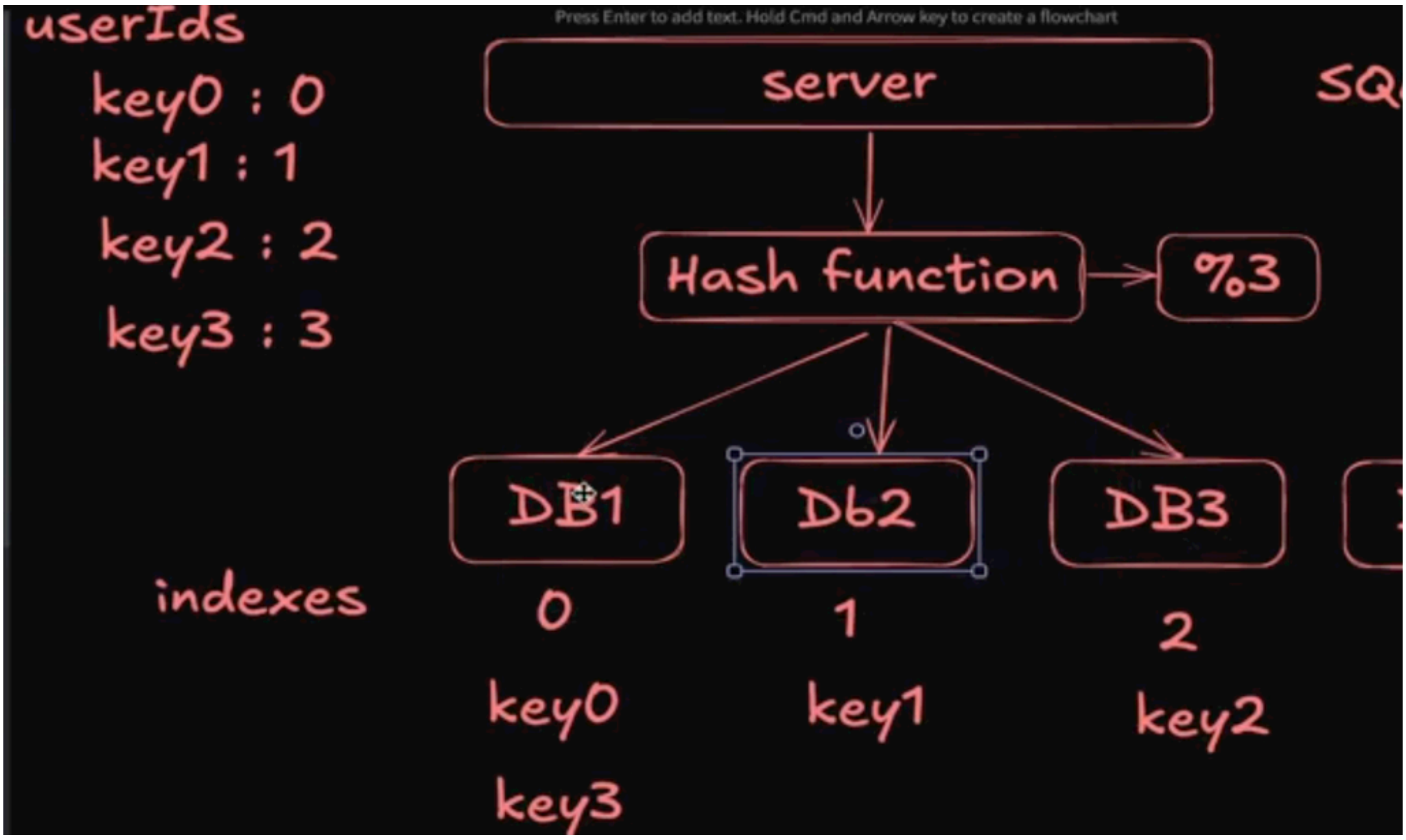
Consistent Hashing and LB

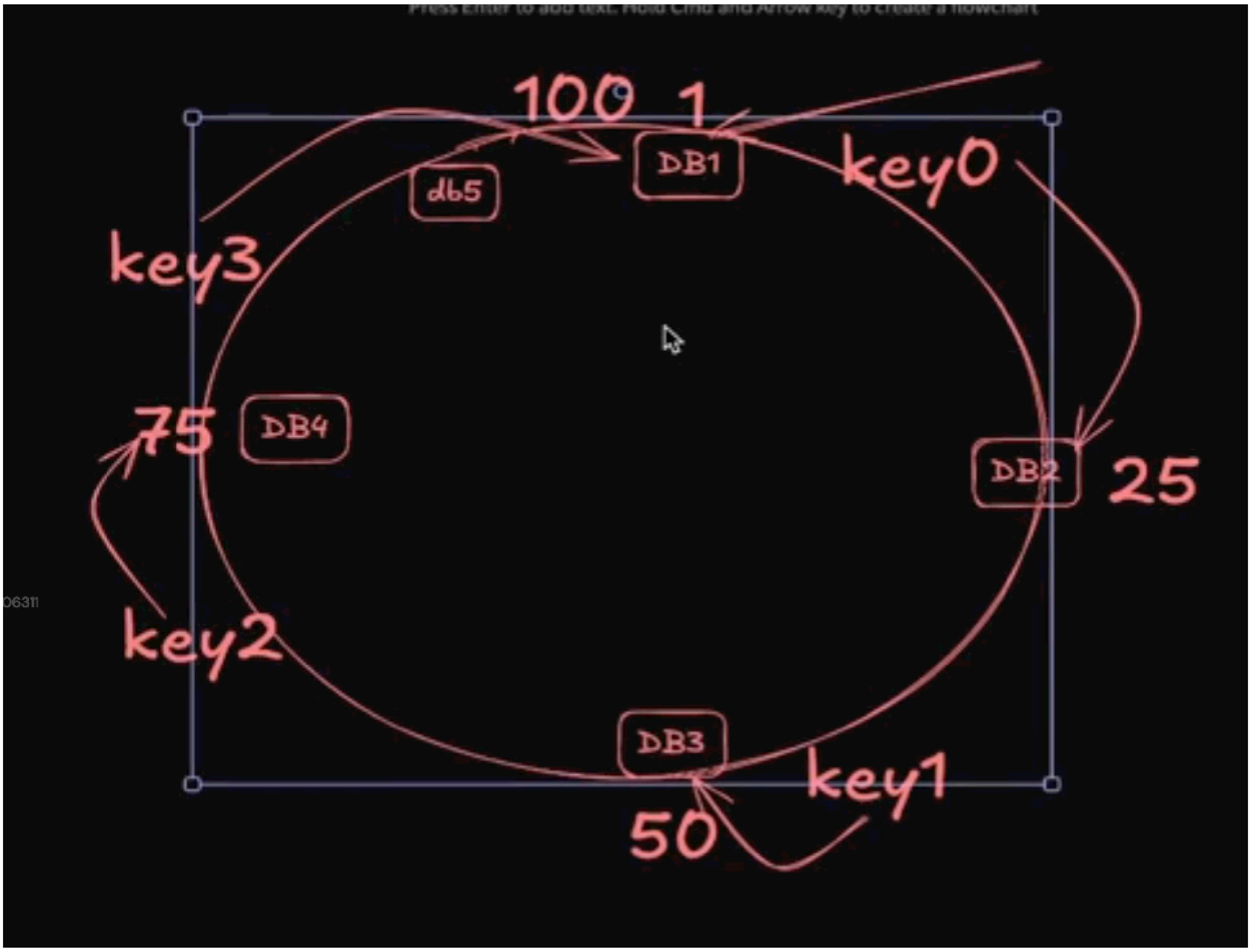


Consistent Hashing?

Consistent Hashing is a technique used to distribute data across multiple servers (nodes) in a way that:

- Works efficiently when servers are added or removed
- Minimizes the amount of data that needs to be moved

It solves the problem of data redistribution in distributed systems.



Press Enter to add text. Hold Cmd and Arrow key to create a flowchart
100
1
DB1
key0
db5
key3
75
DB4
DB2
25
key2
DB3
key1
50

1.Why Do We Need It?

Imagine we have 3 servers:

Server A, Server B, Server C

And we store user data using a simple hash function:

hash(key) % number_of_servers

Example:

hash("user1") % 3 → Server B

hash("user2") % 3 → Server A

Now suppose Server C crashes or we add Server D.

Then:

hash(key) % 4

Almost all keys will move to different servers.

This causes:

- Massive data movement
- High downtime
- System becomes unstable

Consistent hashing prevents this.

Core Idea

Instead of mapping keys directly to servers,
we map both keys and servers to a circular hash ring.

Think of it like a clock (0–360°) or a ring.

0 → 100 → 200 → 300 → 0 (wrap around)

2.How It Works

1.Create a Hash Ring

We hash all servers and place them on the ring.

Example:

hash(Server A) → 120

hash(Server B) → 250

hash(Server C) → 340

Now servers are located like this:

0 --- A(120) --- B(250) --- C(340) --- 0

2.Hash the Key

Example:

hash("user1") → 200

Now we move clockwise and assign the key to the next server.

user1 (200) → next server is B (250)

So Server B stores user1.

3.Add or Remove Servers

If we add Server D:

hash(Server D) → 300

Ring becomes:

A(120) → B(250) → D(300) → C(340)

Now only keys between 250 and 300 move from B to D.

Only small portion of data moves.

If Server B is removed:

Keys that belonged to B now go to D.

Again, only small portion of data moves.

Advantages

- Scales easily
- Minimal data movement
- Fault tolerant
- Great for distributed caches and databases

Used in:

- CDNs
- Distributed caches (Redis Cluster, Memcached)
- Load balancers
- Distributed DB sharding

Limitations of Consistent Hashing

1.Uneven Data Distribution

If servers are not placed properly on the ring:

- One DB/server may get too much data
- Another server may get very little

This causes load imbalance

Solution: Use Virtual Nodes

2.Complexity

- More complex than simple hash % n
- Requires maintaining the hash ring
- Needs logic for clockwise lookup

Harder to implement and maintain

3.Requires Extra Metadata

To know:

- Which key belongs to which node
- Where each node is located on the ring

System needs to store mapping info

4.Rebalancing Still Happens

Although minimal, when:

- A node is added or removed
- That node’s data must be moved

Not zero movement, just reduced movement

5.Hotspot Problem

If one key becomes very popular:

- That key always goes to same server
- That server becomes overloaded

Leads to performance issues

Example: One celebrity user data gets too many requests.

6.Difficult to Debug

Because:

- Data is spread across nodes
- Hard to track where a specific key is stored

Makes troubleshooting harder

7.Network Overhead

If node changes:

- Keys must be moved over network
- Can create temporary latency

8.Does Not Consider Server Capacity

Consistent hashing treats all nodes equally.

But in real world:

- Some servers are powerful
- Some are weak

Data should not be equally divided

Solution: Weighted hashing

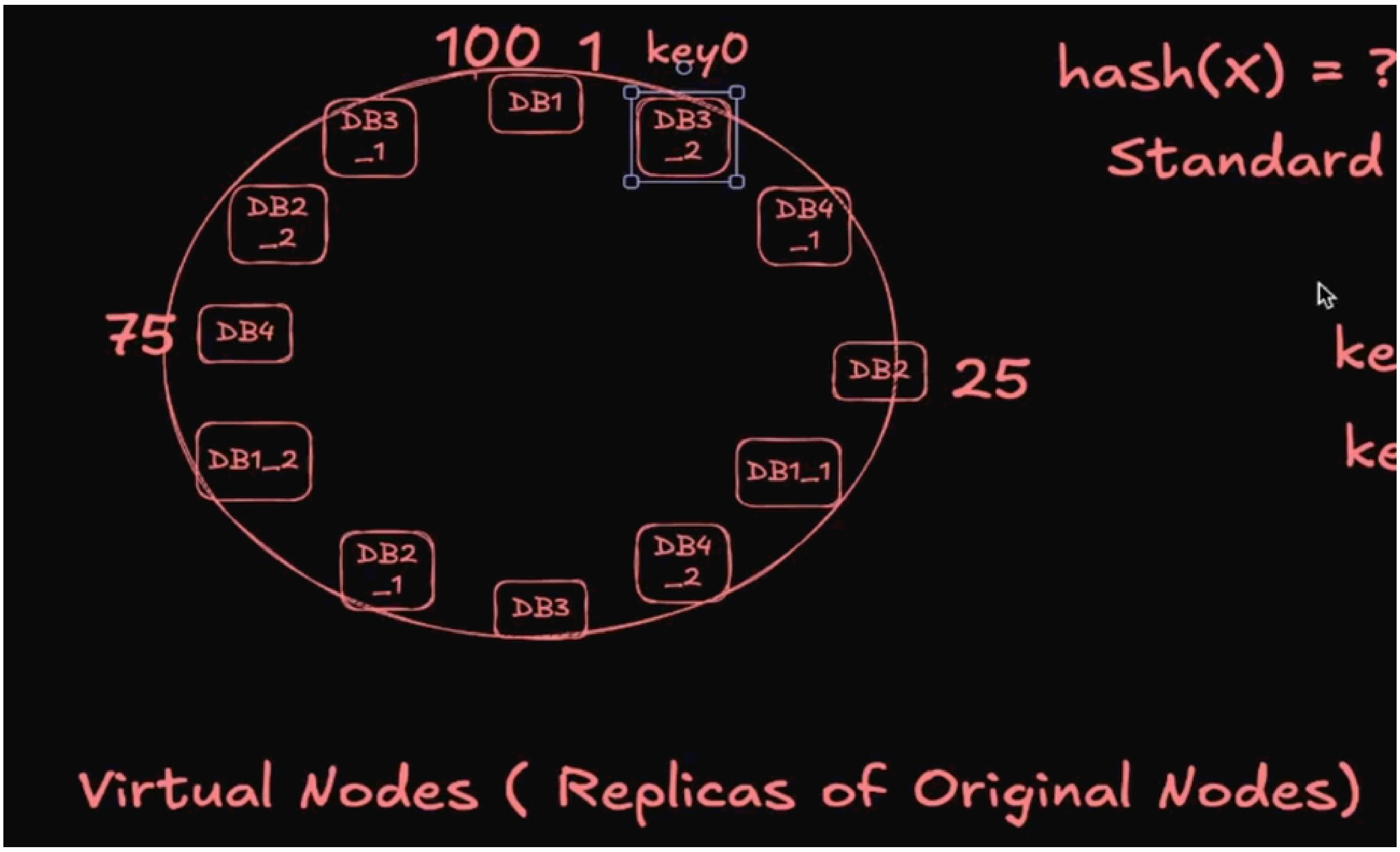
What are Virtual Nodes?

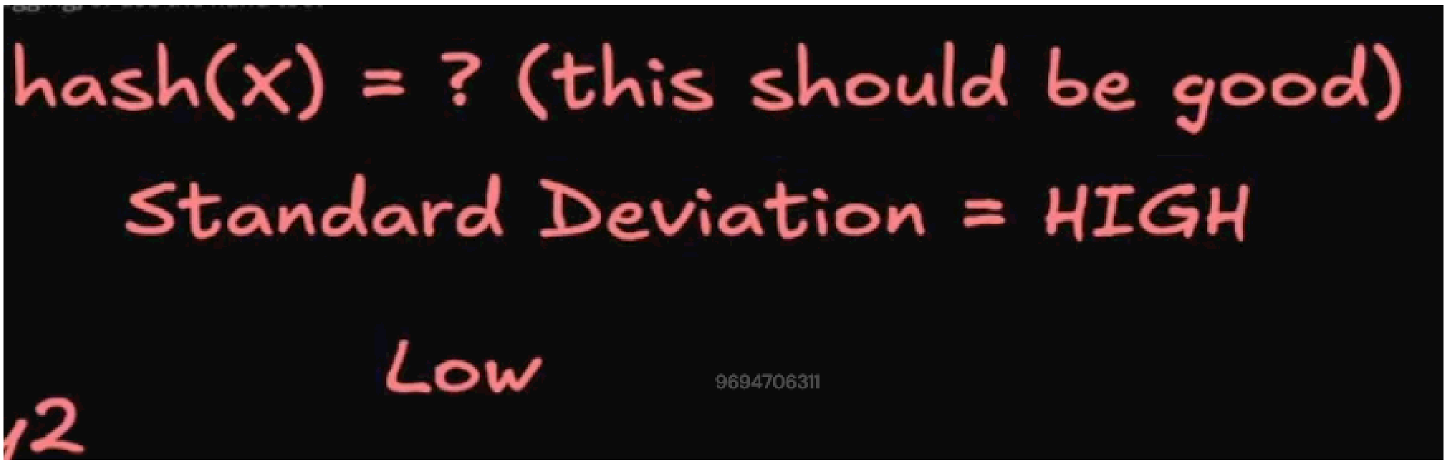
Virtual nodes (vnodes) are fake nodes created for each real server in consistent hashing.

Instead of putting one server on the hash ring, we put many virtual copies of it.

So:

Real Server → Multiple Virtual Nodes on the ring





1.Why we use Virtual Nodes?

We use them to balance the load properly.

Without virtual nodes:

- Some servers get too much data
- Some servers get very little data

With virtual nodes:

- Data is spread evenly
- Load becomes fair
- When a server fails or joins, less data moves

Virtual nodes = multiple positions of one server on the hash ring.

We use them for:

- Better load balancing
- More stable distribution
- Less data movement

Load Balancer Algorithms

We divide them into:

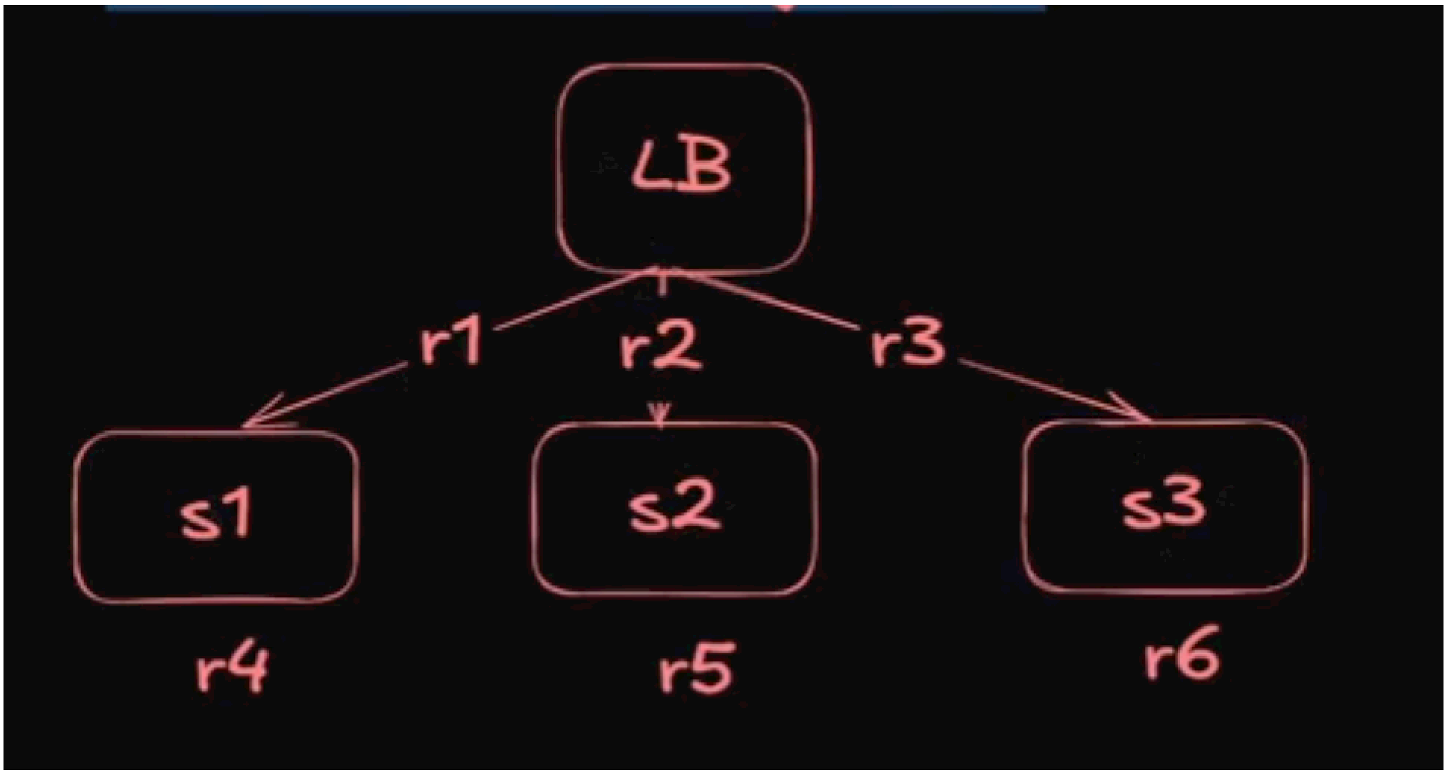
1. Static Algorithms
2. Dynamic Algorithms

STATIC LOAD BALANCING ALGORITHMS

Static means:

No real-time server load checking
Traffic is distributed using fixed rules.

1.Round Robin



How it works

Load balancer sends each new request to the next server in order.

Example:

Servers: S1, S2, S3

Req1 → S1

Req2 → S2

Req3 → S3

Req4 → S1

Req5 → S2

It keeps rotating.

Why we use it

- Very simple
- All servers get equal traffic
- No need to track server load

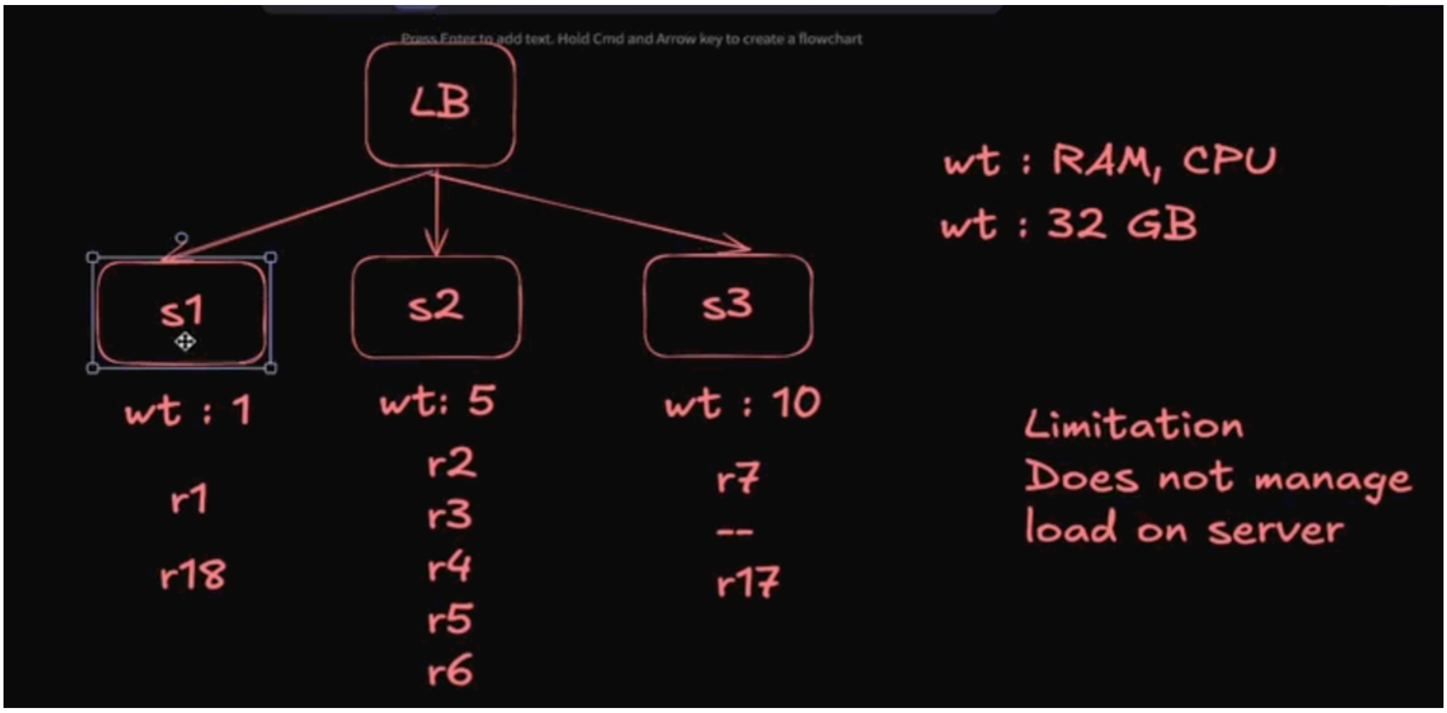
When it fails

- If S1 is slow and S2 is fast → both still get same traffic
- Can overload weak servers

Best Use Case

- All servers have same capacity
- Good for small or simple systems

2.Weighted Round Robin



How it works

Servers get traffic according to their capacity (weight).

Example:

Servers:

S1 weight = 3

S2 weight = 1

Traffic:

Req1 → S1

Req2 → S1

Req3 → S1

Req4 → S2

Req5 → S1

So S1 gets more traffic because it is stronger.

Why we use it

- Supports different server capacities
- Prevents small servers from overload

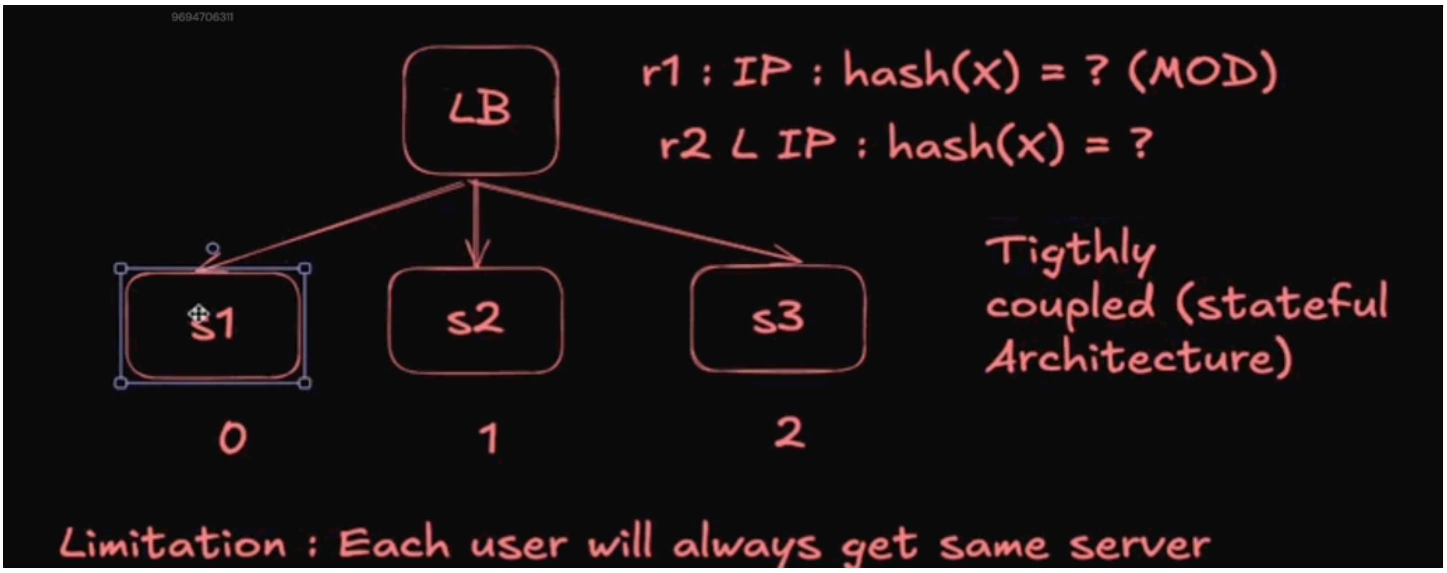
When it fails

- Still doesn't check live load
- If S1 becomes slow, traffic still goes

Best Use Case

- Cloud servers with different power
- Works well for fixed server capacity environments

3.IP Hash


LB
r1 : IP : hash(x) = ? (MOD)
r2 L IP : hash(x) = ?
s1
s2
s3
0
1
2
Tigthly coupled (stateful Architecture)
Limitation : Each user will always get same server

How it works

Load balancer uses client IP address to decide server.

server = hash(IP) % N

Example:

IP 192.168.1.10 → S2

IP 192.168.1.20 → S1

Same IP always goes to same server.

Why we use it

- Provides sticky sessions
- Same user always reaches same server

- Good when session data is stored in server memory

When it fails

- If a server fails → users mapped to it lose connection
- Can cause uneven load distribution

Best Use Case

- Login/session-based systems
- When caching is server-specific

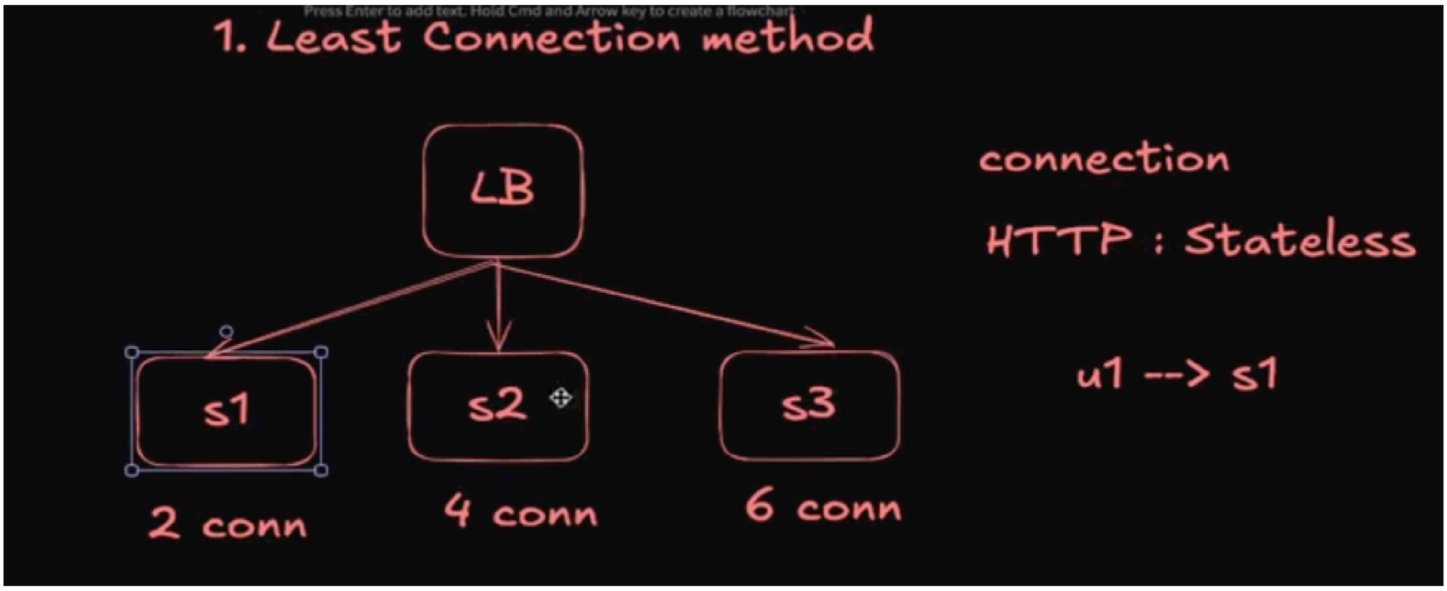
Algorithm	Balancing Method	Best For	Limitation
Round Robin	Order	Equal servers	Ignores server health
Weighted RR	Capacity	Unequal servers	Still static
IP Hash	Client IP	Sticky sessions	Uneven load possible

DYNAMIC LOAD BALANCER ALGORITHMS

Dynamic means:

Load balancer checks server's current load and health
Then decides where to send the request.

1.Least Connections Time


Press Enter to add text. Hold Cmd and Arrow key to create a flowchart
1. Least Connection method
LB
s1
s2
s3
2 conn
4 conn
6 conn
connection
HTTP : Stateless
u1 --> s1

How it works

Request goes to the server that has the fewest active connections.

Example:

S1 → 10 users connected

S2 → 3 users connected

S3 → 7 users connected

Next request → S2

Why we use it

- Prevents overload
- Works well when requests are long (like video streaming)

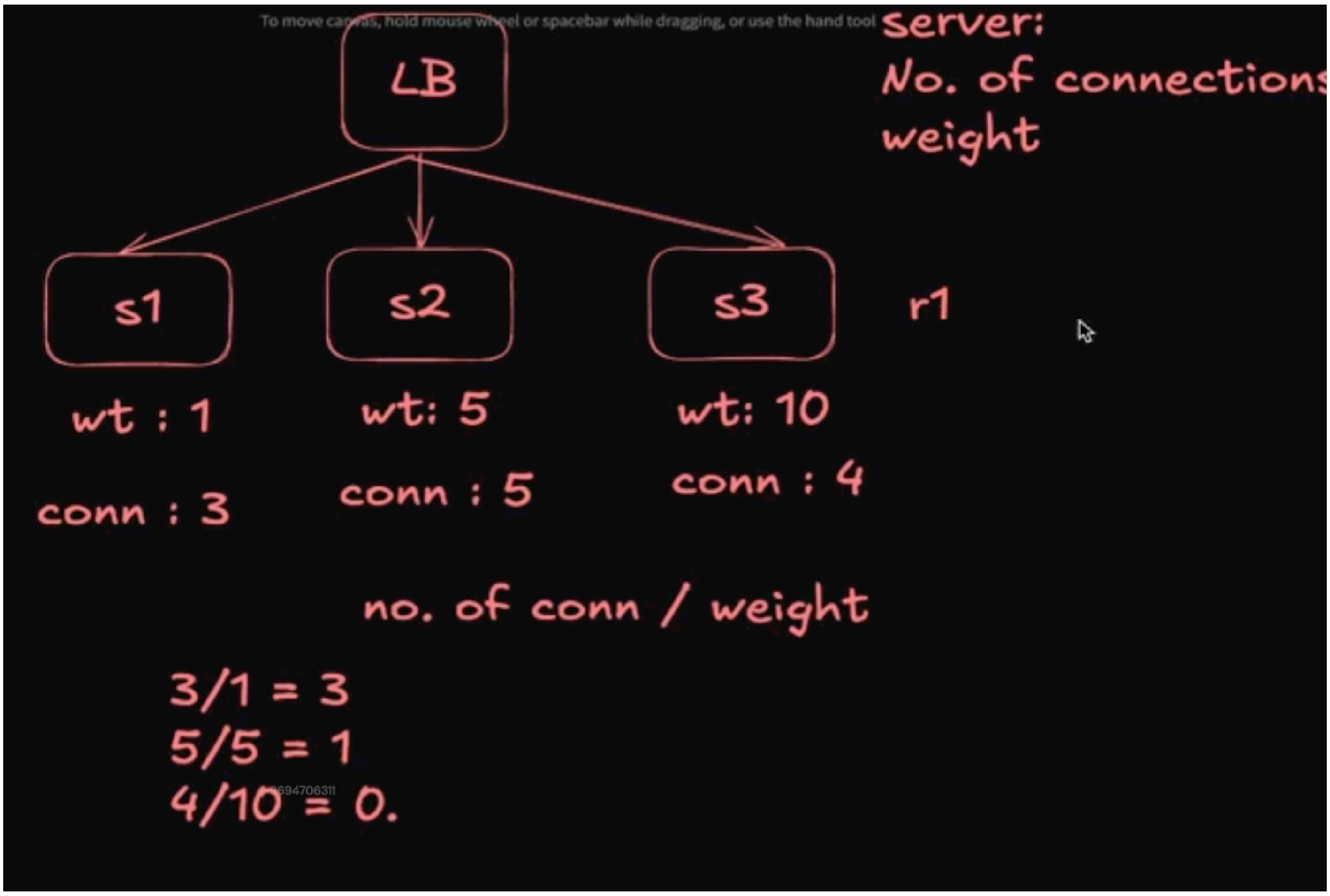
Limitation

- Doesn’t consider server power (all treated equal)

Best Use Case

- Chat apps
- APIs with long-running sessions

2.Weighted Least Connections



How it works

Like least connections, but also considers server capacity.

Example:

S1 → weight 3 (strong)

S2 → weight 1 (weak)

Traffic is adjusted based on weight + connections.

So strong servers handle more users.

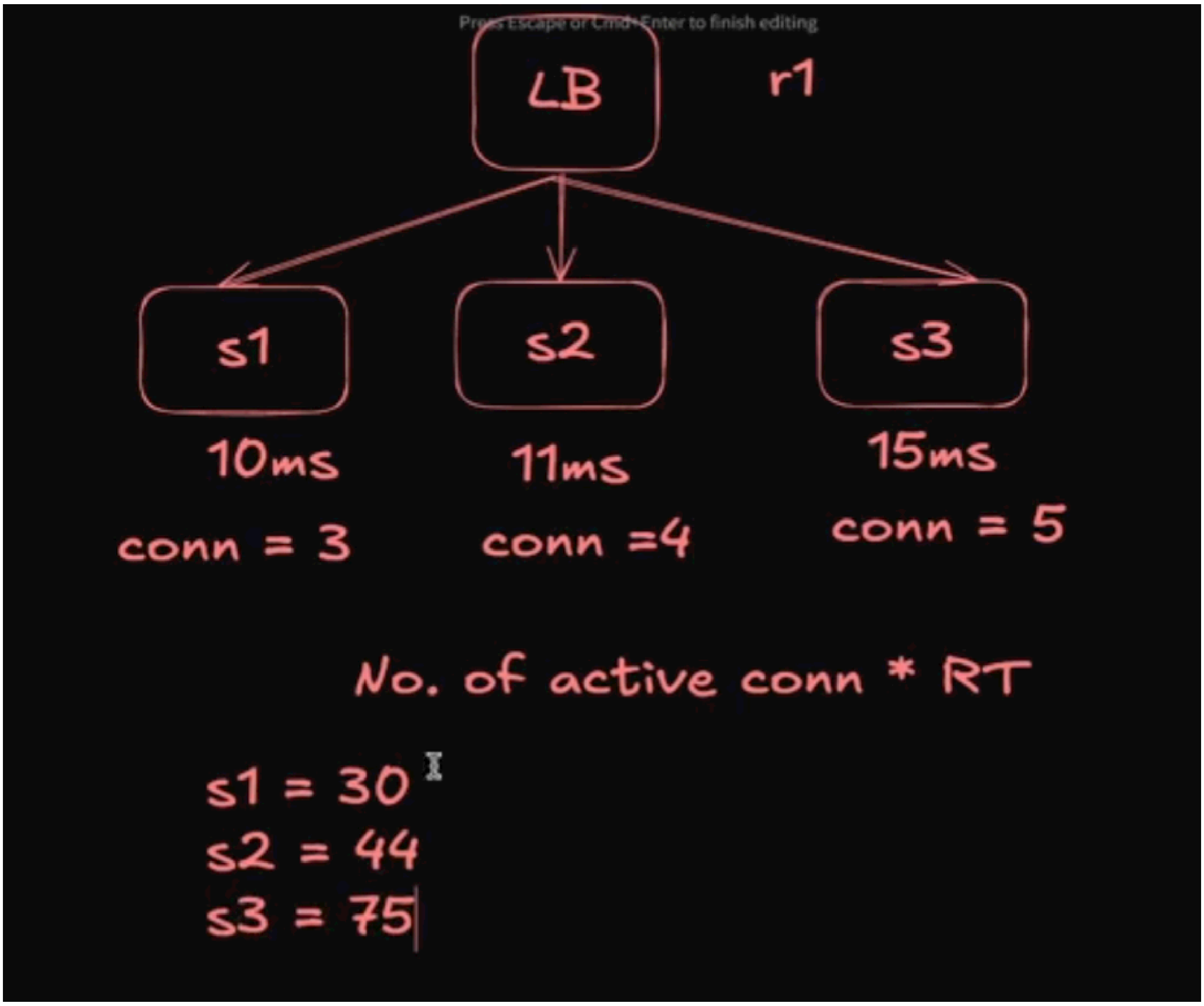
Why we use it

- Good when servers are not equal
- Prevents weak servers from overload

Best Use Case

- Cloud systems
- Large distributed apps

3.Least Response Time



How it works

Request goes to the server that is responding fastest.

Example:

S1 → 120ms

S2 → 30ms

S3 → 60ms

Next request → S2

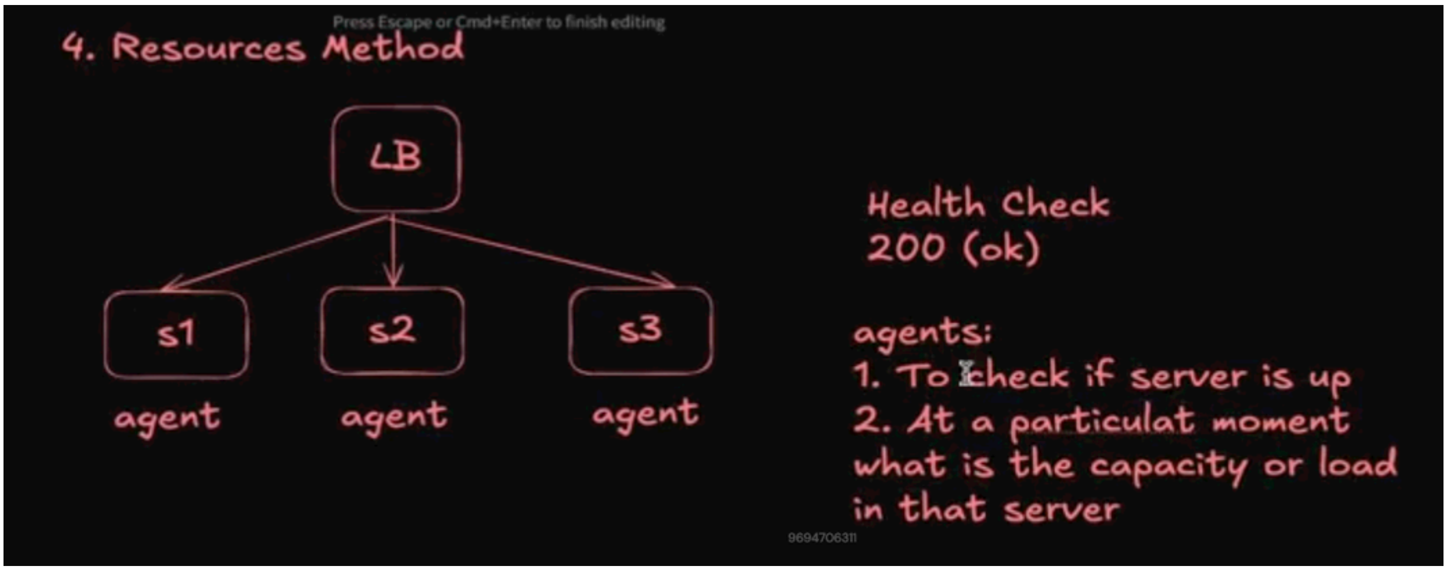
Why we use it

- Improves performance
- Reduces latency

Best Use Case

- Instagram feed
- Google search
- Gaming servers

4.Resource-Based Algorithm


Press Escape or Cmd+Enter to finish editing
4. Resources Method
LB
s1
s2
s3
agent
agent
agent
Health Check
200 (ok)
agents:
1. To check if server is up
2. At a particulat moment
what is the capacity or load
in that server
969470631l

How it works

Load balancer checks:

- CPU usage
- RAM usage
- Queue length

Then sends traffic to the server with most free resources.

Why we use it

- Best for large-scale systems
- Ensures efficient use of infrastructure

Best Use Case

- Cloud services
- High-traffic applications

Algorithm	Based On	Best For
Least Connections	Active users	Long sessions
Weighted Least Conn	Capacity + Load	Unequal servers
Least Response Time	Speed	High performance
Resource-Based	CPU/RAM	Cloud scale

Static LB	Dynamic LB
Pre-defined rules	Real-time decision
Simple but risky	Smarter and scalable
Good for small apps	Used in Instagram, Netflix, AWS