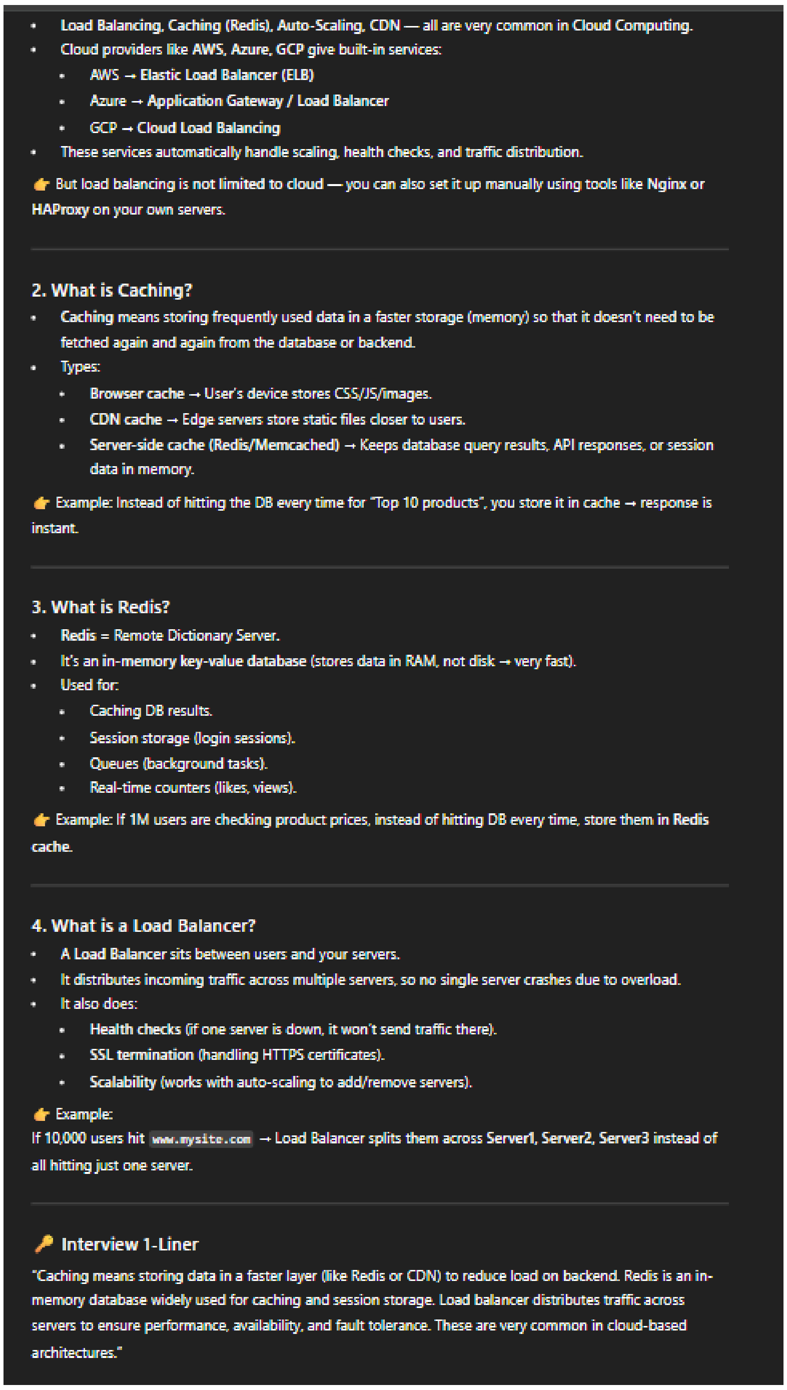


- **Load Balancing, Caching (Redis), Auto-Scaling, CDN** — all are very common in **Cloud Computing**.
- Cloud providers like **AWS, Azure, GCP** give built-in services:
  - AWS → **Elastic Load Balancer (ELB)**
  - Azure → **Application Gateway / Load Balancer**
  - GCP → **Cloud Load Balancing**
- These services automatically handle scaling, health checks, and traffic distribution.

👉 But load balancing is not limited to cloud — you can also set it up manually using tools like **Nginx** or **HAProxy** on your own servers.

---

2. What is Caching?

- **Caching** means storing frequently used data in a faster storage (memory) so that it doesn't need to be fetched again and again from the database or backend.
- Types:
  - **Browser cache** → User's device stores CSS/JS/images.
  - **CDN cache** → Edge servers store static files closer to users.
  - **Server-side cache (Redis/Memcached)** → Keeps database query results, API responses, or session data in memory.

👉 Example: Instead of hitting the DB every time for "Top 10 products", you store it in cache → response is instant.

---

3. What is Redis?

- **Redis** = Remote Dictionary Server.
- It's an **in-memory key-value database** (stores data in RAM, not disk → very fast).
- Used for:
  - Caching DB results.
  - Session storage (login sessions).
  - Queues (background tasks).
  - Real-time counters (likes, views).

👉 Example: If 1M users are checking product prices, instead of hitting DB every time, store them in **Redis cache**.

---

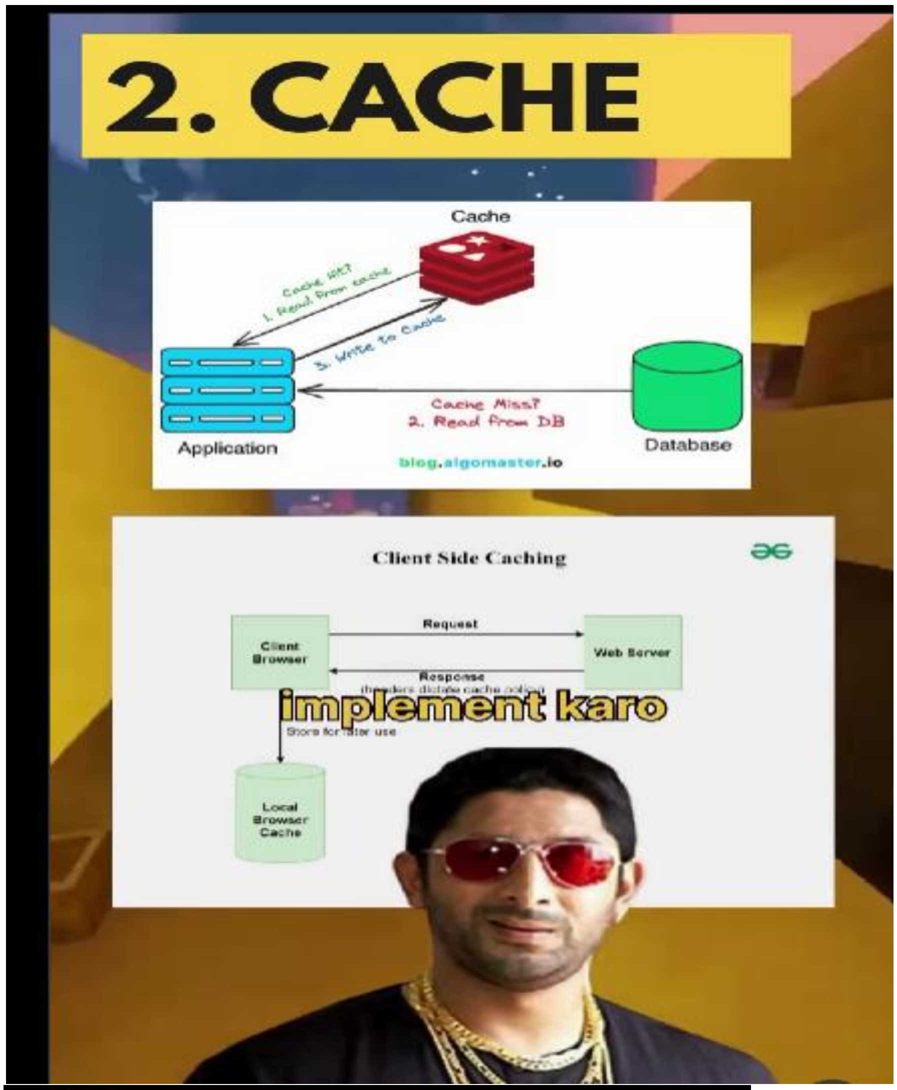
4. What is a Load Balancer?

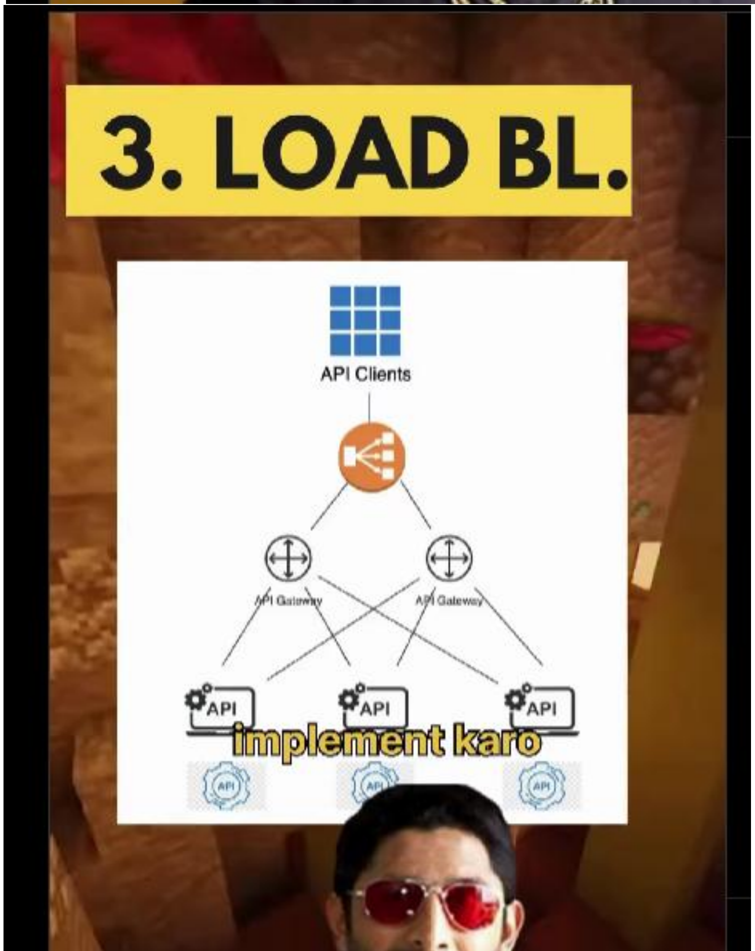
- A **Load Balancer** sits between users and your servers.
- It distributes incoming traffic across multiple servers, so no single server crashes due to overload.
- It also does:
  - **Health checks** (if one server is down, it won't send traffic there).
  - **SSL termination** (handling HTTPS certificates).
  - **Scalability** (works with auto-scaling to add/remove servers).

👉 Example:
If 10,000 users hit `www.mysite.com` → Load Balancer splits them across **Server1, Server2, Server3** instead of all hitting just one server.

🔑 Interview 1-Liner

"Caching means storing data in a faster layer (like Redis or CDN) to reduce load on backend. Redis is an in-memory database widely used for caching and session storage. Load balancer distributes traffic across servers to ensure performance, availability, and fault tolerance. These are very common in cloud-based architectures."


2. CACHE
Cache
Cache Hit?
1. Read from cache
3. Write to Cache
Cache Miss?
2. Read from DB
Application
Database
blog.algomaster.io
Client Side Caching
Client Browser
Request
Web Server
Response
(headers dictate cache policy)
implement karo
Store for later use
Local Browser Cache


3. LOAD BL.
API Clients
API Gateway
API Gateway
API
API
API
implement karo

Here's your content with corrected spelling and grammar:

If your API has slow response, what will you do? I will use Database Optimization Techniques (indexing, sharding, caching), load balancing, pagination, asynchronous operations.

✅ **1. Database Optimization Techniques**

👉 **Definition:** Database optimization techniques are methods we use to make queries faster, reduce server load, and improve performance. It will reduce the response time, provide high availability, scalability, performance insights, and management is easy. We can process requests efficiently so response time is less.

👉 **Why we use them:** • To reduce response time. • To handle more requests without crashing. • To reduce cost (optimize before scaling hardware).

👉 **Techniques:** • **Indexing** → Create indexes on frequently used columns for faster lookups. • **Query Optimization** → Avoid SELECT *, use joins wisely, apply proper filters. • **Connection Pooling** → Reuse DB connections instead of creating new ones for every request. • **Caching (Redis, Memcached)** → Store frequently accessed results in memory. • **Replication** → Multiple copies of DB (1 master for write, many slaves for read). • **Partitioning / Sharding** → Split large data into smaller parts across servers.

✅ **2. Load Balancing**

👉 **Definition:** Load balancing is the process of distributing incoming traffic across multiple servers so that no single server gets overloaded.

👉 **Why we use it:** • To improve performance. • To avoid downtime if one server fails. • Response time is high, so we use ELB like AWS ELB to distribute traffic among servers. • To scale horizontally by adding servers.

👉 **Example:** 10,000 users hit www.shop.com. Instead of all requests going to one server, a Load Balancer (LB) distributes them: • 3,000 → Server A • 3,000 → Server B • 4,000 → Server C

✅ **3. Scaling**

👉 **Definition:** Scaling means increasing the capacity of your system or server so it can handle more load. We do scaling for load balancing.

👉 **Types of Scaling:**

Vertical Scaling (Scale-Up): • Increase resources (CPU, RAM, storage) of a single server. • Advantage: Simple to implement. • Disadvantage: Expensive, has limits (cannot scale infinitely).

Horizontal Scaling (Scale-Out): • Add more servers and distribute traffic (via Load Balancer). • Advantage: Cost-effective, can scale infinitely. • Disadvantage: More complex setup.

✅ **4. Partitioning**

👉 **Definition:** Partitioning means splitting one big database table into multiple smaller parts to improve performance.

👉 **Why we use it:** • To reduce query size and improve response time. • Parallelization will be achieved and easy to manage. • To make data easier to manage.

👉 **Types of Partitioning:**

Vertical Partitioning: • Divide table by columns. • Example: Store user_profile data in one table and user_login data in another. • Advantage: Reduces size of each table. • Disadvantage: If query needs both sets of data → requires joins → slower.

Horizontal Partitioning: • Divide table by rows. • Example:

- IDs 1–1000 → DB1
- IDs 1001–2000 → DB2 • Advantage: Query load spread across servers. • Disadvantage: More complex to route queries.

👉 **Advantages of Partitioning:** • Faster queries. • Easier management of large data. • Can scale horizontally.

👉 **Disadvantages of Partitioning:** • Complexity in managing queries. • If partitioning is not done properly, some partitions may get overloaded.

✅ 5. Sharding

👉 **Definition:** Sharding means splitting a large database into smaller independent parts (shards) and storing them across multiple servers. Each shard holds only a portion of the total data.

👉 **Why we use it:** If millions of users are stored in a single DB, they are unmanageable, and if multiple requests come to the same database, response time is less and the system may crash. It is difficult to handle all of them. It will slow down the API. We can use sharding. Sharding reduces load by dividing data + traffic across multiple servers and shards.

👉 **How Sharding Works:** • Data is split by user_id, region, or hash key (partition key) into multiple shards. Here we divide the table horizontally and we call them shards. • We assign the shard to multiple servers and do scaling of servers as well to improve response time. Each server is related to the shard and handles specific data. • A routing layer (middleware/gateway) decides which shard to send the request to. Here a router is present which decides which request is sent to which server to handle it and the shard. • Also, API Gateway can be implemented.

Example: • Users 1–1,000,000 → Shard 1 (Asia) • Users 1,000,001–2,000,000 → Shard 2 (Europe) • Users 2,000,001–3,000,000 → Shard 3 (US)

👉 **Advantages of Sharding:** • High availability (if one shard fails, others work). • Improved response time (smaller DBs = faster queries). • Efficient load handling (requests spread across shards). • Easy to scale (just add another shard).

👉 **Disadvantages of Sharding:** • Complex to implement (needs routing logic). • Rebalancing data when adding new shards is difficult. • Some queries (like joins across shards) become complex.

✅ 6. Example: Sharding in My E-Commerce Project

Problem: • I had an e-commerce project with millions of users and orders. • A single database could not handle the load. • Response time was slow. • Risk of server crash if traffic increased.

Solution (Sharding): • I divided the data based on region (user_id range).

- Shard 1 → Asia users (IDs 1–1,000,000)

- Shard 2 → Europe users (IDs 1,000,001–2,000,000)
- Shard 3 → America users (IDs 2,000,001–3,000,000) • I implemented a routing layer in the backend:
- When a user logs in, system checks user_id/region.
- Request is routed to the correct shard.

Benefits: • Faster response time (queries hit smaller DBs). • High availability (if Shard 2 is down, Shard 1 & 3 still work). • Scalable (new shard can be added for Africa in future).