Design a Key-Value Store

Image Courtesy: https://www.singlestore.com/blog/how-to-use-singlestore-as-a-key-value-store/

Contents Covered:

Prologue for our understanding, need not be discussed with the interviewer:

What is the difference between a key-value database and a key-value cache?

1. Where a key-value database can be used as a database to persist data, caches are used in conjunction with a database when there is a need to increase read performance.

2. Caches are not used to enhance write or update performance yet key-value databases are very effective.

3. Where key-value databases can be resilient to server failure, caches are stored in RAM and cannot provide you with transactional guarantees if the server crashes.

Properties of Key-Value Database

Most key-value storage systems seek the following properties.

Consistency: Most key-value database systems claim to be “eventually consistent,” which means that data updates must be propagated across all nodes.

Throughput: The throughput statistic, which focuses on the number of transactions that can be performed in a given unit of time, is one of the fundamental goals of database systems.

Scalability: We say a key-value database system satisfies the scalability criteria if it can distribute data and a load of read/write operations across multiple servers, potentially increasing the system’s performance. The term “horizontal scalability” is used here. Some systems take into account “vertical scalability,” in which many CPUs share RAM and storage. One essential strategy for ensuring vertical scalability is to keep data in RAM rather than on a disc.

Availability: The data must be available to the user at any given time.

Partition Tolerance: Data must be stored in several places. The system can handle failures of some individual servers that can't be contacted because of distribution and continue to function as a whole, ensuring that earlier functionalities are maintained.

When to use a key-value database?

We can use this database in the following scenarios:

Session management: One of the best examples is multiplayer online games, where we use a key-value database to manage each player's session.

In-memory data caching: We can it as an effective cache mechanism for frequently accessed but rarely updated data to accelerate application responses.

User preference storage and personalization: We can use it to provide product recommendations and store personalized lists of items for individual customers. One of the best examples is e-commerce.

Implementing blockchain-based solutions: This database is common when implementing blockchain-based ideas. Here the key is a hash value, and the value is the corresponding block entity.

Real-time random data access: Key-value databases offer fast in-memory access. So we can use it in a situation when our application needs to handle lots of small continuous reads and writes.

Storing basic information: We can use it in an application to keep some basic data based on key-value pairs. For example, we can store web pages with the URL as the key and the webpage as the value.

Content Delivery Networks: We can use it for multimedia storage and serve large web objects (video, images, audio, etc.) in a high availability environment.

Really understand the problem before designing

1. Are we thinking to design a key-value database like Dynamo or a key-value in-memory database/cache like Redis? It's the other way to ask whether we need a highly persistent system. (As Redis lacks in persistence, will discuss more on this.)
2. What is the amount of data DB should accommodate?
3. What should be the maximum size of value? Can a value be so big that it does not fit on a single machine?
4. It would be a read-heavy system or a write-heavy system?
5. Our DB should fall under which category of CAP theorem, AP or CP?
6. What would the estimated queries per second (QPS) be for this DB?

Functional Requirements:

- Set a key with a value (Update the value if it already exists).
- Get the value given by the key.
- Delete the key value (Unset the key).

Non Functional Requirements:

- Highly Scalable (Should be flexible to increase instances at runtime during load increase)
- Consistent (Should provide consistent and correct response on each call).
- Durable (No data should be lost during network partition failures).
- Availability (No single point of failure).

How would we design a simple key-value storage system in a single machine?

The simplest straightforward solution is to store key-value pairs in a hash table, which is how most of these systems currently function. A hash table is a type of data structure that allows us to read and write a key-value pair in real-time and is very simple to use. This is something that most languages have built-in support for.

Something similar is already there in Leetcode which can be a coding interview question:

https://leetcode.com/problems/time-based-key-value-store/

However, there is a disadvantage. We must normally store everything in memory when using a hash table, which is not always practicable when the data set is large. Two options are commonly used:

Compressing the data: This should be the first thing to consider, as there is frequently a lot of information that can be compressed. We can, for example, save references rather than actual data. Instead of float64, we can use float32. Furthermore, other data representations such as bit arrays (integer) or vectors can be useful.

Storing in a disk: We can save a portion of the data on the disk if it's difficult to put everything in memory. We can think of the system as a caching system to further optimize it. The data that is often accessed is kept in memory, while the remainder is stored on a disk.

Distributed key-value storage:

Because a single machine cannot store all of the data, the general idea is to divide the data among numerous machines (partitioning) according to some rules, with a coordinator machine directing clients to the machine that has the necessary resources.

What is the minimum number of machines required to store the data?

As data to be stored is 100 TB, and each record can be as large as 1 GB. Let's assume the size hard disk for each of the storage machines is 10TB. Thus the minimum number of machines required will be 10. This is without any replication, so if we go for data replication, or sharding if we go for lowering the number of QPS on each machine; Thus, total storage might increase significantly as well as the number of machines required.

Through the Microscopic Lens of System Design

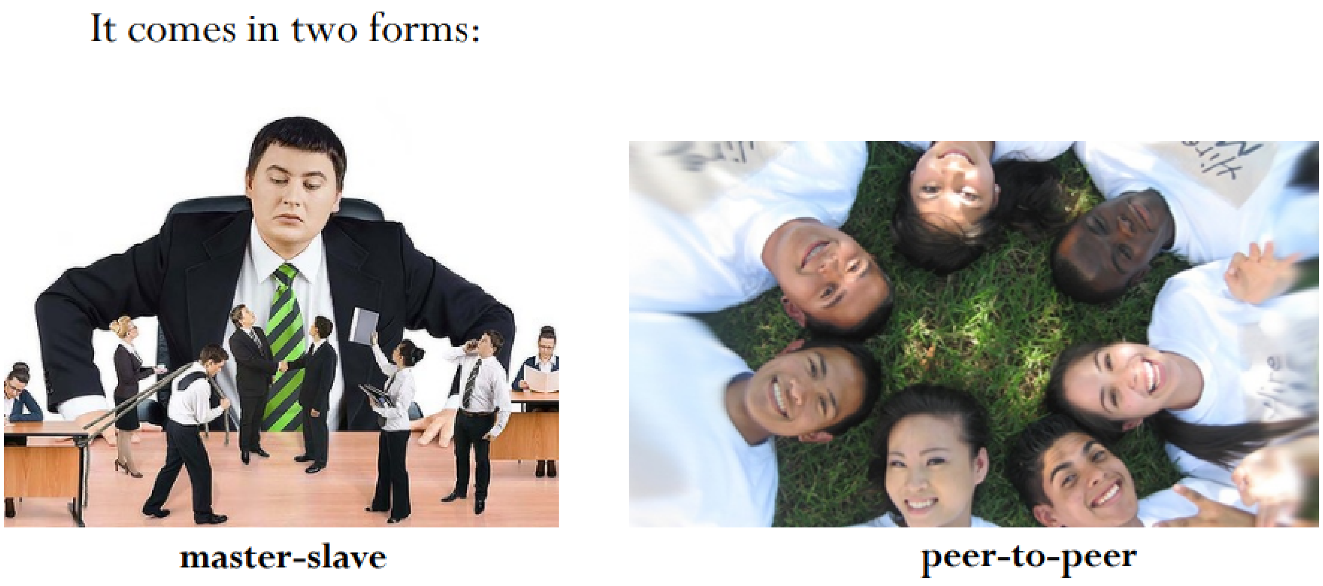
What are the challenges in a distributed database system?

1. **Replication**: When data is stored on a server and there is a server crash, how to cope with data loss?

2. **Partitioning**: How to handle when single DB storage is not enough?
3. **Failure Detection and Handling Temporary failures**: How to ensure high availability and durability when some nodes/replicas are down?
4. **High Availability for Reads in a Read intensive system:** Irrespective of which system we are in, network partition is bound to happen, in that case how our system can be highly available for reads?
5. **High Availability for writes in a write-intensive system**: Irrespective of which system we are in, network partition is bound to happen, in that case how our system can be highly available for writes?
6. **Recovering from Permanent failures**: How to synchronize with divergent replicas?

Replication:

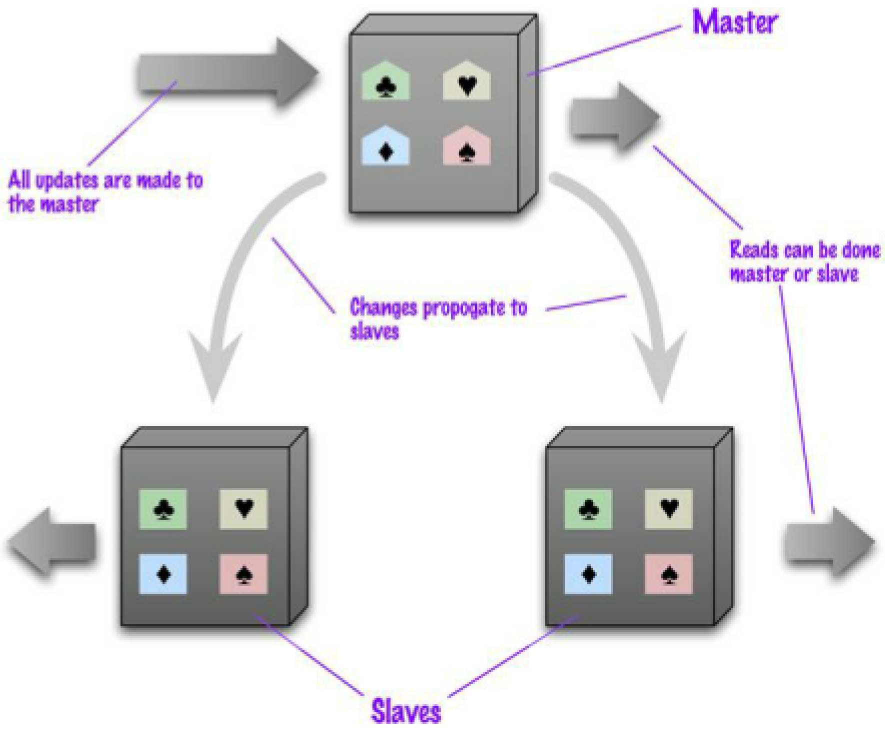
When data is stored on a server and there is a server crash, there is a chance that the data will be lost. To avoid this, the technique of data replication is used. The data is stored on two or more servers instead of a single server.



Courtesy: Google

Master-Slave Replication:

One of the servers is a master, and the other servers are called slaves. The slaves are connected to the master. All the writes happen to the master and then the master, sends these changes to the slaves. The reads can then be handled by the slaves. Through this process, the load is distributed.


Master
All updates are made to the master
Reads can be done from master or slave
Changes propogate to slaves
Slaves

Pros:
More read requests:

1. Add more slave nodes.

2. Ensure that all read requests are routed to the slaves

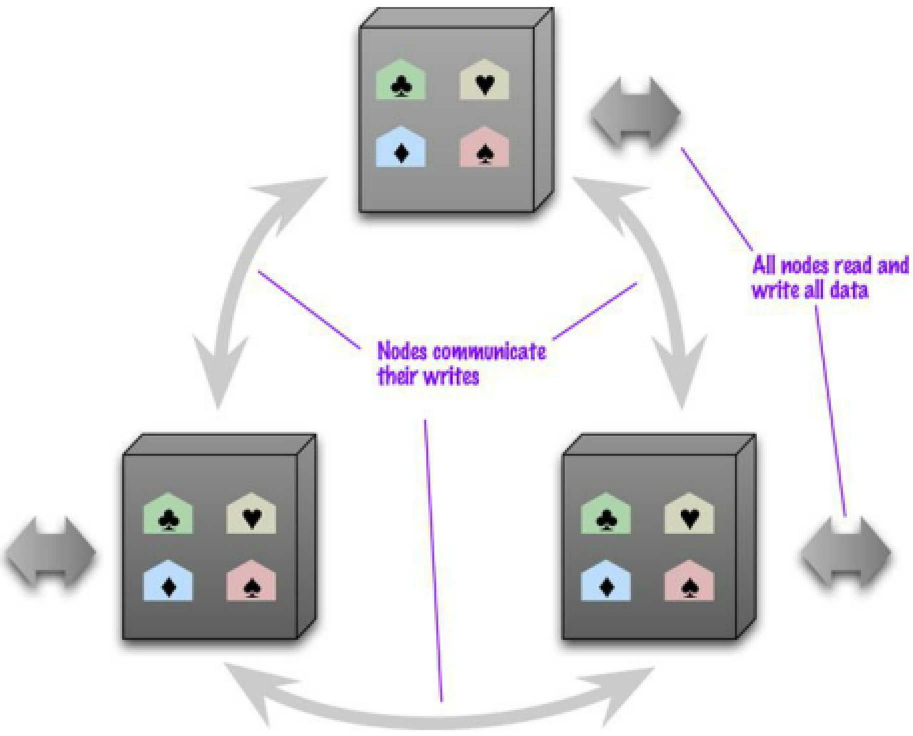
Should the master fail, the slaves can still handle read requests
Good for datasets with a read-intensive dataset
Cons:
1. The master is a bottleneck. Limited by its ability to process updates and to pass those updates on Its failure does eliminate the ability to handle writes until:
1. the master is restored or
2. a new master is appointed
Bad for datasets with heavy write traffic

So We can think of master-slave replication in the case of Read heavy systems. So it's important to clarify beforehand the kind of system (Read heavy or write-heavy)we are intended to design.

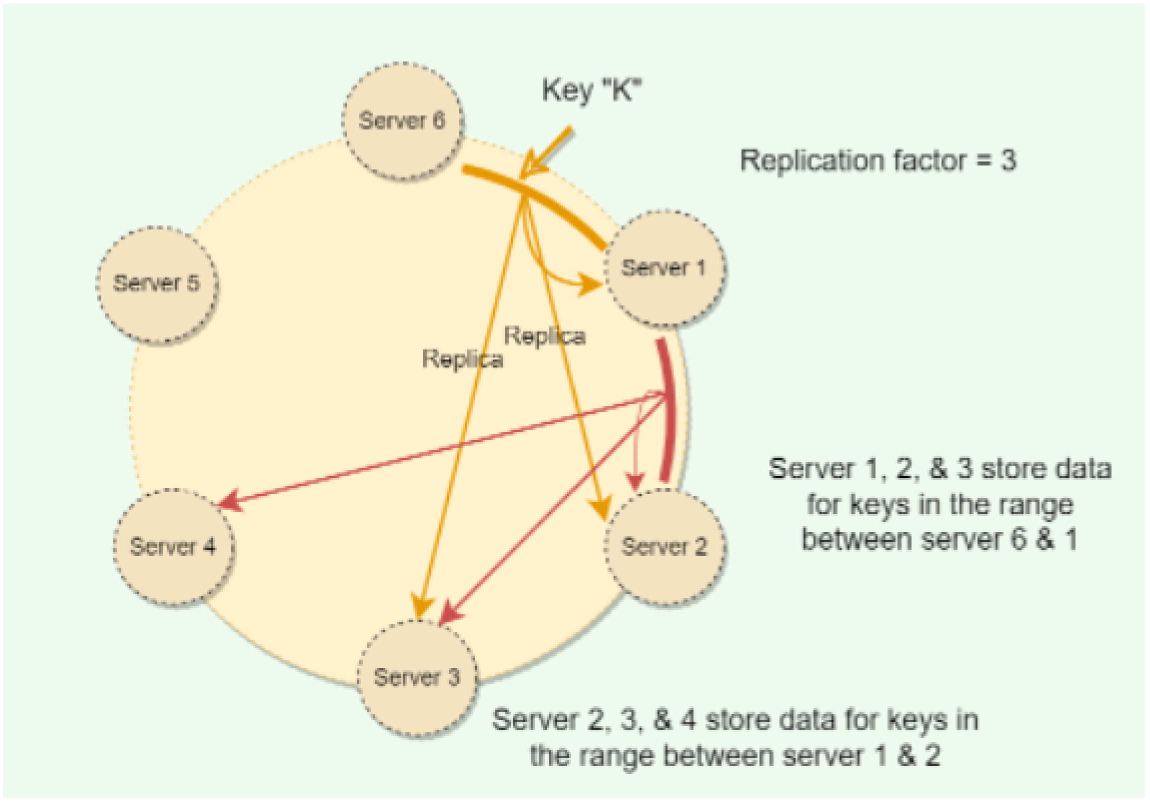
Peer-to-Peer Replication or Optimistic replication:

1. All the replicas have equal weight, they can all accept writes

2. The loss of any of them doesn't prevent access to the datastore


All nodes read and write all data
Nodes communicate their writes

In this replication scheme, each data item is replicated on multiple N nodes, where N is known as the replication factor. There is a technique is called as **consistent hashing** to direct user requests. The first node encountered while traversing the ring clockwise upon hashing a key is known as the **coordinator node**. The coordinator node serves the client requests and replicates the data on the next N-1 clockwise successor nodes.


Key "K"
Replication factor = 3
Server 6
Server 1
Server 5
Replica
Replica
Server 4
Server 2
Server 3
Server 1, 2, & 3 store data for keys in the range between server 6 & 1
Server 2, 3, & 4 store data for keys in the range between server 1 & 2

Educative.io

Pros:
you can ride over node failures without losing access to data
you can easily add nodes to improve your performance

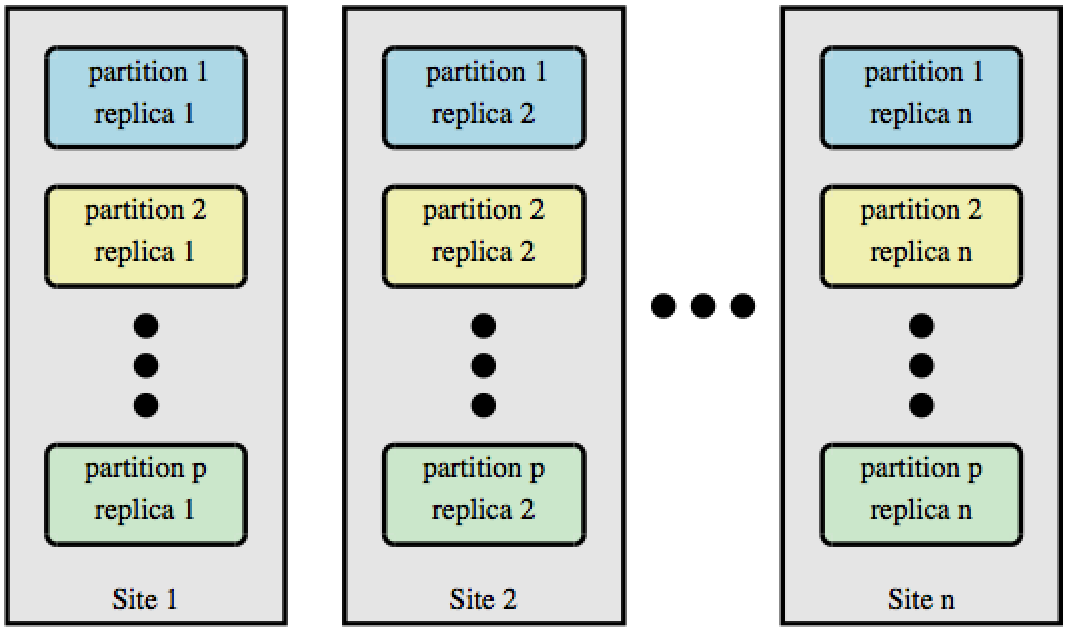
Cons:
Two people can update different copies of the same record stored on different nodes at the same time — ***a write-write conflict***.
Inconsistent writes are forever.

Partitioning:

The definition and pros and cons of it are for our own understanding If the interviewer asks we can provide some input on it otherwise it is not needed.

With replication, we make multiple copies of the data, and each server contains the complete data. But if the data is too large and can't be saved on a single instance? In that case, we may need to split the data and store it in different instances. This is called **partitioning** the data.

Partitioning the data will not solve the problem of data safety. If a node goes down, the data stored in that node will be lost. So, replication is required in addition to partitioning. Although it will increase the amount of data that needs to be stored, the data will not be lost in the case of a failure.



Google image

Advantages of partitioning:

Partitioning helps in ***horizontal scaling*** *of the data. If the data is spread across different servers, then the traffic can be routed to different servers, and and each server will have less load.*

Sometimes it may not be possible to store data on a single server because of its size. In this case, using partitioning is the only option left.

Disadvantages of partitioning:

If a transaction involves multiple keys and the keys are available on different servers then the transaction is not supported.

Certain operations that involve multiple keys, such as the union of two sets are not possible if the keys are present in different instances.

Which partitioning technique to use?

Range Partitioning:

Let’s suppose we have employee data stored in our DB, where the employee ID is the key. If we have three nodes available, then we can define that all the keys from range 0 to 10,000 will be stored on Node 1. Keys from the range of 10,000 to 20,000 will be stored on Node 2 and so on.

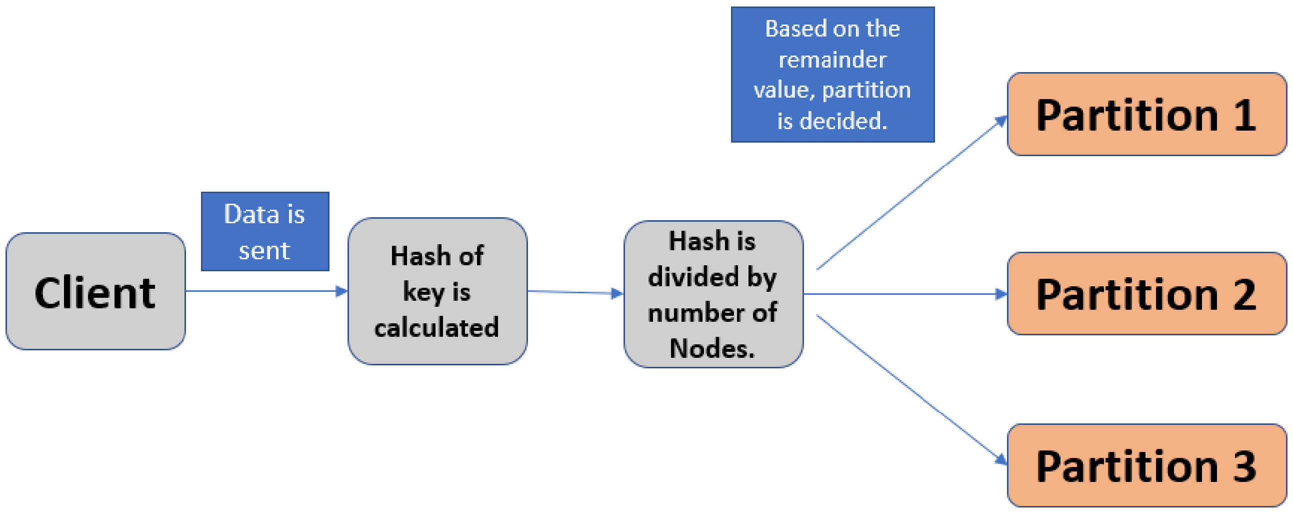
This partitioning can lead to an uneven distribution of keys. Let’s say about 70 percent of the keys start with A. Node 1 will then wave most of the load, and the other nodes will be relatively empty.

Hash partitioning:

In this partitioning, we need to have a 32-bit hash function that takes a string. Whenever you give it a new string, it returns a seemingly random number between 0 and pow(2, 32)−1. Even if the input strings are very similar, their hashes are evenly distributed across that range of numbers.

For partitioning purposes, the hash function need not be cryptographically strong: for example, MongoDB uses MD5, Cassandra uses Murmur3, and Voldemort uses the Fowler–Noll–Vo function, Redis uses CRC16.

In this method of partitioning, the keys are evenly distributed. The number of nodes must be a prime number to get the most effective partitioning.


Based on the remainder value, partition is decided.
Data is sent
Client
Hash of key is calculated
Hash is divided by number of Nodes.
Partition 1
Partition 2
Partition 3

Hash partitioning mechanism

Educative.io

At this point, we know which partitioning strategy to use? How this strategy can be used in the case of master-slave replication and peer-to-peer replication? Well, I meant to say how this strategy can be used in case of read-heavy systems and write-heavy systems?

Answer: Consistent Hashing in Peer to Peer replication (write-heavy), Hash slot in case of master-slave replication(read heavy).

Consistent Hashing and Hash Slot or Presharding:

Consistent Hashing is a distribution scheme that does not depend directly on the number of servers, so that, when adding or removing servers, the number of keys that need to be relocated is minimized.

This allows servers and objects to scale without affecting the overall system. It is a circle with the power of 0–2 ^ 32–1.

The main operation steps are as follows:

1. To hash a node, usually use its node's IP or data with a unique label to hash (IP), and distribute its value on this closed circle.
2. Hash (key) the stored key, and then distribute its value on this closed circle.

3. A node found clockwise from the location where hash(key) is mapped on the circle is the node that stores the key. If no node is found at 0 on the circle, the first node in the clockwise direction after 0 is the key storage node.

Hash Slots take care of the importance that the master-slave model in a sharded environment does not have, such as adding and removing a node.

The hash slots are **pre-calculated** for which it is solely determined by the administrator of the Cluster. For example, the distributed algorithm that a Cluster can use to map keys to hash slots is,

```
HASH_SLOT = CRC16(key) mod HASH_SLOTS_NUMBER
```

for example, you may have a cluster with 3 nodes, where: Node A contains hash slots from 0 to 5500. Node B contains hash slots from 5501 to 11000. Node C contains hash slots from 11001 to 16383. This allows to add and remove nodes (scale) in the cluster easily and does not require any downtime.

Fail-over Procedure:

How do detect node failures?

👉 Each node knows information about other nodes and they are constantly gossiping with each other using a **gossip protocol**.

So nodes will know if some node is failed in the cluster. This helps the cluster to take necessary actions so it could avoid single-point failure.

Every node has a unique ID in the cluster. This ID is used to identify each and every node across the entire cluster using the gossip protocol. So, a node keeps the following information within itself;

```
node ID, IP, and port a set of flags what is the Master of the node if it is flagged as “slave”
last time a node was pinged last time the pong was received.
```

What will happen when a node goes down?

In a master-slave architecture what will happen when the master goes down?

Cluster admin is configured to point to a master you need to be managed. It will then query the master for all connected slaves. Once it has determined the configuration of the pod it will proceed to monitor each node and upon confirmation of a down master, it will promote a slave and reconfigure each slave to point to the new master.

For example, Redis Sentinel works in this way.

Sentinel uses a voting+quorum mechanism so you need at least three Sentinel nodes working together. I refer to this as a Sentinel constellation.

So this gives another discussion: ***what is a Quorum*?**

In a master-slave architecture what will happen when the slave goes down?

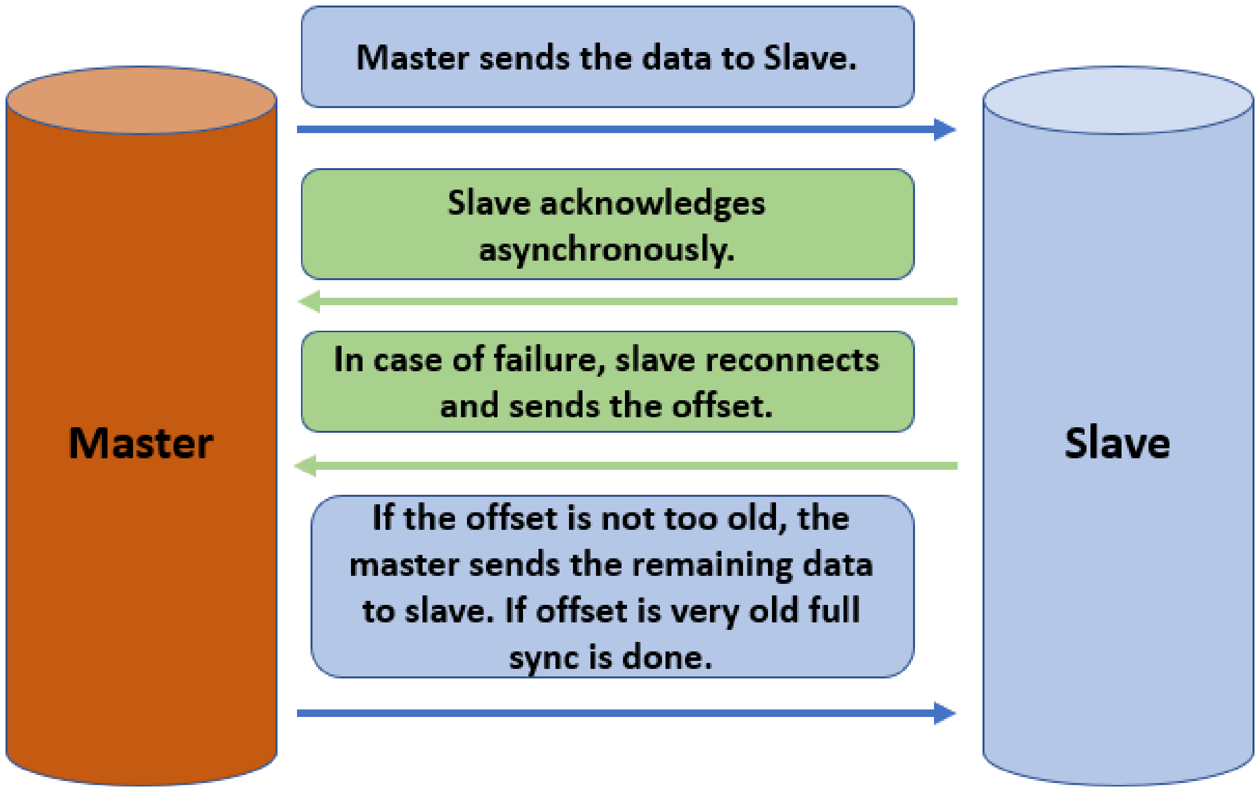
If a slave gets disconnected from the master, it automatically reconnects and tries to be the exact copy of the master **asynchronously**. There are two approaches that a slave takes to get in sync with the master.

- **Partial Synchronization**: The slave tries to get only the data that it missed while it was disconnected. The slave sends the master information about the point to which it has the data. We will discuss this in more detail later.
- **Full Synchronization**: If partial synchronization is not possible, then full synchronization takes place. The master sends all the data to a file for example rdb file used by **Redis**. This file is sent to the slave. The slave then saves it to its disk. After that is done, the slave loads the data from the rdb file to its memory. There is an option of *disk-less replication* as well.

if a slave is disconnected and reconnected, it syncs only the data that it missed,* how does the master know what data the slave missed*?

The **replication ID** and **offset** are used to figure this out. Each master server has a **replication ID** that is also inherited by a slave. The master also has an offset that increments for every byte of replication stream that is produced to be sent to replicas. When a slave is reconnected, it sends its offset to the server.

Let's suppose the master has a replication ID of **123gh3** and an offset of **12,000**. The slave has the replication ID of **123gh3** and the offset of **10,000**. When the slave connects with the master, it will send its offset of 10,000 to the master. The master will then know which bytes the slave is missing and it will resend those bytes to the slave.


Master sends the data to Slave.
Slave acknowledges asynchronously.
In case of failure, slave reconnects and sends the offset.
If the offset is not too old, the master sends the remaining data to slave. If offset is very old full sync is done.
Master
Slave

In Peer-to-peer replication architecture:

How to ensure our system is always writeable?

While writing data, if the required node is unreachable, another node can accept writes on its behalf. The write is then kept in a local buffer and sent out once the destination node is healthy again. This process is called **Hinted handoff**, and it makes our system "***always writeable***" even in the extreme case where only a single node is alive, write requests will still get accepted and eventually processed.

How to reconcile concurrent updates on an object:

Most of the NoSQL stores prefer **availability over consistency. They are always writable (AP systems in terms of CAP theorem) — in the presence of partition, they are capable to accept writing**. So most of the NoSQL system provides eventual consistency. Our system also ensures eventual consistency.

There are several issues that could happen in a distributed KV system where data is replicated asynchronously (Eventual consistent system):

- If a network partition happens (a node failure also has a similar effect) between two replicas storing the same data, they go out of sync. In a very high-scale environment, the replica nodes would end up with different values for the same key after some time.
- If different concurrent clients update the same key in different replicas at the same time, potentially their values would diverge.
- If a node loses some data or somehow data corruption happens, it goes out of sync with other replicas.

All these above scenarios **create conflicts and anomalies:**

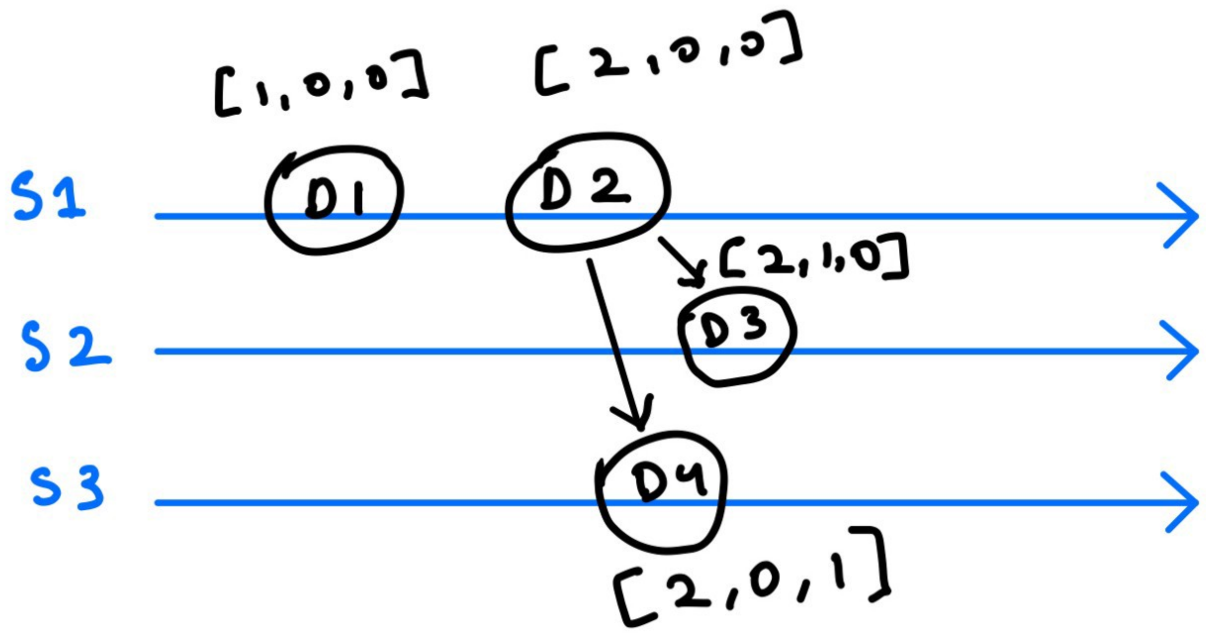
- A **single node** might end up with different versions of value objects for the same key due to concurrent client updates.
- The same key could exist across **different nodes** with different versions of value objects.

So we can use vector clocks in an attempt to provide logical ordering between the versions.

Inconsistency Resolution in Databases using Vector clocks:

Vector clocks act as a versioning system for every operation and are commonly used in BASE databases like Cassandra, DynamoDB, etc. The problem of version conflict in such databases is common because of concurrent updates they receive and these databases are highly distributed. Vector clocks can tell whether states are conflicted or not and conflict can be resolved by some other entity or client.

Imagine a database cluster with 3 write nodes or servers [S1, S2, S3] i.e any of these 3 servers can cater to a write. Below is the scenario:



- A client writes to S1 some data item as D1.
- Another client reads D1 and updates it to D2 and writes it back. S1 handles this write.
- Now two clients read D2 concurrently and update it. One of them updates data to D3 and writes it, which is handled by S2. On the other hand, the second update it to D4 and write it.

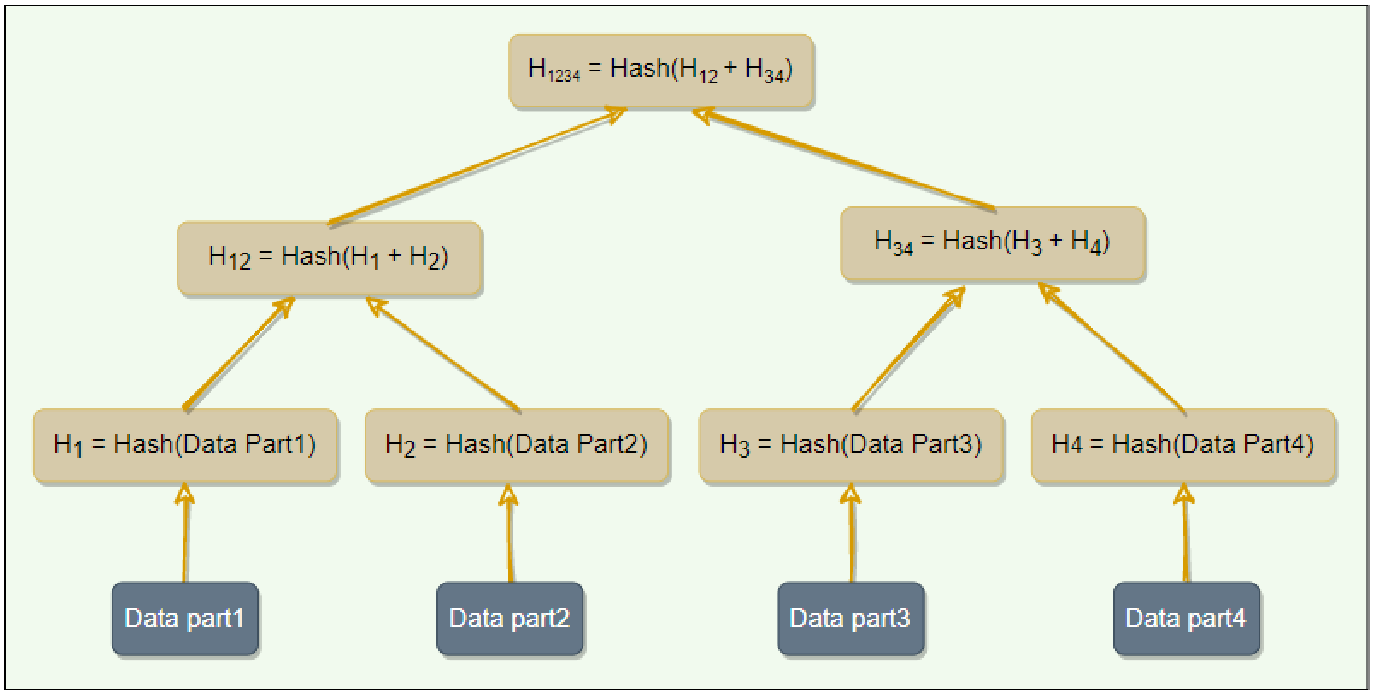
The vector clocks of D3[2,1,0] & D4[2,0,1] indicate that these two are concurrent events and show a conflict as these vector violate our two constraints discussed above.

Conflicts can be resolved by either client or some consensus algorithms.

Is there any good way to resolve conflicts in the background?

We have used vector clocks to remove conflicts while serving read requests. Now, if a replica falls significantly behind others, it might take a very long time to resolve conflicts using just vector clocks. It would be nice to be able to automatically resolve some conflicts in the background. To do this, we need to quickly compare two copies of a range of data residing on different replicas and figure out exactly which parts are different.

The solution is : ***Merkle Tree***



Merkle tree

Comparing Merkle trees is conceptually simple:

1. Compare the root hashes of both trees.
2. If they are equal, stop.
3. Recurse on the left and right children.

Ultimately, this means that replicas know precisely which parts of the range are different, and the amount of data exchanged is minimized.

Here is a summary of system design patterns used in our KV store:

- **Consistent Hashing or Hash Slot:** To distribute its data across nodes.
- **Replication**: To ensure high availability and durability.
- **Quorum:** To ensure data consistency, each KV store writing operation can be configured to be successful only if the data has been written to at least a quorum of replica nodes.
- **Gossip protocol:** We have used the gossip protocol that allows each node to keep track of state information about the other nodes in the cluster.

- **Hinted Handoff:** We have used Hinted Handoff to remember the write operation for failing nodes.
- **Vector clocks:** To reconcile concurrent updates on an object.
- **Merkle trees:** For anti-entropy and to resolve conflicts in the background.