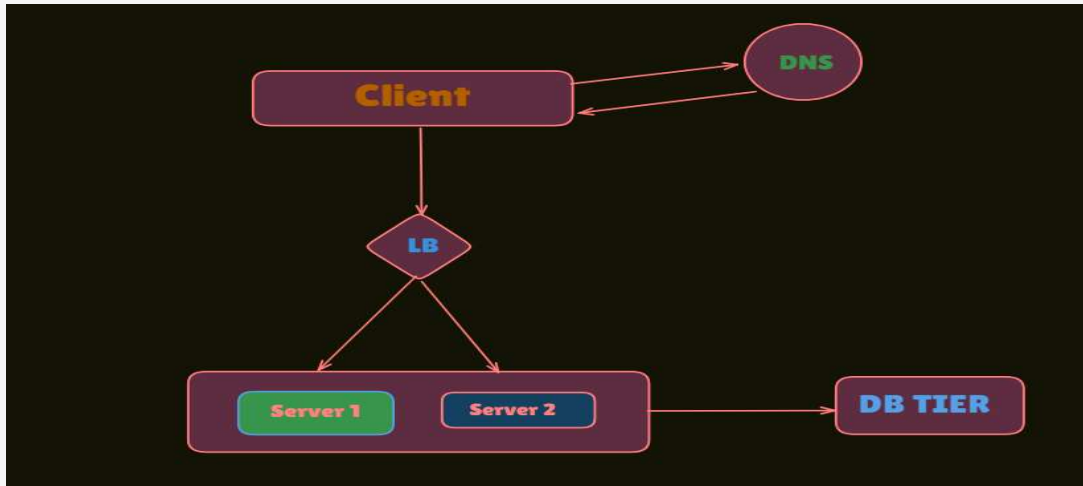


# Lecture 4 : Consistent Hashing | Load Balancer

---

## Load Balancer Overview:



### Flow Diagram:

Client → DNS → Load Balancer (LB) → Servers (S1, S2, S3...)

◆ Load Balancer का काम होता है – incoming requests को multiple servers में distribute करना।


---

## Replication (Master-Slave Architecture)

### Definition:

- Replication का मतलब: एक ही DB के multiple **replica (copies)** बनाना।
- जो data DB1 में है → वही DB2, DB3 में भी मौजूद होगा।

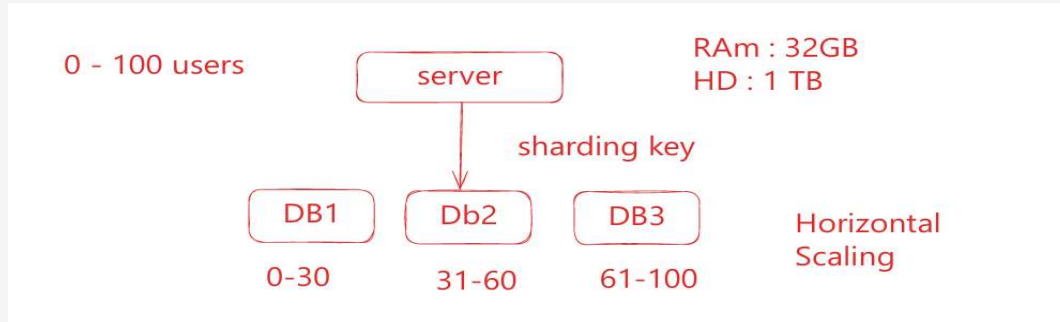
### Why Replication?

- यदि main DB crash हो जाए ❌ या respond न करे:
    - तब request को किसी **replica** की ओर redirect किया जा सकता है ✅
    -  **High Availability** सुनिश्चित होती है।
-

## Sharding (Horizontal Scaling)

RAM ya Hard Disk kabhi na kabhi full ho sakti hai, isliye Sharding ka use hota hai taaki data ko chhote-chhote parts (shards) mein tod kar alag-alag machines ya storage units par distribute kiya ja sake. Isse system scalable, fast aur efficient ban jata hai.

### Diagram :



📌 **Sharding Layer** → Data को range-based DBs में divide करता है:

Databas e	User ID Range
--------------	---------------

DB1	0 – 30
-----	--------

DB2	31 – 60
-----	---------

DB3	61 – 100
-----	----------

🧠 इससे हर DB का **load कम** हो जाता है

🔊 **Scalability** improve होती है

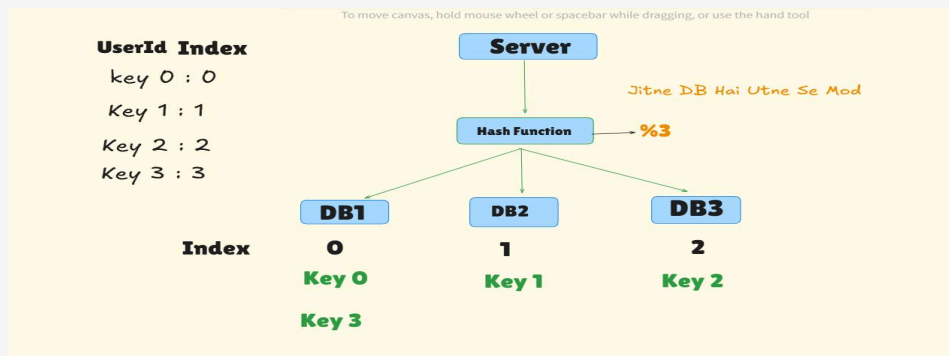
---

### 💡 Example Request Flow:

- यदि User ID = 32 → Request → **DB2**
- यदि User ID = 15 → Request → **DB1**

---

## Hashing Method – Module Operator Algorithm



## Examples:

User ID    Hash (%)    DB Index

40            40 % 4 = 0    DB0

3             3 % 4 = 3    DB3

12            12 % 4 = 0    DB0

43            43 % 4 = 3    DB3

Modulo operation से user requests को shard किया जाता है।

## ⚠ Problems with Traditional Sharding


### 📦 Load बढ़ने पर क्या होगा?

- कोई DB full होने लगे →
  - 🔧 एक नया **DB add** करना पड़ेगा
  - ⚠ या कोई server **down** हो जाए तो उसे **remove** करना पड़ेगा
  - 🧠 Solution: **Autoscaling** support होना चाहिए


## ⊖ Limitation:

- जब भी DB की संख्या बदलती है →
  - 🔄 **Hash Function** भी बदलता है
  - ⚠ पुरे DB का **Re-indexing + Migration** करना पड़ता है
  - 💰 High cost + complexity

## ✂ New DB Add करने पर क्या होगा? (e.g., DB4)

 Hash function: %3 → %4

 पूरा data **Redistribute** करना पड़ेगा

 **Re-hashing of all user IDs**

 **Heavy DB Migration**


---

## ❌ If a DB Goes Down (e.g., DB2)

 क्या होगा अगर DB2 डाउन हो गया?

- फिर से सभी keys को adjust करना पड़ेगा
  - Hash function वापस %3 करना होगा
  - Mapping change करनी पड़ेगी
  - 🧠 यह process बहुत complex और unreliable हो सकता है
- 

 **Key Takeaway:**

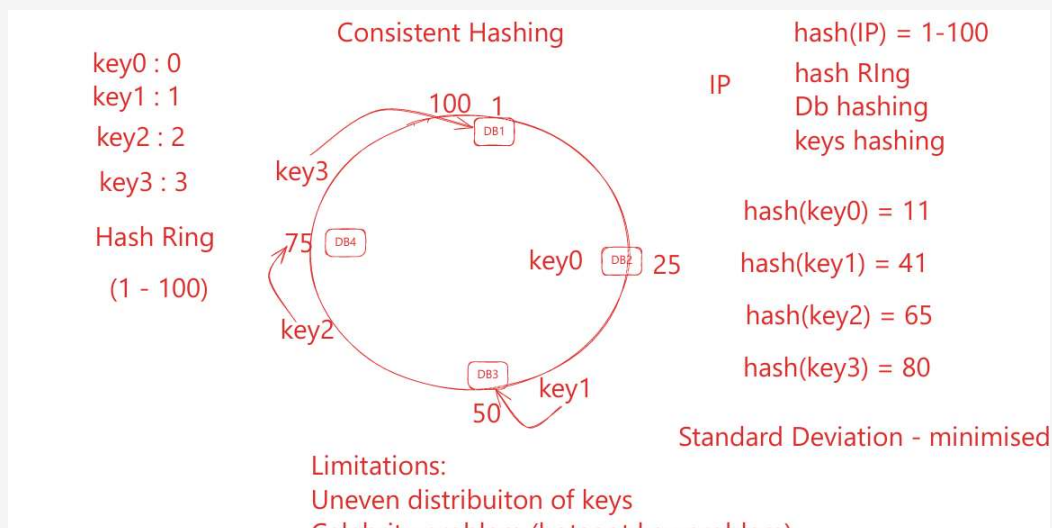
Traditional hashing-based sharding में **scaling** और **failure handling** बहुत costly और unstable हो जाता है, इसीलिए **Consistent Hashing** की जरूरत होती है (आगे cover होगा) 

## Consistent Hashing :

---

### Definition

- Consistent Hashing एक technique है जो **minimum data migration** के साथ scaling allow करता है (add/remove DB nodes).
- इसमें एक **Hash Ring (या Hash Circle)** का इस्तेमाल होता है:
  - जिसमें **keys** और **database nodes** दोनों को circle के points पर map किया जाता है.
  - Hash Ring का range होता है: 0 - 100



## 🎯 Key Mapping Logic

- किसी key को store करने के लिए:
  - उस key को hash किया जाता है → फिर clockwise direction में move करते हैं → जो पहला DB node मिलता है, वही उस key को handle करता है.

## ✚ Adding a New DB Node (e.g., DB5)

- DB5 को add किया गया और उसका **hashed position = 40**
- अब अगर **\*\*hash(key1) = 41\*\*** है
  - तो **key1**, जो पहले **DB3 (50)** को point कर रहा था
  - अब clockwise direction में **DB5 (40)** को point करेगा

➡ **Only key1 affected** (minimum migration)

➡ बाकी keys अपनी जगह बनी रहती हैं

## ⚠ Limitations of Consistent Hashing

### ! 1. Uneven Distribution of Keys

- जब keys के hash values में कम variation होता है →
  - ➡ सभी keys एक ही DB में map हो सकती हैं (e.g., all < 25 → DB2)
- ❌ **Load imbalance** होता है

- एक DB (e.g., DB2) hot spot बन जाता है
- बाकी DBs (DB1, DB3, DB4) **idle/underutilized** रहते हैं

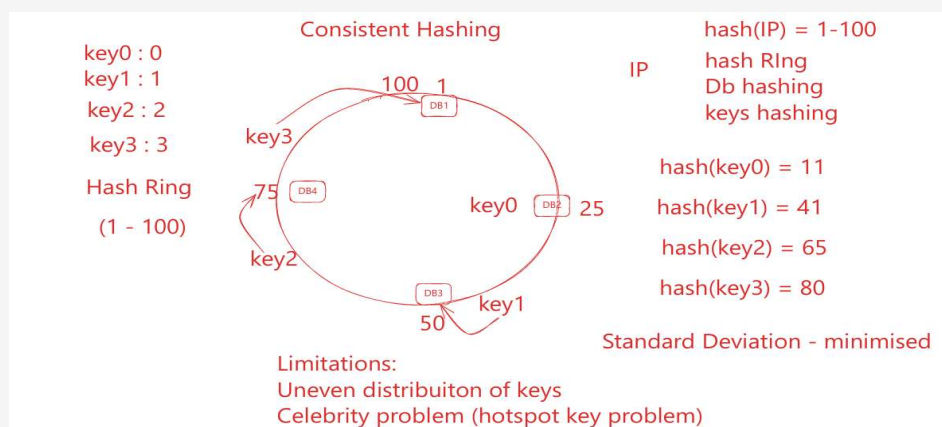
## ✨ 2. Celebrity Problem / Hotspot Key Problem

- कोई एक key पर बहुत ज़्यादा traffic आ रहा है
- Even with consistent hashing, वो key **एक ही node** को hit करता रहेगा
- Result:
  - ✨ **Single DB overload**
  - ▼ System-wide performance degrade

## 🔧 Solution: Virtual Nodes (VNodes)

### ✅ What Are Virtual Nodes?

- हर physical DB को multiple **virtual nodes (or replicas)** के रूप में represent किया जाता है
- Example:
  - DB1 → DB1-1, DB1-2
  - DB2 → DB2-1, DB2-2 ...



### 🔄 How Do They Work?

- ये virtual nodes hash ring पर अलग-अलग जगह पर distribute होते हैं
- इससे **database presence density** बढ़ जाती है

---

### 🧠 Why is This a Solution?

- Keys अब ज्यादा evenly distribute होते हैं
- एक ही region में multiple DBs के virtual nodes हो सकते हैं
  - इससे **load spread** होता है
  - **Hot spot** बनने की problem कम होती है
  - **Low standard deviation** issue solve होता है

## ⚖️ Load Balancing

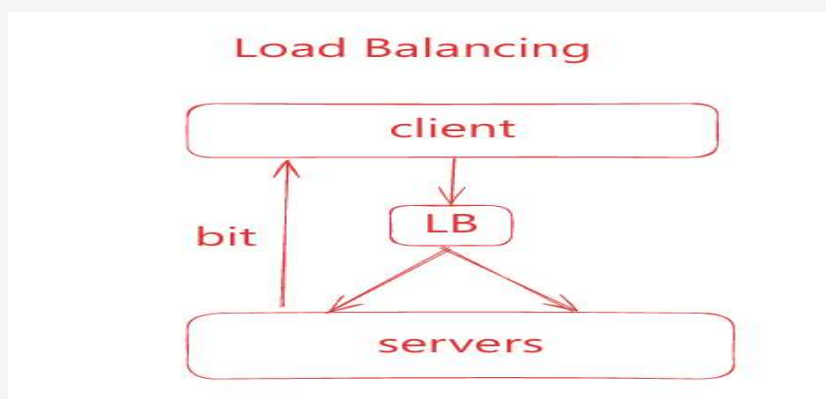
---

### 📌 What is It?

- Load Balancing एक ऐसी technique है जो **incoming traffic** को multiple **backend servers** में distribute करती है
- इसका मकसद है कि हर server पर **balanced load** रहे और कोई भी server **overloaded** न हो

#### 📋 Diagram Flow:

Client → Load Balancer (LB) → Server (S1, S2, S3...)



### 🎯 Key Function

- Load Balancer यह decide करता है कि:
  - कौन सा server **available** है


- कौन सा server **handle** कर सकता है request
- इसके बाद वो request को उसी server पर भेजता है

- ➡ इससे कोई single server overload नहीं होता
  - ➡ System की **efficiency** और **availability** बढ़ती है
- 

## Applications of Load Balancing

---



### 1. Scalability

-  **Autoscaling** supported
  - Traffic बढ़ने या घटने पर, automatically server pool को **scale up/down** किया जा सकता है
  - ⚡ Efficient handling of **fluctuating traffic**
- 

### 2. Availability

- यदि कोई server **fail** हो जाता है:
    - Load Balancer उसे detect करता है
    - Requests को उस server पर भेजना **बंद** कर देता है
  - 💡 Ensures **High Availability** and **Zero Downtime**
- 

### 3. Security

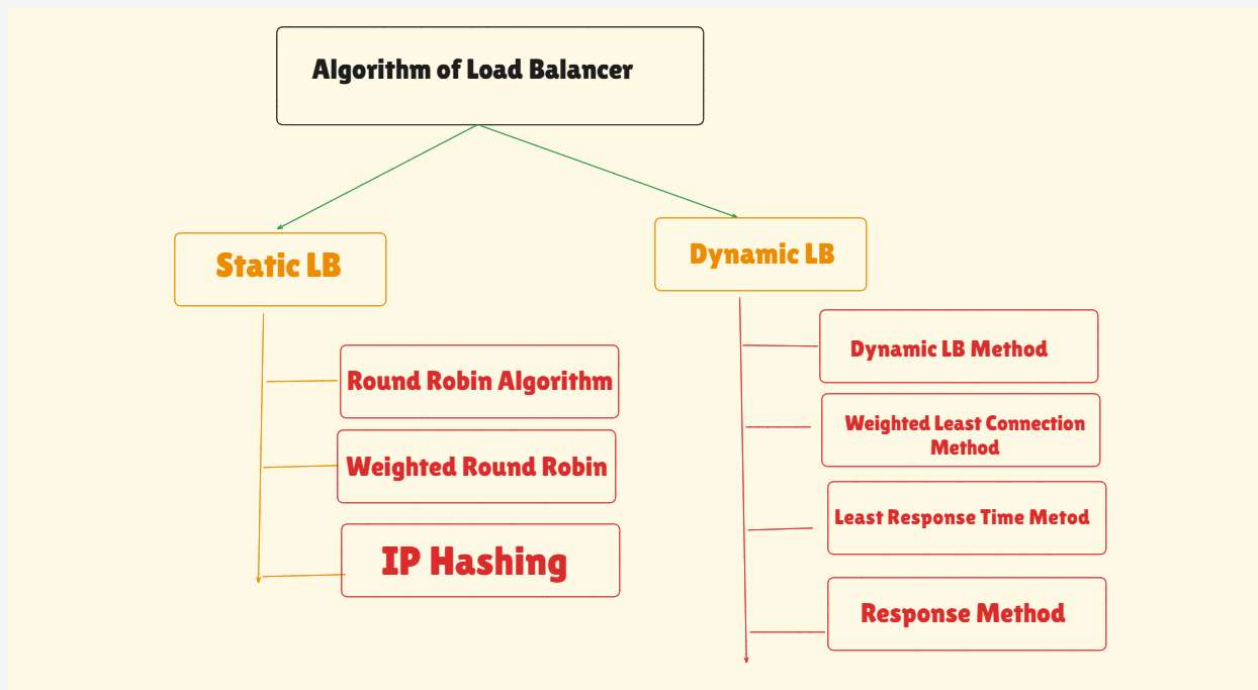
-  **DDoS Attack Prevention:**
    - Malicious traffic को filter करता है
    - Backend servers तक नहीं पहुंचने देता
  -  **Firewall Integration:**
    - Bot, Malware जैसी unwanted requests को block करता है
-



## ⚡ 4. Responsiveness

- Requests का load multiple servers में divide होता है
- 🔄 इससे servers fast respond करते हैं
- 🔄 Better **user experience** ensure होता है

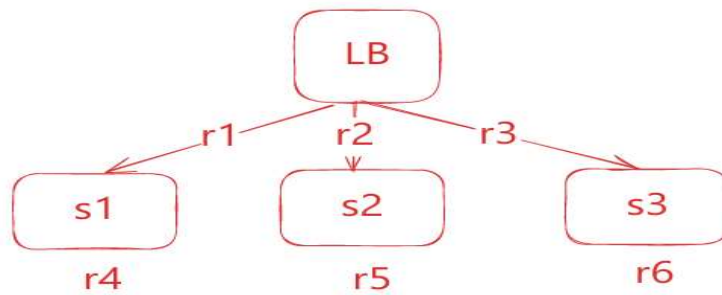
## 📊 Algorithms of Load Balancing



## 📁 Static Load Balancing Algorithms

### 🔄 1. Round Robin Algorithm :


## 1. Round Robin Algorithm



Limitations:

Do not consider load on any server

### How it Works:

- सबसे simple और easy-to-implement algorithm
- Requests को **sequentially** servers को assign करता है:
  - Request 1 → Server 1
  - Request 2 → Server 2
  - Request 3 → Server 3
  - फिर से घूमा कर: Request 4 → Server 1 ... and so on 

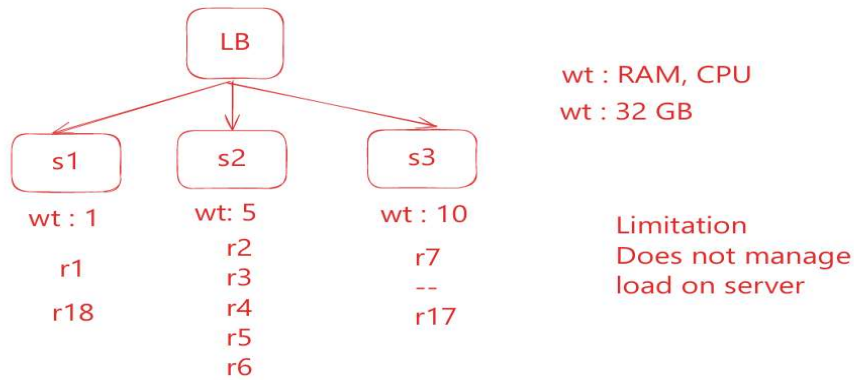
### Limitation:

- Server के **current load** को ध्यान में नहीं रखता ✗
- सभी servers को equal request देता है, चाहे वो busy हो या free
- ✓ केवल **small applications** के लिए सही है



## 2. Weighted Round Robin Algorithm :

## 2. Weighted Round Robin Algorithm



### 📌 How it Works:

- हर server को उसकी capacity (RAM, CPU etc.) के हिसाब से एक **weight** दिया जाता है
- ज्यादा weight वाले server को ज्यादा requests मिलती हैं

### 📊 Example:

Server	Weight	Request Distribution (Proportional)
S1	3	🔄 3 requests
S2	5	🔄 5 requests
S3	10	🔄 10 requests

➡ Total requests 18 हों तो distribution इसी ratio में होगा

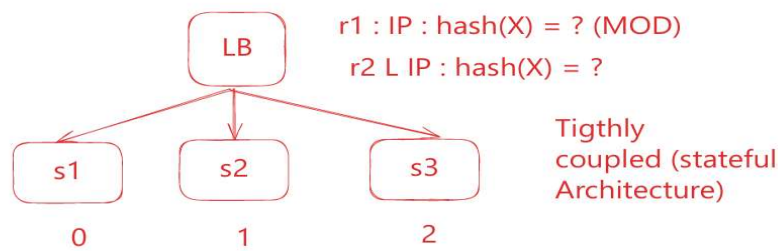
### ⚠ Limitation:

- Server के **real-time load** को monitor नहीं करता
- Weight fixed रहता है, dynamic load variation को handle नहीं करता

---

## 🌐 3. IP Hashing Algorithm :

### 3. IP Hashing



Limitation : Each user will always get same server  
2. Proxy (VPN) : IP

### How it Works:

- Client के **IP address** को hash करके determine करता है कि उसे कौन से server पर भेजना है

Formula:

```
server_index = hash(client_IP) % number_of_servers
```

- 

 इससे एक ही client को हमेशा एक ही server पर भेजा जाता है (जब तक infra same हो)

### Limitation:

- **Hotspot Problem:**
  - अगर कई clients **same public IP** share कर रहे हैं (e.g., office या college network)
  - तो सबकी requests एक ही server को जाएंगी → वो server **overload** हो सकता है
  - बाकी servers underutilized रहेंगे

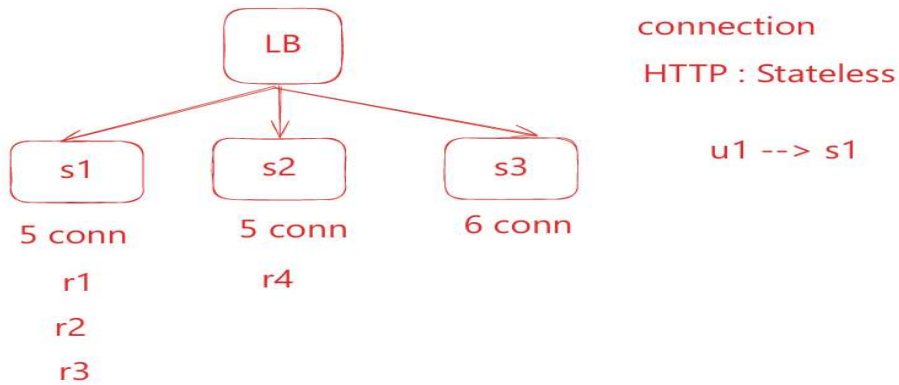
## Dynamic Load Balancing Algorithms :

### Definition:

Dynamic algorithms request distribution का decision **real-time server state** के आधार पर लेते हैं — जैसे कि active connections, response time, या server health.

### 1. Least Connection Method :

### 1. Least Connection method



#### 📌 How it Works:

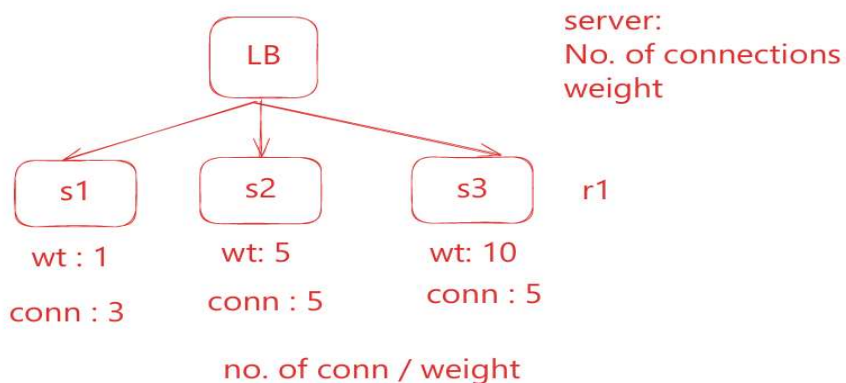
- Request उस server को भेजी जाती है जिसके पास **सबसे कम active connections** हैं
- Best suited when:
  - सभी connections का size similar हो
  - Connections **long-lived** हों

#### ✅ Advantage:

- Uneven load distribution को avoid करता है
- अधिक busy servers को बचाता है

## ⚖️ 2. Weighted Least Connection Method :

### 2. Weighted Least connection method



#### 📌 How it Works:

- Least Connection का enhanced version है
- हर server को एक **weight** assign किया जाता है (RAM, CPU के हिसाब से)

Load balancer calculate करता है:

$\text{active\_connections} / \text{weight}$

- और उसी ratio के अनुसार request को assign करता है

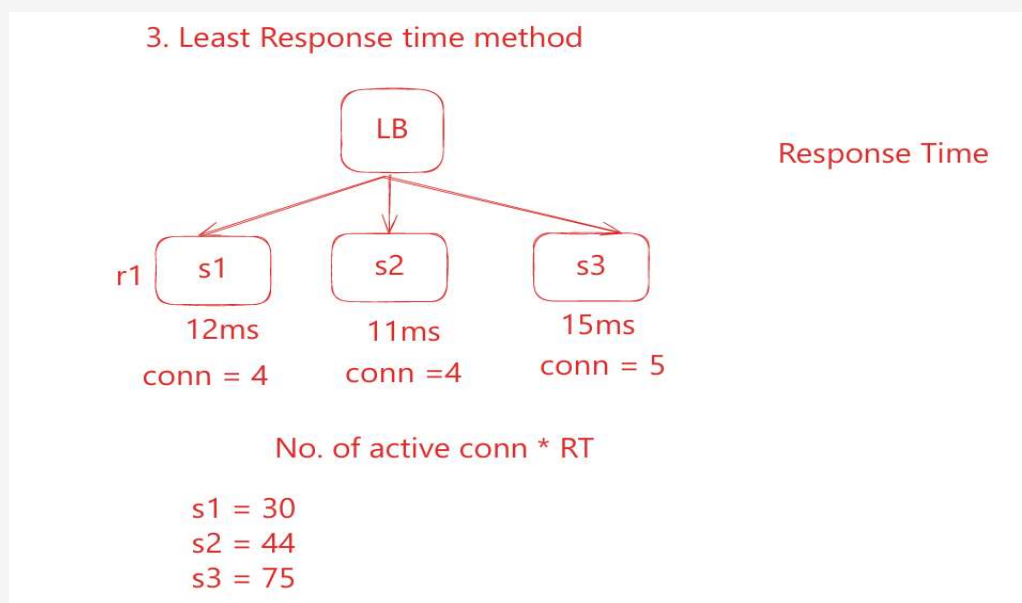
### Example:

अगर S1 का weight 5 है और S2 का 10, और दोनों पर 10 connections हैं, तो S2 को next request मिलेगी क्योंकि:

S1:  $10/5 = 2.0$

S2:  $10/10 = 1.0 \rightarrow \text{lower ratio}$

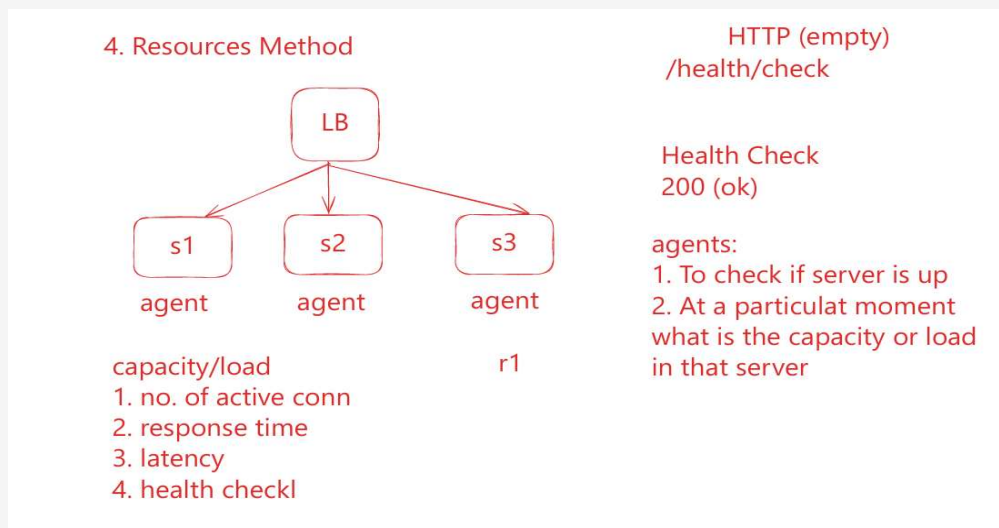
## 3. Least Response Time Method :



### How it Works:

- Request को उस server पर भेजा जाता है जिसका **response time** सबसे कम है
- Response time को health check के माध्यम से measure किया जाता है
- Useful for:
  - High traffic websites
  - Applications जहाँ **low latency** critical है

## 4. Resource Method :



### How it Works:

- Load balancer हर server के **live resources** को monitor करता है:
  - CPU, RAM, Active Connections, Response Time, Latency
- जिस server के पास सबसे **कम load** होता है, request वहीं जाती है

### Agents Involved:

- हर server पर एक **Agent** होता है
- ये agent real-time metrics collect करता है और load balancer को भेजता है

## Health Checks

### Purpose:

- Determine करता है कि कोई server **"up"** है या **"down"**

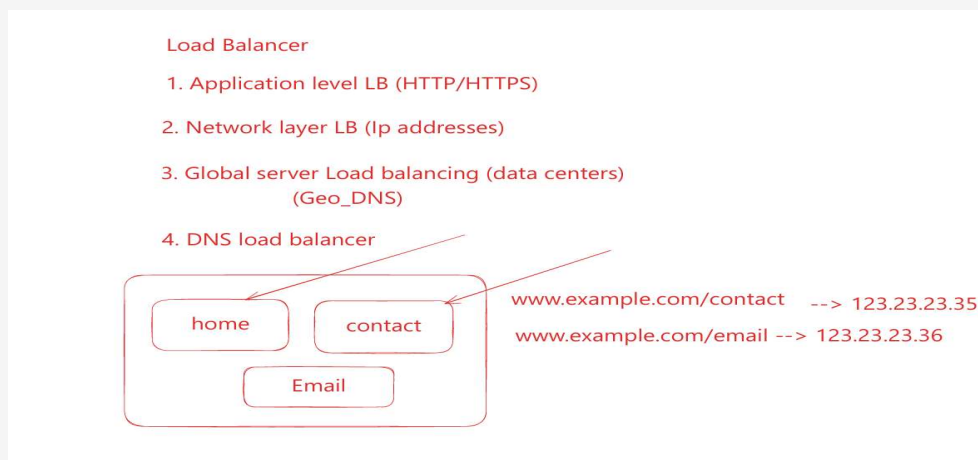
### How it Works:

- Load balancer server को **HTTP request** भेजता है (e.g., **/health**)
- अगर response: **HTTP 200 OK** आता है → server healthy मना जाता है
- अगर health check **fail** हो जाता है →
  -  Load balancer उस server को request भेजना **बंद** कर देता है

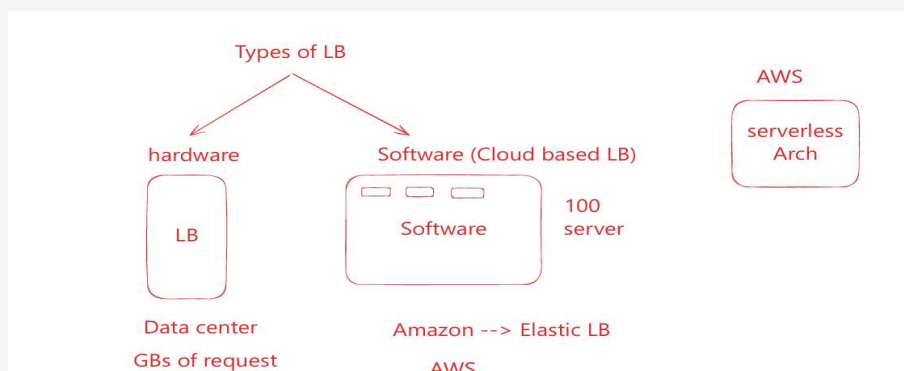
## 🧠 Why Important?

- Faulty या dead servers को avoid करने से:
  - System की **availability** बनी रहती है
  - Users को better experience मिलता है

## 📁 FOCUS OF LOAD BALANCER : ON THE BASIS OF LAYERS :



## TYPES OF LB :



## Summary :

🔄 Load Balancer



- Request को multiple servers में distribute करता है
  - Ensures: 🏗️ Scalability | 🛡️ Availability | 🔒 Security | ⚡ Fast Response
- 

## 🔄 Replication (Master-Slave)

- Same data in multiple DBs
  - 🛡️ Backup in case main DB fails
- 

## 📦 Sharding (Horizontal Scaling)

- Data split into ranges (DB1: 0–30, DB2: 31–60...)
  - 📈 Improves performance and scalability
  - ❌ Adding/removing DB → full data migration
- 

## 🔄 Consistent Hashing

- Hash Ring (0–100): Keys & DBs placed on ring
- Clockwise mapping to nearest DB
- ➕ Add DB → only few keys move
- ✅ Less migration
- ❌ Issues: Uneven key distribution, Hotspot key

🔧 Fix: Virtual Nodes → Better load distribution

---

## ⚖️ Load Balancing Algorithms



### 📁 Static:

1. **Round Robin** – Turn-wise request allocation
2. **Weighted Round Robin** – Based on server capacity
3. **IP Hashing** – Same IP → same server (may cause hotspot)

## **Dynamic:**

1. **Least Connection** – Fewest active connections
  2. **Weighted Least Connection** –  $\text{Load} \div \text{weight}$
  3. **Least Response Time** – Fastest responding server
  4. **Resource Method** – Based on CPU, RAM, etc. via agents
- 

## **Health Checks**

- Server को periodic HTTP check भेजना (e.g., `/health`)
  -  200 OK → Healthy |  Fail → Excluded from routing
- 

## **Focus of Load Balancer:**

- Scalability | Availability | Fault Tolerance | Real-Time Routing