem.

The read path is shown in Figure 21 when data is not in memory.

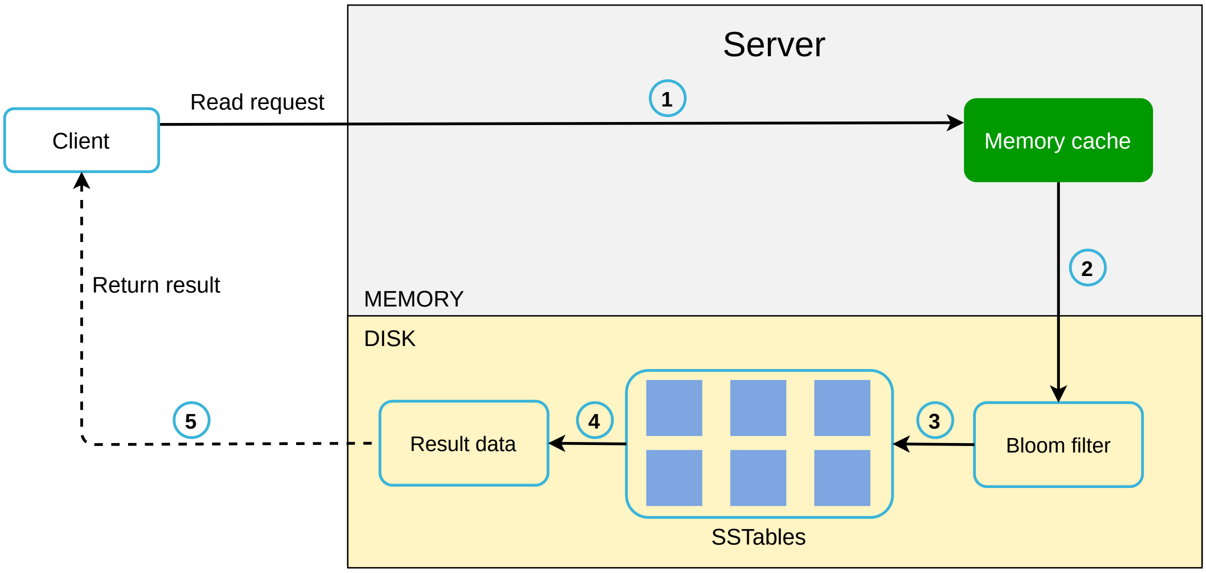

Figure 21

1. The system first checks if data is in memory. If not, go to step 2.

2. If data is not in memory, the system checks the bloom filter.

3. The bloom filter is used to figure out which SSTables might contain the key.

4. SSTables return the result of the data set.

5. The result of the data set is returned to the client.

Summary

This chapter covers many concepts and techniques. To refresh your memory, the following table summarizes features and corresponding techniques used for a distributed key-value store.

Goal/Problems	**Technique**
Ability to store big data	Use consistent hashing to spread load across servers
High availability reads	Data replication Multi-datacenter setup
Highly available writes	Versioning and conflict resolution with vector clocks
Dataset partition	Consistent Hashing
Incremental scalability	Consistent Hashing
Heterogeneity	Consistent 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

Table 2

Reference materials

[1] Amazon DynamoDB: https://aws.amazon.com/dynamodb/

[2] memcached: https://memcached.org/

[3] Redis: https://redis.io/

[4] Dynamo: Amazon’s Highly Available Key-value Store: https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[5] Cassandra: https://cassandra.apache.org/

[6] Bigtable: A Distributed Storage System for Structured Data: https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

[7] Merkle tree: https://en.wikipedia.org/wiki/Merkle_tree

[8] Cassandra architecture: https://cassandra.apache.org/doc/latest/architecture/

[9] SStable: https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb/

[10] Bloom filter https://en.wikipedia.org/wiki/Bloom_filter