Key Value Store/No-SQl Db

1.NoSQL

NO SQL is a type of database used to store data without using tables like SQL.

NoSQL = “Not Only SQL”

It stores data in different formats, not just rows & columns.

Example (easy)

Instead of a table like this (SQL):

Name | Age | City

Vijay | 22 | Delhi

NoSQL stores data like this:

```
{
  name: "Vijay",
  age: 22,
  city: "Delhi"
}
```

Why NoSQL is used

- **Handles huge data**
- **Very fast**
- **Good for real-time apps**
- **Easy to scale (add more users/data)**

Where NoSQL is used

- **WhatsApp**
- **Instagram**
- **Facebook**
- **Netflix**

Types of NoSQL (just names)

- **Document (MongoDB)**
- **Key-Value (Redis)**

- Column (Cassandra)
- Graph (Neo4j)
- Amazon Dynamo Db(key-value)

2.Create our own Key-Value Store

STEP 1.: Simple Key-Value Store (Client–Server DB)

What are we building?

A very basic Key-Value Database.

Example:

key = "user1"

value = "Vijay"

Architecture (simple)

Client ---> Server (DB)

- Client sends GET / PUT request
- Server stores data in memory (Map / Dictionary)

Example operations

- PUT(key, value)
- GET(key)

Advantages

- Very simple
- Very fast
- Easy to build
- Good for small apps

Disadvantages

- Single server → Single Point of Failure
- Limited storage
- Cannot handle many users
- If server crashes → data lost

Problem identified: Scalability & reliability

STEP 2: Horizontal Scaling of Database

What is Horizontal Scaling?

Instead of one big server, we use multiple servers.

Client

|

Load Balancer

|

DB1 DB2 DB3

Now data is distributed across servers.

STEP 2.1: First Approach — Modular (Modulo) Based Partitioning

Idea

Use key % number_of_servers

Example:

servers = 3

key = 7

7 % 3 = 1 → store in DB1

How it works

- Hash the key
- Use modulo
- Decide which DB stores the data

Advantages

- Easy to implement
- Fast lookup
- Even data distribution (initially)

Disadvantages

- If server count changes → everything breaks
- Data needs to be rehashed
- Massive data movement
- Not scalable in real systems

This approach is not production friendly

STEP 3: Problem with Modulo Approach

Example problem

- Initially: 3 servers
- Later: add 1 more → now 4 servers

Old:

key % 3

New:

key % 4

Result:

- Almost all keys move to different servers
- System downtime
- Huge cost

We need a better hashing strategy

STEP 4: Consistent Hashing (Correct Solution)

What is Consistent Hashing?

A technique where adding/removing servers affects only a small portion of data.

How it works

- Hash keys and servers on a ring (0–360°)
- Each key goes to the next server clockwise

---- S1 ---- S2 ---- S3 ----

↑

key

Adding a new server

- Only nearby keys move
- Rest of data stays untouched

Advantages of Consistent Hashing

- Minimal data movement
- Easy horizontal scaling
- No downtime

- Used by real systems:
  - Cassandra
  - DynamoDB
  - Redis Cluster

Disadvantages

- Slightly complex to implement
- Needs virtual nodes (vnodes) for balance

What is the Celebrity Problem?

In consistent hashing, sometimes one server becomes too famous

Meaning:

Too many keys map to one single server

That server gets:

- Too much traffic
- Too much data
- High load
  While other servers stay almost idle.

That overloaded server is called the “Celebrity Node”.

Why does the Celebrity Problem happen?

Reason 1: Uneven hash distribution

- Servers are placed randomly on the hash ring
- Some servers get large gaps
- Keys falling in that big gap all go to one server

Example

Ring:

S1 ---------- S2 -- S3

- Huge gap before S2
- Most keys land there
- S2 becomes celebrity

Problems caused by Celebrity Node

- Server crashes due to load
- High latency
- Poor performance

- Unequal resource usage

How Virtual Nodes (VNodes) solve this

What is a Virtual Node?

One physical server appears multiple times on the hash ring

Example:

Physical Server S1

Virtual nodes: S1_1, S1_2, S1_3

Hash Ring with Virtual Nodes

Instead of:

S1 ---- S2 ---- S3

We get:

S1_1 -- S2_1 -- S3_1 -- S1_2 -- S2_2 -- S3_2

Now keys are evenly distributed 🎯

How VNodes fix the Celebrity Problem

Before VNodes

- One server gets a large range
- Too many keys → overload

After VNodes

- Load split into small chunks
- No big gaps
- Traffic is balanced

No single server becomes celebrity

Extra benefit of Virtual Nodes

When a server fails

- Only its virtual nodes are reassigned
- Small data movement
- System stays stable

STEP 5: Client – Coordinator – S0, S1, S2

How Consistency Is Achieved

1. High-Level Architecture

Client

|

v

Coordinator (any one server)

|

v

S0 S1 S2 (replica nodes)

- **The client never talks directly to replica nodes.**
- **The server that receives the request becomes the Coordinator.**

2. What is the Coordinator?

The Coordinator is responsible for:

- **Finding replica nodes using consistent hashing**
- **Sending read/write requests to replicas**
- **Waiting for responses based on consistency rules**
- **Returning the final response to the client**

Any node in the cluster can act as a coordinator.

3. What are S0, S1, S2?

- **They are replica servers**
- **Same data is stored on multiple servers**
- **This is done for:**
  - **Fault tolerance**
  - **High availability**

Example:

Replication Factor (N) = 3

Data is stored on S0, S1, S2

4. WRITE Operation (PUT)

Step-by-step flow

1. **Client sends a write request:**

PUT(key, value)

1. **Coordinator finds replicas using the hash ring:**

Primary → S0

Replicas → S1, S2

1. **Coordinator sends write request to all replicas.**

5. Write Consistency Level (W)

The client specifies how many replicas must confirm the write.

W = 1

- **Only 1 replica must respond**
- **Very fast**
- **Weak consistency**

W = 2

- **2 replicas must respond**
- **Balanced choice (commonly used)**

W = 3

- **All replicas must respond**
- **Strong consistency, slower**

6. READ Operation (GET)

Step-by-step flow

1. **Client sends a read request:**

GET(key)

1. **Coordinator sends read requests to replicas:**

S0, S1, S2

1. **Coordinator waits for responses based on read consistency.**

7. Read Consistency Level (R)

The client specifies how many replicas must respond.

R = 1

- Fast
- May return stale data

R = 2

- Coordinator compares responses
- Returns the latest value

R = 3

- All replicas must respond
- Strongest consistency

8. Achieving Consistency (Quorum Rule)

The key rule:

R + W > N

Where:

- R = number of replicas read
- W = number of replicas written
- N = total replicas

Example:

N = 3

W = 2

R = 2

2 + 2 > 3

This guarantees strong consistency.

9. What If a Replica Is Down?

Example:

S1 is down

- Coordinator writes to S0 and S2
- Keeps a temporary record for S1 (hinted handoff)
- Syncs data when S1 comes back

The client is not affected.

10. Version Conflicts

If replicas return different values:

- Coordinator uses timestamps or vector clocks
- Chooses the latest version
- Or returns multiple versions (eventual consistency)

Points to remember

- Client talks only to the coordinator
- Coordinator manages all replicas
- Replication factor ensures reliability
- Consistency is controlled using R, W, and N
- Quorum rule guarantees correctness

Consistency Models

What is Consistency?

Consistency means:

What value a client sees when it reads data, especially after a write.

In distributed systems, multiple copies (replicas) of data exist, so consistency defines when and how all replicas become the same.

1.Strong Consistency

Meaning

After a write completes, all future reads return the latest value.

There is no stale data.

Example

Write: x = 10

Read: x → always returns 10

How it works

- **Write goes to all replicas (or a quorum)**
- **Read waits until replicas agree**

Advantages

- **Simple to reason about**
- **Always correct data**

Disadvantages

- **Slower**
- **Low availability if a node is down**
- **Hard to scale**

Where used

- **Banking systems**
- **Payment systems**
- **Critical financial data**

2.Weak Consistency

Meaning

After a write, reads may or may not return the latest value.

No guarantee of freshness.

Example

Write: x = 10

Read: x → may return 5 or 10

Advantages

- **Very fast**
- **High availability**

Disadvantages

- **Unpredictable reads**
- **Not suitable for important data**

Where used

- Caches
- Analytics
- Monitoring data

3.Eventual Consistency (Most Important)

Meaning

If no new writes happen, all replicas will eventually become consistent.

Temporary inconsistency is allowed.

Key idea

System prefers availability and speed over immediate correctness.

Example (Real-life style)

1. **You update your profile name on Instagram**
2. **You refresh immediately on another device**
3. **Old name still shows**
4. **After a few seconds → new name appears everywhere**

That's eventual consistency.

How it works internally

- **Write is accepted by one or few replicas**
- **Background processes sync data to other replicas**
- **Reads may return stale data temporarily**

Advantages

- **Very fast**
- **Highly available**
- **Scales to millions of users**

Disadvantages

- **Temporary stale data**
- **Conflict resolution needed**

Where used (VERY IMPORTANT)

- **Cassandra**
- **DynamoDB**
- **Redis Cluster**
- **Social media apps**
- **E-commerce carts**

- DNS

What is a Collision?

A collision happens when two different keys or two different updates end up at the same place or create ambiguity.

In distributed key-value stores, collision usually means:

- Two different writes happen at the same time
- System cannot clearly decide which value is latest

Collision Example (Realistic)

Two clients update the same key at the same time:

Key = "userName"

Client A → write "Vijay"

Client B → write "Ajay"

Because of:

- Network delay
- Multiple replicas
- Eventual consistency

Replicas may store different values:

S0 → "Vijay"

S1 → "Ajay"

S2 → "Vijay"

This is a write collision / conflict.

2.How Are Collisions Solved?

There are multiple strategies:

1. Last Write Wins (LWW)

- Uses timestamp
- Latest timestamp overwrites others

Simple
Data loss possible

2. Client-side Resolution

- System returns multiple versions
- Client decides which one to keep

No data loss
More complexity

3. Versioning (Most important)

- Track history of writes
- Detect conflicts instead of blindly overwriting

This brings us to Versioning Pairs.

What Are Versioning Pairs?

A versioning pair is:

(value, version)

Example:

("Vijay", v1)

("Ajay", v2)

The version tells:

- When
- In what order
- From which update

1.Why Do We Need Versioning Pairs?

Problem Without Versioning

System sees:

"Vijay"

"Ajay"

Question:
Which one is correct?

System has no idea.

With Versioning Pairs

("Vijay", version=3)

("Ajay", version=5)

Now system knows:
"Ajay" is newer

2.How Versioning Solves the Problem

Detects Conflicts

- **If two versions are independent**
- **Neither is newer**

Example:

("Vijay", v3)

("Ajay", v3)

Conflict detected

Enables Safe Resolution

- **Merge**
- **Ask client**
- **Keep both temporarily**

3.Advanced Versioning: Vector Clocks (Concept)

Instead of one number:

version = {S0:2, S1:1}

This helps to:

- **Track who wrote what**
- **Detect concurrent writes**

Used in:

- **DynamoDB**
- **Cassandra**

Problem	Without Versioning	With Versioning
Concurrent writes	Data lost	Conflict detected
Eventual consistency	Confusing	Controlled
Replica mismatch	Overwrite	Reconciled
Client safety	Unsafe	Safe

Collision:

A collision occurs when multiple writes to the same key happen concurrently and replicas cannot agree on a single value.

Versioning Pairs:

A versioning pair stores a value along with its version to detect and resolve write conflicts in distributed systems.

- **Collisions are inevitable in distributed systems**
- **Versioning does not prevent collisions**
- **Versioning detects and safely resolves them**

Handling Failures in Distributed Systems

In distributed databases, failures are normal.
Systems are designed assuming nodes will fail.

Failures are mainly of two types:

1. **Temporary failure**
2. **Permanent failure**

1.Temporary Failure

What is a Temporary Failure?

A server goes down for a short time and comes back.

Reasons:

- **Network issue**
- **Restart**
- **GC pause**

- Power glitch

Example:

S1 is down for 30 seconds

Problems Caused

- Writes cannot reach S1
- Reads may miss data
- Inconsistency risk

How Systems Handle Temporary Failure

1. Replication

- Data already exists on other nodes
- System keeps working

2. Hinted Handoff (Very Important)

Idea

“I’ll temporarily keep your data for you.”

Example

Replicas: S0, S1, S2

S1 is down

Coordinator:

- Writes to S0 and S2
- Stores a hint for S1

Later:

S1 comes back

→ coordinator sends missed writes

Client never notices the failure.

3. Read Repair

- During reads, replicas compare data
- Outdated replica is updated automatically

4. Sloppy Quorum

- Write to any available nodes
- Maintain availability even if original replica is down

Temporary Failure

- Node is expected to return
- Data is buffered
- Synced later
- No data loss

2.Permanent Failure

What is a Permanent Failure?

A server is gone forever.

Reasons:

- Disk crash
- Hardware failure
- Node removed intentionally

Example:

S2 is dead permanently

Problems Caused

- Data replicas lost
- Reduced replication factor
- Risk of data loss

How Systems Handle Permanent Failure

1. Failure Detection

- Health checks
- Heartbeats
- Timeouts

System marks node as dead.

2. Replica Replacement

- System chooses a new node
- New node becomes replica

Old: S0, S1, S2

New: S0, S1, S3

3. Data Rebalancing

- Data is copied to the new node
- Uses consistent hashing
- Minimal data movement

4. Anti-Entropy (Background Sync)

- Periodic comparison between replicas
- Ensures correctness

Permanent Failure

- Node does not return
- Data is rebuilt
- Replication restored
- System remains safe

Aspect	Temporary Failure	Permanent Failure
Node returns	Yes	No
Data buffered	Yes	No
Fix method	Hinted handoff	Re-replication
Risk	Low	Medium
Client impact	None	Minimal

How Systems Detect Failure & Inconsistency

There are two different problems here:

1. Is a node alive or dead? → Gossip Protocol
2. Is data same or different across replicas? → Merkle Tree

Let's do them one by one.

1.Gossip Protocol (Failure Detection)

What is Gossip Protocol?

A gossip protocol is a way for servers to continuously share their health status with each other.

Just like people gossip:

- **One server tells another server what it knows**
- **That information spreads to everyone**

Why Do We Need Gossip?

Problem without gossip:

- **Central monitor = single point of failure**
- **Slow detection**
- **Not scalable**

Solution:

- **Decentralized**
- **Scales well**
- **Fast failure detection**

How Gossip Works (Step by Step)

Assume we have 4 servers:

S0, S1, S2, S3

Step 1: Periodic gossip

Every few seconds:

- **Each server randomly picks one or two servers**
- **Shares:**
  - **Who is alive**
  - **Who is suspected dead**

Example:

S0 → S2 : "I think S1 is alive, S3 is slow"

Step 2: Heartbeats

Each server maintains a heartbeat counter:

S1 heartbeat = 101, 102, 103...

If heartbeat stops increasing:
Node is suspected failed

Step 3: Suspicion phase (Important)

- Node is not marked dead immediately
- It is first marked as suspected
- Gossip spreads this suspicion

Why?

- Network delays can cause false failures

Step 4: Failure confirmation

If many nodes agree:

“S1 has not responded for T seconds”

Node is marked dead

Advantages of Gossip Protocol

- No central coordinator
- Fault tolerant
- Fast convergence
- Used by:
  - Cassandra
  - Dynamo
  - Kubernetes (conceptually)

Gossip protocol detects failures by periodically exchanging heartbeat and health information among nodes in a decentralized manner.

2.Merkle Tree (Detecting Data Inconsistency)

Important:

Merkle Tree is NOT for detecting node failure
It is for detecting data mismatch between replicas

What is a Merkle Tree?

A Merkle Tree is a hash-based tree used to efficiently compare large datasets.

Instead of comparing all data:
👉 Compare hashes

Problem Merkle Tree Solves

Suppose:

- **S0 and S1 store millions of keys**
- **Need to check if data is same**

Naive way:

- **Send all data over network**
- **Very slow & expensive**

Merkle Tree:

- **Compare small hashes**
- **Find exact differences**

How Merkle Tree Works (Step by Step)

Step 1: Split data into ranges

Range1 | Range2 | Range3 | Range4

Step 2: Hash each range (leaf nodes)

H1 = hash(Range1)

H2 = hash(Range2)

H3 = hash(Range3)

H4 = hash(Range4)

Step 3: Build tree upward

H12 = hash(H1 + H2)

H34 = hash(H3 + H4)

Root = hash(H12 + H34)

Step 4: Compare roots

S0 Root == S1 Root → Data same

S0 Root != S1 Root → Data different

Step 5: Narrow down differences

- **Compare child hashes**
- **Go deeper only where hashes differ**
- **Find exact keys to sync**

Very efficient

How Merkle Tree Helps in Failure Handling

After:

- Temporary failure
- Network partition

Merkle tree helps:

- Detect missing / outdated data
- Sync only required keys
- Restore consistency

This process is called Anti-Entropy

Advantages of Merkle Tree

- Minimal network usage
- Scales to large datasets
- Fast comparison
- Used in:
  - Cassandra
  - Dynamo
  - Git
  - Blockchain

A Merkle Tree detects data inconsistencies between replicas by comparing hierarchical hashes instead of entire datasets.

Aspect	Gossip Protocol	Merkle Tree
Purpose	Detect node failure	Detect data mismatch
What is shared	Heartbeats, status	Hashes
Works on	Nodes	Data
Cost	Low	Very low
Used when	Always running	Background sync

- Gossip → “Who is alive?”
- Merkle Tree → “Is data same?”

- **Together they make the system:**
  - **Reliable**
  - **Scalable**
  - **Self-healing**