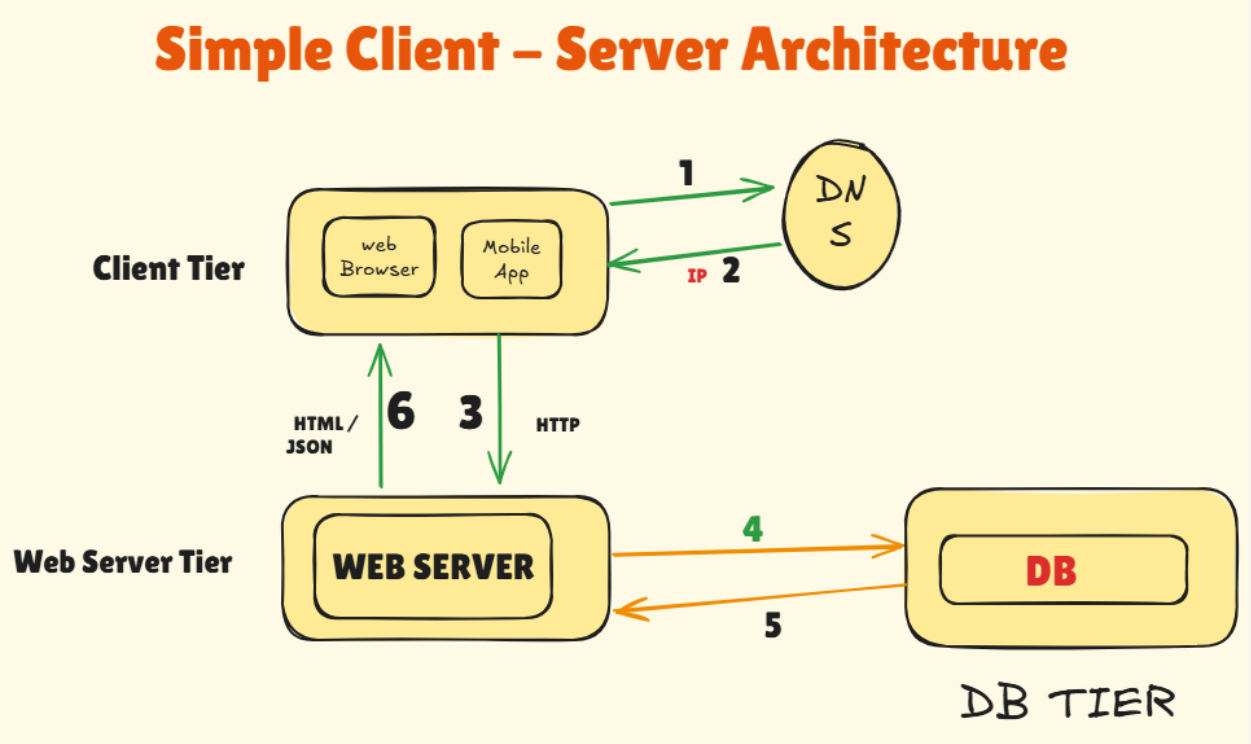
📘 Lecture 2: Scaling from 0 → Million Users

🔹 **Main Goal of HLD** → Application should **scale better** with increasing users.

---

① Simple Architecture (Starting Point)

👉 **Single Client–Server Architecture**



📌 Flow:

1. **Client** (Web Browser / Mobile App) request करता है.
2. **DNS** → Domain (e.g., `www.example.com`) को IP में convert करता है (e.g., `223.23.23.23`).
3. **HTTP Protocol** used for communication: **Iske inside jata kya hai**
    - Methods: `GET`, `POST`, `PUT`, `DELETE`
    - URL: `www.example.com`
    - Headers: Extra info (Auth, Content-Type, etc.)

- Body: JSON format में data जाता है

Example Request Body:

```
{
  "userId": 1,
  "uName": "aditya"
}
```

4. **Web Server** request को process करता है:
    - Data fetch करना
    - Data store करना
    - कोई भी logic execute करना
5. Server → Client को **Response** भेजता है:
    - HTML
    - JSON

---

② Limitations of Simple Architecture

⚠️ Issue:

- केवल एक **Web Server** है.
- उसी server के साथ database भी integrated है.
- ज्यादा load / users आने पर application scale नहीं कर पाएगी.

👉 Example:
अगर हम अपनी **YouTube** या **Instagram** जैसी **application** इस model पर बनाएंगे → तो ये कुछ users के बाद ही crash हो सकती है।
क्योंकि एक server ही request handle कर रहा है और उसी में database भी है।

👉 Solution:

- Database को अलग **Server** पर deploy करो.
- इससे Application server और Database server अलग-अलग काम करेंगे.

⚠️ **Current Limitation** → Database web server के अंदर ही है → Scalability problem.
✅ **Improvement (Next Step)** → Database को अलग server पर निकालना.

Introducing Database Layer

---

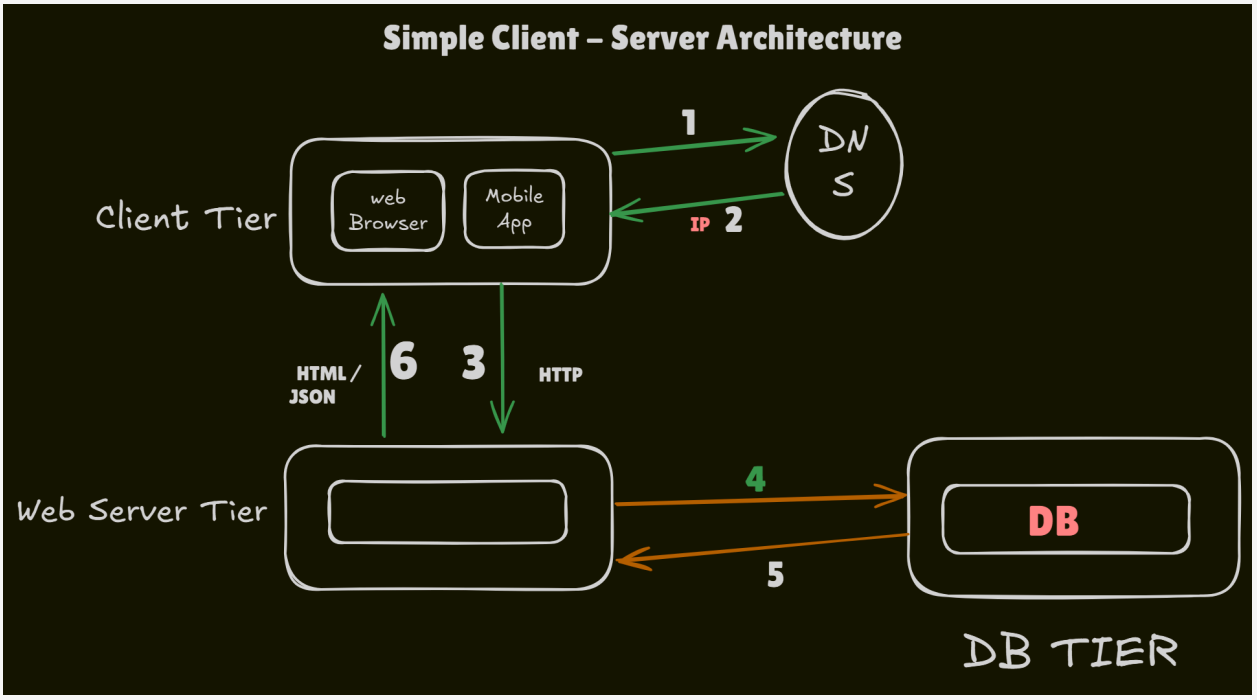
② Introduce DB Layer

👉 अब हम अपनी application में **Database Server** को introduce करते हैं।

📌 **Working:**

1. Client (Web Browser / Mobile App) → Web Server को Request भेजता है।
2. Web Server → Database Server से Data fetch/store करता है।
3. Database → Response देता है Web Server को।
4. Web Server → Client को **HTML / JSON Response** return करता है।

📊 DIAGRAM: With DB Layer



✅ **Benefit:**

- अब application split हो गई है → scalability improve हो गई है।

- Future में अगर ज़रूरत पड़े तो:
    - Web Server को scale कर सकते हैं।
    - Database Server को भी scale कर सकते हैं।

⚡ Types of Databases

① Relational DB (SQL Based)

- Example: **MySQL, PostgreSQL, Oracle SQL**
- Structured data के लिए use होता है।
- Data relations clear होते हैं।
- Query optimize और indexing possible।

---

② Non-Relational DB (NoSQL)

चार प्रकार के DB आते हैं:

1. **Key-Value Store** → Amazon DynamoDB
    - Data store होता है key-value pair में।
2. **Column Based Store** → Cassandra DB
    - Data को column-wise store करता है।
3. **Document Store** → MongoDB
    - Data JSON format में store होता है।
4. **Graph Store** → GraphQL
    - Data relationships को graph structure में store करता है।
    - Example: LinkedIn, Facebook, Instagram (Follow / Friend connections → Bi-directional graph).

🚨 Problem: Wrong DB Selection

❌ अगर आप पूरा JSON response को SQL Database में store करेंगे →

- Queries optimized नहीं होंगी।
- Data fetch slow होगा।
- Relational DB unstructured data के लिए efficient नहीं है।

🧠 How to Decide: Relational vs Non-Relational?

👉 Step 1: **Check Data Structure**

- **Structured Data** → Use **Relational DB** (SQL).
- **Unstructured Data** → Use **Non-Relational DB** (MongoDB, Cassandra).

👉 Step 2: **Check Response Time**

- **Fast Response Needed** → Use **Non-Relational DB**.
- **Can work with slower response** → Use **Relational DB**.

---

③ Server की Limitation aa hi jayegi ?

- एक web server की capacity fix होती है।
    - Normal → **8 GB – 16 GB RAM**
    - Max scale → **32 GB** तक
- Limitation → Infinitely scale नहीं कर सकते।

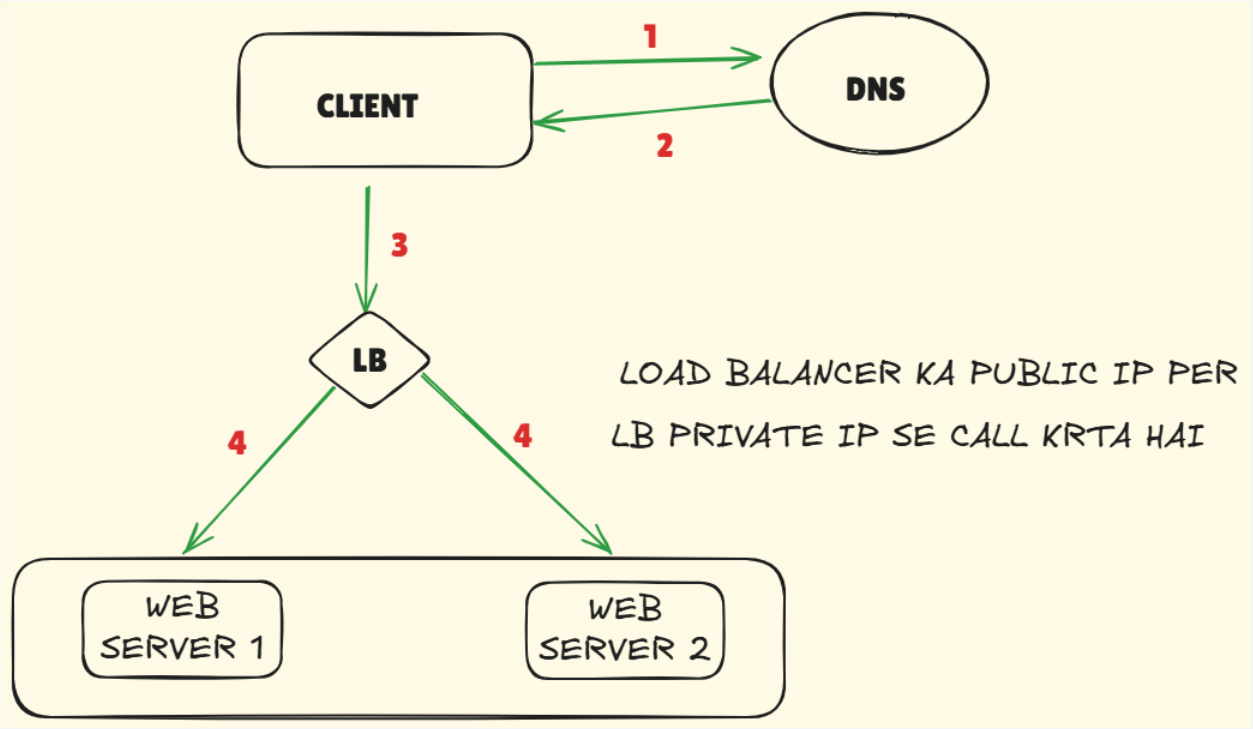
Single Point of Failure (SPOF)

- अगर एक ही **server down** हो गया → पूरी **application down**।
- इसलिए **Horizontal Scaling** करते हैं → multiple servers जोड़कर।

Multiple Servers Architecture

- Client request को **multiple servers** में distribute किया जाता है।
- Users का request किस server पर जाएगा?
    - ये decide करेगा **Load Balancer**।

📊 DIAGRAM: MULTIPLE SERVER ACHITECTURE



Load Balancer

- Client → DNS → **Public IP (Load Balancer)**
- Load Balancer request को
    - Web Server 1 (**10.0.0.1**) या
    - Web Server 2 (**10.0.0.2**) पर भेजेगा।
- अगर किसी server पर ज्यादा load है → traffic को automatically दूसरे server पर redirect कर देगा।
- DNS हमेशा **Load Balancer** का **IP** देगा।
- Protection → server को **direct call** नहीं कर सकते।

Database Failure

- Database भी **SPOF (Single Point of Failure)** हो सकता है।
- अगर DB fail हो गया → पूरा **Data loss**।
- Solution → **Database Scaling (Replication)**
    - Multiple copies of DB → एक fail हो तो दूसरा handle कर ले।

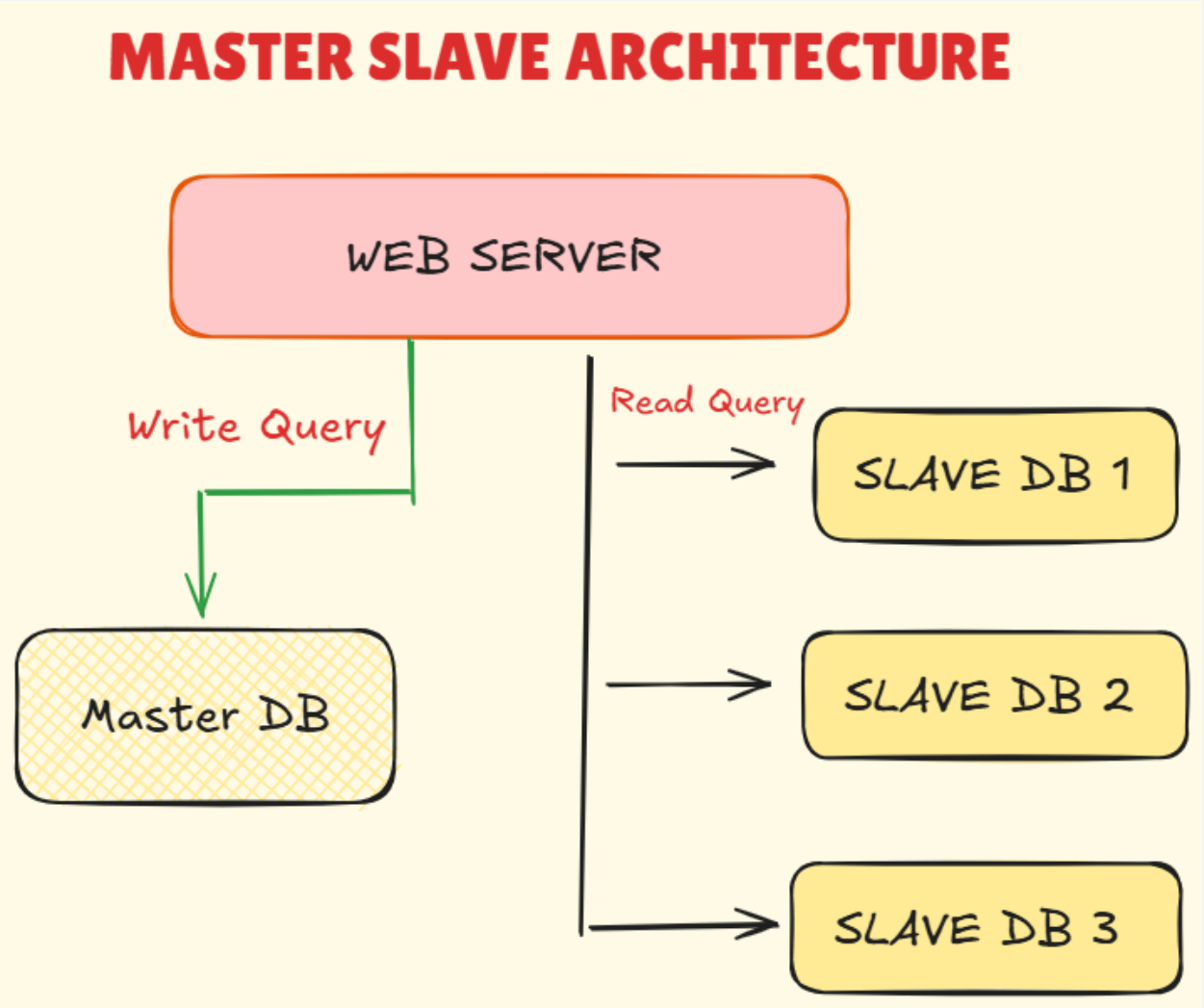
④ Database Scaling

Problem – Single DB Failure

- अगर एक ही **Database fail** हो गया → पूरा **Data loss**।
- इसलिए **Master-Slave Architecture** introduce करते हैं।

🔹 Master–Slave Architecture

- **Write Query** → **Master DB**
- **Read Query** → **Slave DBs** (Slave DB1, Slave DB2, Slave DB3)
- Master DB का काम → Update करना और **time-to-time Slaves** तक **data replicate** करना।
- Slave DB internally **Load Balancer** use करता है → read queries को distribute करने के लिए।
- Slave DB1 का data backup → Slave DB2 और Slave DB3 में।



📌 **Problem –** अगर **Master DB down** हो जाए?

- Write queries रुक जाएँगी।
- Master से Slaves तक update नहीं पहुँच पाएगा।

👉 **Solution:**

- किसी Slave को **Temporary Master** बना दिया जाता है।
- Slave DBs voting algorithm से decide करते हैं कि कौन Slave **temp master** बनेगा।
- इसे **Temporary Master/Slave Promotion** कहते हैं।
- जब तक original Master वापस fix या नया Master deploy न हो, temp master काम करता है।

⚡ Benefit → अब Database में **Single Point of Failure (SPOF)** नहीं रहा।

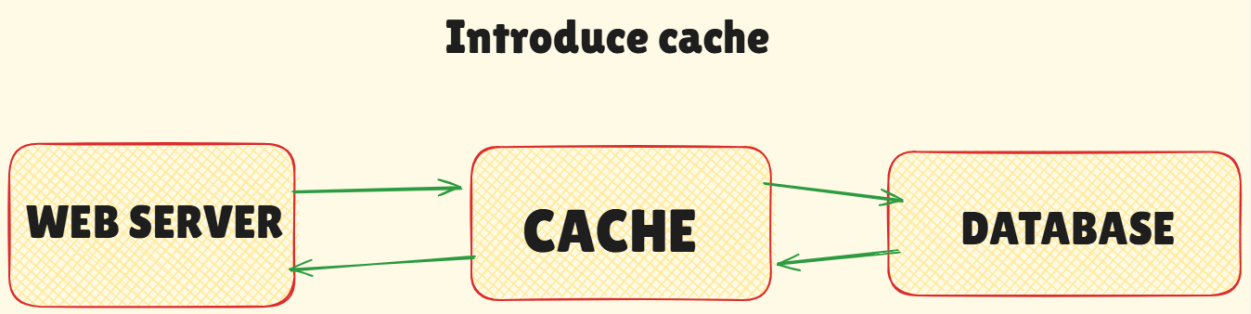
🔹 **Web Server + DB Tier**

- Web Server → DB Tier (Master + Slaves) से जुड़ा होता है।
- अगर future में कोई DB down हो भी जाए, तब भी बाकी servers से data मिलता रहेगा।

---

⑤ Introduce Cache

- अब भी millions of users को handle करने में lag हो सकता है → इसलिए introduce करते हैं **Cache**।



📌 **Cache Flow:**

- Web Server पहले Cache से data check करेगा।
- अगर data **Cache** में **available** है → वहीं से **fast response** मिलेगा।
- अगर data **Cache missing** है → **DB** से **fetch** करके **Cache** में **store** किया जाएगा।
- इससे DB पर unnecessary load नहीं पड़ेगा।

✅ Result → Database operations कम होंगे और response time fast मिलेगा।

⑥ Problem: Static Content ज्यादा होना

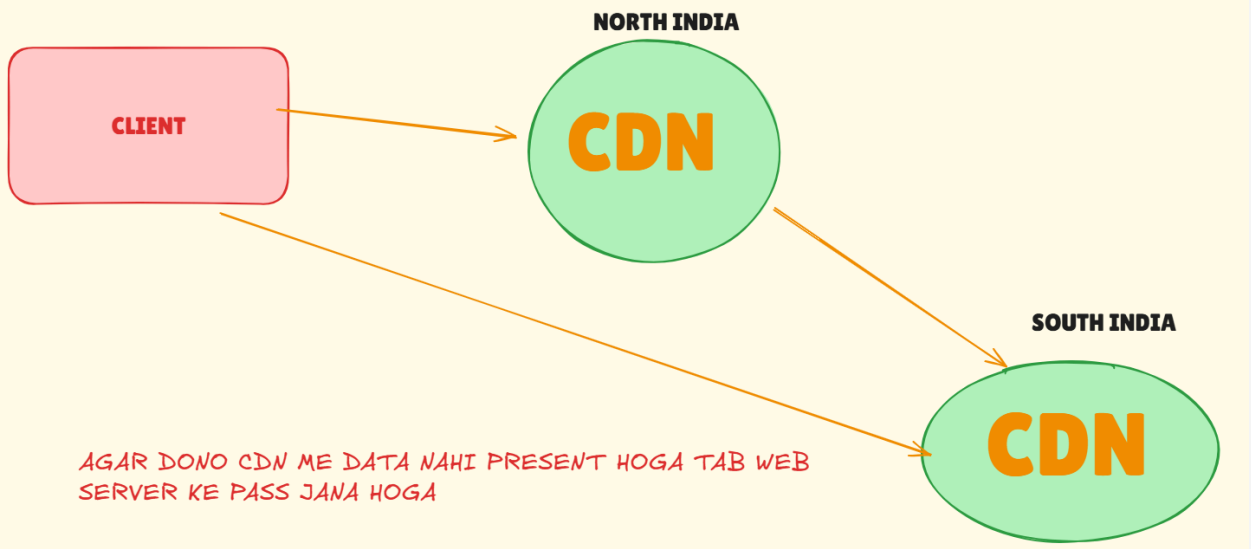
Static Pages

- Same **HTML, CSS, Images, JS files** बार-बार load होती हैं।
- हर बार server को hit करना inefficient है।

✅ Solution: CDN (Content Delivery Network)

- **CDN = Third Party Vendor + Proxy**
- Proxy मतलब → Client को लगता है server से बात हो रही है, लेकिन वास्तव में वो **CDN** से बात करता है।
- CDN का काम → Static content (HTML, CSS, JS, Images) को fast deliver करना।

CDN DIAGRAM :



📌 **CDN** कैसे काम करता है**:**

1. Client की request पहले **CDN** तक जाती है।
2. अगर content **CDN** में **available** है → **fast response**।
3. अगर content **CDN** में नहीं है → **Web Server** से **fetch** करके **cache** करता है।
4. CDN अलग-अलग locations पर मौजूद होते हैं (edge servers)।

⚡ **Examples:**

- Amazon → CloudFront
- Akamai
- Netflix → Movies को region-wise CDN पर store करता है।
    - इंडिया के users की request → **India CDN** से serve होगी।

📌 **DNS + CDN Integration:**

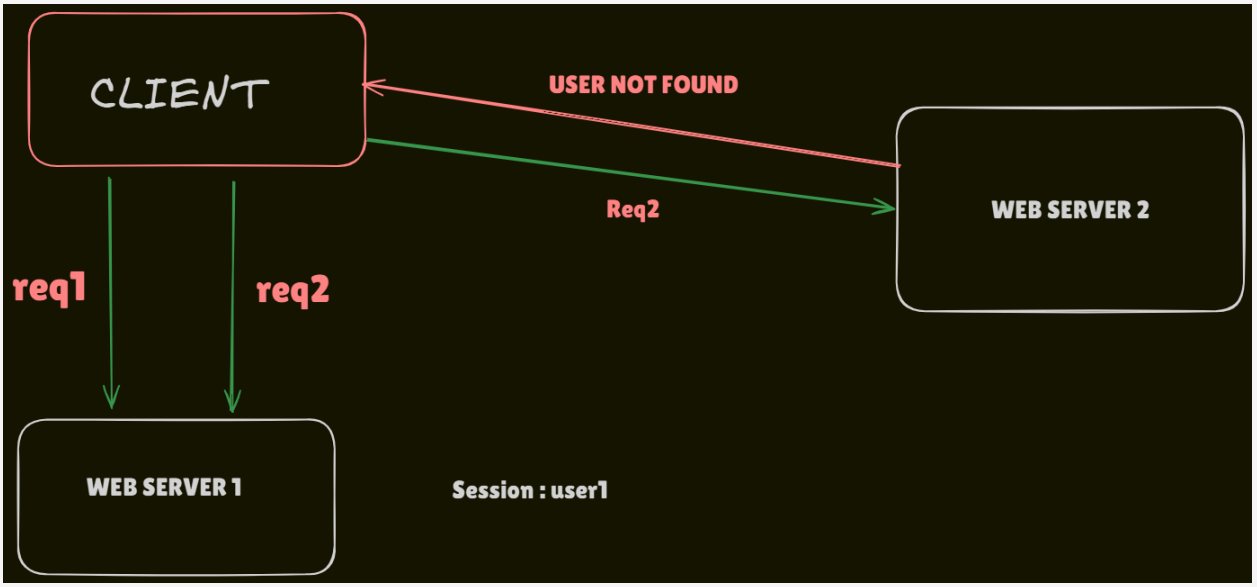
- Client जब request करता है → **DNS client** को **CDN** का **IP** देता है।
- Result → Client सीधे CDN से connect करता है, न कि main Web Server से।

---

⑦ Problem: Auto-Scaling Not Possible

- Web server की auto-scale नहीं हो पा रही है।
- ज्यादा request आने पर server scale कर तो जाएगा, लेकिन **client sessions** खो जाएंगे।
- Reason: **HTTP is stateless protocol** → हर request independent होती है।

Client Session ARCHITECTURE :



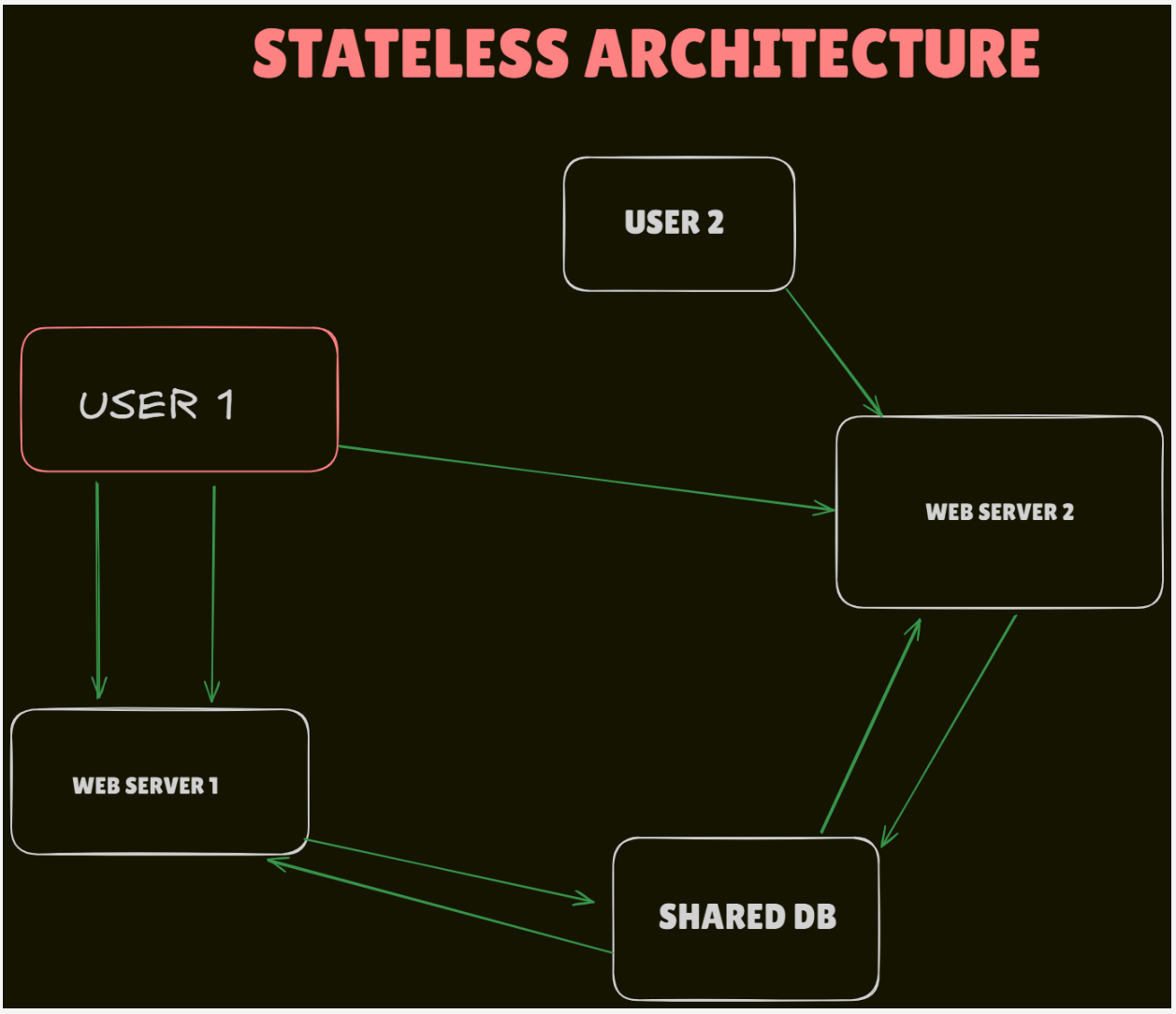
- **Definition:** कोई भी ऐसी information जो server को client को पहचानने के लिए रखनी पड़ती है।
- Example: Login state, cart items, user preferences.

📌 अगर session एक server पर है और अगली request दूसरे server पर चली गई → तो नया server client को नया **user** समझेगा।

- REQ 1 → Web Server 1 (Login Success)
- REQ 2 → Web Server 2 (Session Info missing → Fail)
- REQ 3 → Web Server 3 (Again missing → Fail)

👉 इसे ही कहते हैं **Stateful Architecture Problem**

Stateless Architecture → Solution



- Architecture को **Stateful** → **Stateless** बनाना पड़ेगा।
- यानी session को किसी एक server पर रखने की बजाय **shared place** पर रखा जाए।

✅ Shared DB Approach

- सभी web servers एक **shared DB** से connect रहते हैं।
- Session DB में store होता है → इसलिए कोई भी web server request handle कर सकता है।
- Web server scale करना आसान हो जाता है।

Cookies Concept

- कुछ session info server पर रखने की बजाय **client browser** में रखी जाती है।
- Server client को पहली बार request पर **cookies send** करता है।

- Future requests में client वही cookies server को भेजता है।

📌 Example:

- अगर user ने cart में item add किया → वो info cookie में stored रहेगी।
- Server को हर बार DB check नहीं करना पड़ेगा → load कम हो जाएगा।

👉 इस तरह **load balancing + stateless web server** possible हो जाता है।

---

8. Problem: International Users

- अगर सारे data servers सिर्फ **India** में हैं → तो USA/Europe से आने वाले users को website slow लगेगी।
- **Latency** और **load time** बढ़ जाएगा।

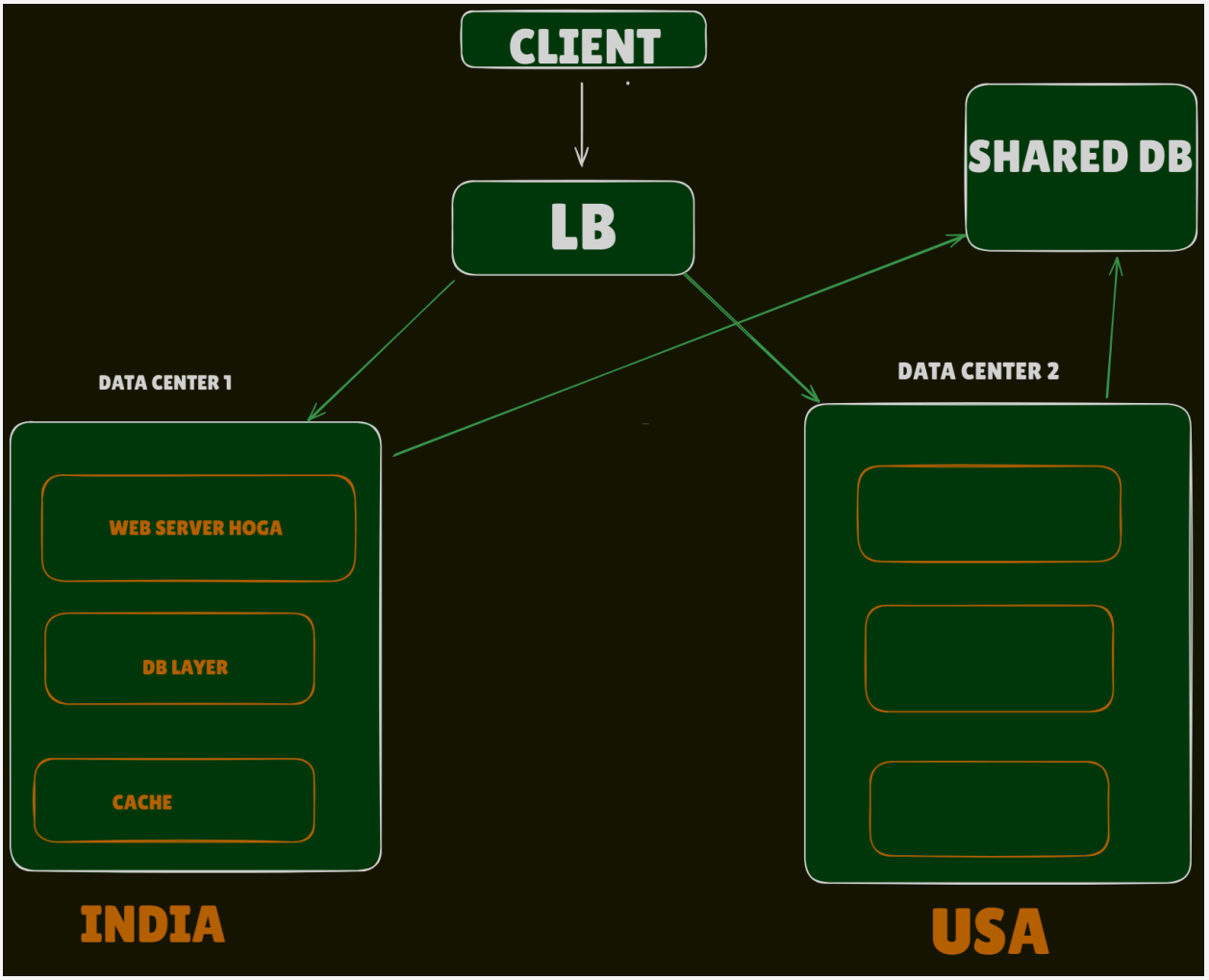
✅ Solution: Multiple Data Centers

- अलग-अलग **geographical locations** में data centers deploy करने पड़ते हैं।
- हर Data Center में:
    - अपने **Web Servers**
    - अपना **Database Layer**
    - अपना **Cache Layer**

📊 **Architecture Flow (Diagram के हिसाब से):**

```
Client → Load Balancer (LB)
LB → Data Center 1 (India)
   → Data Center 2 (International location)

Each Data Center:
   - Web Servers
   - DB Layer (Master-Slave replication)
   - Cache Layer
   - Messaging Queue
```



Key Points:

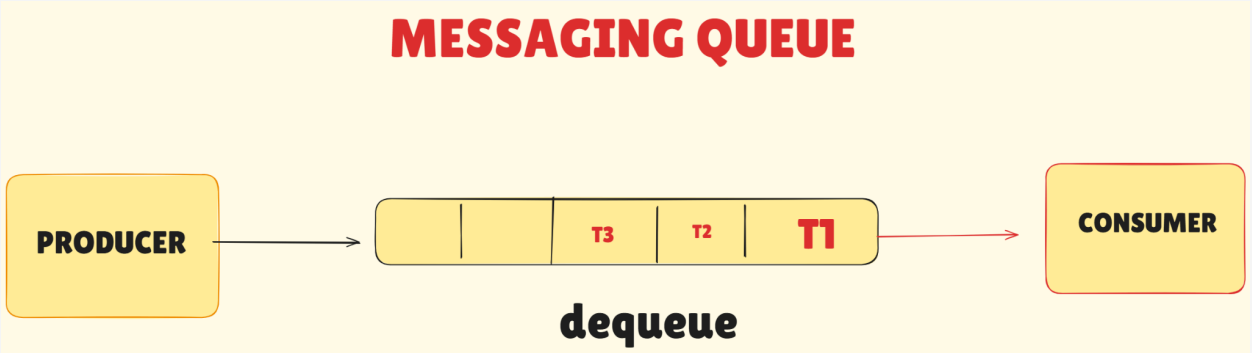
- Load Balancer user को उसके **nearest Data Center** पर redirect करेगा।
- DB layers आपस में **Master-Slave replication** के जरिए sync रहते हैं।
- Messaging Queue async communication handle करता है।

Messaging Queue (Pub-Sub Model)

Problem:

- हमारी application अब international users को handle कर पा रही है, लेकिन **asynchronous tasks** (background jobs जैसे notifications, email sending, video processing) को efficient तरीके से manage करना ज़रूरी है।

Solution: Messaging Queue (Pub-Sub Model)



Flow:

`Web Server (Producer) → Queue → Listener → Consumer`

- **Producer** → Task generate करता है और Queue में push करता है।
- **Queue** → Temporary storage where tasks wait.
- **Listener** → Queue को monitor करता है।
- **Consumer** → Queue से task लेता है और उसे process करता है।
- Task पूरा होने पर **Dequeue** हो जाता है।

👉 **Examples:**

- Kafka, RabbitMQ → Industry standard messaging queues।
- इन्हें बड़े companies async communication के लिए use करती हैं।

📌 **Key Advantage:**

- Application **asynchronous** काम कर सकती है।
- Web server free रहता है और heavy tasks background में होते हैं।

Database Sharding

Problem:

- जब users की संख्या बहुत बढ़ जाती है (billions), तो एक single database **bottleneck** बन जाता है।

- SQL queries slow हो जाती हैं।

Solution: Sharding

- DB को छोटे-छोटे हिस्सों (shards) में **partition** कर दिया जाता है।
- हर shard में total data का एक हिस्सा होता है।

Example:

```
Users Table (id, username)

Users distributed → DB0, DB1, DB2 (based on Sharding Key e.g., mod(id,3))
```

- aditya, alok → DB1
- abhay, alisa → DB2
- harsh → DB0

📌 **Benefit:**

- Load distribute हो जाता है।
- ज्यादा users को easily handle कर सकते हैं।

📌 **Problem:**

- Web server को पता नहीं होता किस DB से data fetch करना है → complexity बढ़ती है।
- Migration problem आती है।
- SQL Joins करना मुश्किल हो जाता है क्योंकि data अलग-अलग shards में split है।

Data De-Normalization (Intro)

Why needed?

- Sharding के बाद **JOIN operations** बहुत slow और complex हो जाते हैं।
- Example: User info DB0 में है और Orders DB2 में हैं → JOIN करना महंगा पड़ेगा।

Solution: De-Normalization

- **Same data** को **multiple places** पर रखना (duplicate copies बनाना)।
- Data redundancy intentionally बढ़ाई जाती है ताकि queries fast हों।

👉 Example:

- Normalized DB → `User` table अलग, `Orders` table अलग।
- De-Normalized DB → Orders table में सीधे user का नाम और email भी store कर देंगे।

📌 **Tradeoff:**

- **Read Performance** ↑ (Fast reads)
- **Write Complexity** ↑ (क्योंकि एक data multiple जगह update करना पड़ता है)।

⚡ Short Summary – Scaling from 0 → Million Users

1. **Single Server** → App + DB एक साथ → जल्दी limit hit कर जाएगा।
2. **DB** अलग करो → SQL vs NoSQL selection important है।
3. **Web Servers scale** → Load Balancer यूज़ करके SPOF हटाओ।
4. **DB Scaling** → Master-Slave (Write → Master, Read → Slave)।
5. **Cache (Redis/Memcached)** → Fast reads, DB load कम।
6. **CDN** → Static content (HTML, CSS, JS, Images) globally fast serve।
7. **Stateless Architecture** → Sessions DB/cache में, scaling आसान।
8. **Multiple Data Centers** → Global users के लिए latency कम।
9. **Messaging Queue (Kafka, RabbitMQ)** → Async tasks efficiently।
10. **Sharding** → DB को parts में तोड़ना, load distribute लेकिन JOINs मुश्किल।
11. **De-Normalization** → Duplicate data रखकर reads fast, writes complex।

👉 Simple flow:
Single Server → DB Separation → Load Balancer → Cache → CDN → Stateless → Multi-DC → MQ → Sharding → De-Normalization