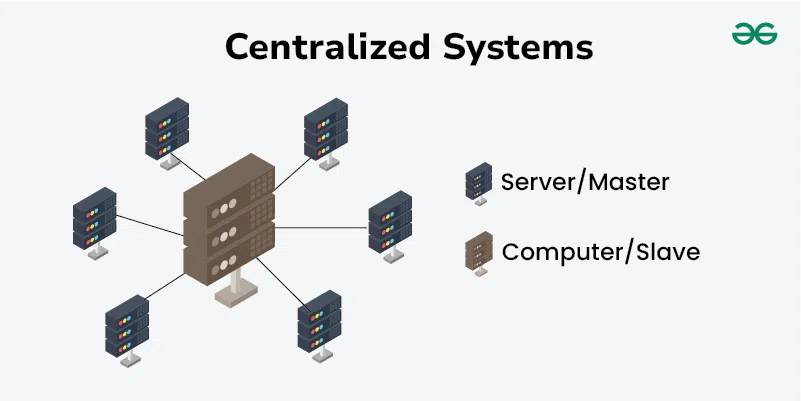
# TECHNICAL TERMS

1. Single Point of Failure (SPOF)
2. Single Point of Control
3. Resiliency
4. Scalability
5. Fault Tolerance
   * Fault tolerance is the ability of a system to continue operating properly in the event of a failure of one or more of its components. The goal is to ensure **high availability**, **reliability**, and **graceful degradation** rather than a complete system crash.
6. Centralized System
   * A **centralized system** is a system where a single central entity (server, database, or authority) controls all operations, decision-making, and data storage.
   * All users or nodes depend on this central entity for access and functionality.
   * **Characteristics of a Centralized System**
     + 1. Single Point of Control
          1. One central node manages the entire system.
          2. Users must communicate with the central server for processing.
       2. Single Point of Failure (SPOF)
          1. If the central server fails, the entire system stops working.
       3. Easy to Manage & Secure
          1. Since all data and processing are centralized, security policies are easier to enforce.
       4. Limited Scalability
          1. Adding more users increases load on the central server, leading to performance issues
       5. Faster Decision-Making
          1. Since all processing happens in one place, decisions are made quickly.
   * Real-Time Examples of Centralized Systems
     + 1. Traditional Banking System (SBI, ICICI, HDFC)
          1. All customer transactions go through a **central banking server**.
          2. If the bank’s **main server goes down**, no one can withdraw money or transfer funds.
          3. **Single point of failure**: If the central database is hacked, all user data is at risk.
       2. Railway Reservation System (IRCTC, Amtrak, Eurostar)
          1. All train ticket bookings, cancellations, and seat availability are managed by a **centralized server**
          2. If the IRCTC server is down, no one can book tickets.
       3. Government ID Systems (Aadhaar, Passport)
          1. All citizen data is stored in a **central government database**.
          2. If the server goes down, no one can access or verify their identity.
   * Advantages of Centralized Systems
     + 1. **Easy to Maintain & Secure** – Since all data is in one place, security and updates are simpler.
       2. **Fast Decision-Making** – Centralized processing speeds up operations.
       3. **Lower Cost** – Requires fewer resources compared to decentralized or distributed systems.
   * Disadvantages of Centralized Systems
     + 1. **Single Point of Failure** – If the central server fails, the entire system stops working.
       2. **Scalability Issues** – As demand grows, the system slows down.
       3. **Risk of Cyber Attacks** – If hackers breach the central system, they gain full access.
   * Summary
     1. **Centralized systems** are widely used but come with risks like **single points of failure** and **scalability issues**.
     2. **Modern systems** are shifting towards **decentralized** and **distributed** architectures for better **fault tolerance** and **scalability**.



1. Decentralized System
   * A **decentralized system** is a system where multiple nodes (servers, entities, or devices) , often spread across different locations , operate independently but still communicate with each other.
   * Unlike centralized systems, **there is no single controlling authority**; instead, decision-making is distributed across multiple nodes.
   * Instead of relying on **one central server**, multiple nodes manage data and processing, reducing the risk of failure and improving resilience.
   * Characteristics of a Decentralized System
     + 1. No Single Point of Failure (SPOF)
          1. If one node fails, the system continues to function.
       2. Multiple Authority Nodes
          1. Instead of a single master, multiple nodes handle decisions and processing.
       3. Better Security
          1. No central database means lower risk of hacking or data breaches.
       4. Scalability
          1. More nodes can be added to handle higher loads without major redesigns
       5. Latency Reduction
          1. Nodes operate independently, reducing dependency on a single data centre.
       6. Redundancy & Fault Tolerance
          1. Data is often replicated across multiple nodes, ensuring availability even if some nodes fail
   * Real-Time Examples of Decentralized Systems
     + 1. Blockchain (Bitcoin, Ethereum, Solana)
          1. **How it works?** Transactions are verified by multiple nodes instead of a central bank.
          2. **Why decentralized?** No single authority (like a bank) controls the network.
          3. **Example:** If one Bitcoin node fails, the network continues to function normally.
       2. Peer-to-Peer (P2P) Networks (BitTorrent, IPFS
          1. **How it works?** Files are shared across multiple computers instead of a central server.
          2. **Why decentralized?** Users (peers) directly exchange data without intermediaries
          3. **Example:** In BitTorrent, even if one peer goes offline, others continue sharing files.
   * Advantages of Decentralized Systems
     + 1. **Resilient & Fault-Tolerant** – If one node fails, others keep the system running.
       2. **More Privacy & Security** – No single entity controls all user data.
       3. **Scalable & Flexible** – Easy to add new nodes without central bottlenecks.
       4. **User Control & Transparency** – Users have more control over data and operations.
   * Disadvantages of Decentralized Systems
     + 1. **Complex to Manage** – Coordination between independent nodes can be difficult.
       2. **Higher Latency in Consensus-Based Systems** – Some decentralized systems (e.g., blockchain) take longer to reach consensus.
       3. **Security Risks in Open Networks** – If not designed well, attackers can manipulate some decentralized networks.

A diagram of several computer systems

AI-generated content may be incorrect.

1. Distributed System
   * A **distributed system** is a collection of multiple computers (or nodes) that work together to achieve a common goal.
   * These systems appear as a **single entity** to the end-user but are actually **spread across multiple locations**.
   * Unlike centralized systems, where everything is controlled by a single server, **distributed systems share resources, processing, and storage across multiple machines**.
   * Characteristics of a Distributed System
     + 1. Scalability
          1. Can handle more load by adding more nodes (computers).
          2. Example: Google adds more servers to process search queries faster.
       2. Fault Tolerance
          1. Even if one server fails, the system keeps working.
          2. Example: In Netflix’s distributed system, if one data centre goes down, another takes over
       3. Concurrency
          1. Multiple requests can be processed simultaneously.
          2. Example: Facebook handles millions of users at once.
       4. Location Transparency
          1. Users don’t need to know where data is stored or processed.
          2. Example: When using Google Drive, you don’t know which server is handling your request.
       5. High Availability
          1. The system is designed to be operational **24/7**.
          2. Example: Amazon’s e-commerce platform works worldwide without downtime.
   * Real-Time Examples of Distributed Systems
     + 1. Netflix, YouTube, Amazon Prime
          1. Videos are **stored on multiple servers** so users in different locations get the fastest response.
          2. If a server in the USA is slow, users in Europe get content from a closer data center.
       2. Content Delivery Networks (CDNs) - Cloudflare, Akamai
          1. Websites like YouTube and Netflix use CDNs to store content **closer to users** for faster streaming.
       3. Google Drive, Dropbox, OneDrive
          1. Data is replicated across multiple data centers so files are never lost.
       4. Cloud Computing Platforms : AWS, Microsoft Azure, Google Cloud
          1. Distributed cloud infrastructure runs applications for businesses worldwide.
       5. Microservices Architectures
          1. Architectures where applications are built as a collection of loosely coupled services.
   * Advantages of Distributed Systems
     + 1. **High Performance** – Handles millions of requests per second.
       2. **Fault Tolerance** – If one server fails, others take over automatically.
       3. **Better Load Balancing** – Distributes workload efficiently.
       4. **Cost-Effective** – Uses multiple cheaper machines instead of a single expensive one.
       5. **Data Replication** – Ensures that no data is lost.
   * Disadvantages of Distributed Systems
     + 1. **Complexity** – Managing multiple servers requires advanced networking and software.
       2. **Network Dependency** – Requires a strong internet connection to function efficiently
       3. **Synchronization Issues** – Keeping all nodes in sync is challenging.

A diagram of a distributed system

AI-generated content may be incorrect.

1. CDN - Content Delivery Network

# CENTRALIZED VS DE-CENTRALIZED VS DISTRIBUTED SYSTEMS

A blue and white logo

AI-generated content may be incorrect.

| **Feature** | **Centralized System 🏢** | **Decentralized System 🌍** | **Distributed System 🌎** |
| --- | --- | --- | --- |
| **Definition** | A single central server or authority controls everything. | Multiple independent entities manage different parts of the system. | A network of connected nodes works together to process tasks. |
| **Control** | Fully controlled by a **single entity** (company, government, etc.). | Control is **spread across multiple independent nodes**. | Nodes share control but **work together as one system**. |
| **Data Storage** | All data is stored in **one central server or database**. | Data is **replicated or split** among multiple independent nodes. | Data is **distributed across multiple locations** for redundancy. |
| **Failure Impact** | **High** – If the central server fails, the whole system goes down. | **Moderate** – Failure of one node doesn’t stop the entire system but may affect performance. | **Low** – If one or more nodes fail, the system continues functioning. |
| **Performance** | **Fast**, but bottlenecks occur due to overloading the central server. | **Varies** – Can be slow due to distributed decision-making. | **High** – Load is distributed, preventing slowdowns. |
| **Security Risks** | **High risk** – A single security breach can compromise the whole system. | **Moderate risk** – No single point of failure, but independent nodes may be vulnerable. | **Lower risk** – Redundancy and data replication provide better security. |
| **Scalability** | **Limited** – Expansion is difficult and requires upgrading the central system. | **Moderate** – Can scale by adding more independent nodes, but efficiency depends on coordination. | **Highly Scalable** – More nodes can be added without affecting performance. |
| **Data Consistency** | **High** – Since data is stored centrally, it is always consistent. | **Lower** – Different nodes may have **inconsistent data** due to lack of synchronization. | **High** – Distributed databases ensure data consistency. |
| **Maintenance & Cost** | **Expensive** – Requires **high infrastructure and IT support**. | **Moderate** – Lower initial costs but requires **complex governance**. | **Cost-Efficient** – Uses multiple standard servers instead of one expensive system. |
| **Example Systems** | **Traditional Banking (Central Banks, Visa, PayPal), Facebook, Google Search**. | **Blockchain (Bitcoin, Ethereum), Peer-to-Peer (P2P) Networks, Decentralized Social Media (Mastodon)**. | **Google Cloud, Netflix CDN, AWS, YouTube, Multiplayer Gaming Servers**. |
| **Best Use Cases** | **Small to Medium-Scale Applications** where centralized control is beneficial. | **Trust less Systems** like cryptocurrency, where users need control over their own data. | **Large-Scale Systems** requiring high performance, fault tolerance, and scalability. |

### ****"Are Distributed Systems Deployed Within a Single Region or Across Multiple Regions?"****

A **distributed system** refers to a collection of independent computers (nodes) that work together to appear as a single system to the end user. The nodes in a distributed system can be located in:

#### ✅ ****Same Region (Intra-region or Local Distribution)****

* All nodes are located within the **same data center** or **same geographic region** (e.g., AWS Mumbai Region).
* **Use case:** Low-latency internal communication, local fault tolerance, cost efficiency.
* **Example:**
  + A retail backend running across 3 availability zones (AZs) in the **Mumbai region** to ensure high availability.
  + Nodes handle different parts of the workload (like inventory, orders, payments).

## 📌 Benefits**:**

* Low latency between nodes
* Easier to manage and sync data
* Good for **high-performance internal systems**

📌Limitations**:**

* Region-wide outage (e.g., natural disaster) can take down the entire system.

#### 🌍 ****Multiple Regions (Inter-region or Global Distribution)****

* Nodes are deployed in **different geographical regions**, often **across continents**.
* **Use case:** Global scalability, geo-redundancy, disaster recovery, reduced latency for global users.
* **Example:**
  + One node cluster in **US-East (Virginia)**, another in **Asia-Pacific (Singapore)**, and another in **Europe (Frankfurt)**.
  + Users are routed to the **nearest region** (via CDN or DNS routing), and data is replicated globally.

📌 Benefits**:**

* Disaster resilience
* Serve customers closer to their location (low latency)
* Enables **geo-failover**

📌 **Challenges:**

* **Data consistency**: Requires eventual consistency or complex sync mechanisms
* Higher cost
* Increased latency in **cross-region replication**

### Network Partition vs Disaster

| ****Network Partition**** | ****Disaster**** |
| --- | --- |
| Services **can’t communicate** due to network issues | Entire **region/data center goes down** |
| System is **partially available** | System is **completely unavailable** |
| Risk of **data inconsistency** (e.g., orders not synced with inventory) | Risk of **total service outage** |
| Example: East US (orders) and West Europe (inventory) can’t sync | Example: Central India region outage — entire retail site down |
| Azure fixes: Retry logic, Eventual sync, Cosmos DB with multi-region | Azure fixes: Geo-redundant storage, Azure Site Recovery, Traffic Manager |

🧠 Summary**:**

* **Network Partition** = Split brain 🧠
* **Disaster** = Dead zone 💀

Use **zones for high availability** and **multi-region for disaster recovery** in Azure.

### CAP THEOREM

CAP Theorem (Consistency, Availability, Partition Tolerance)

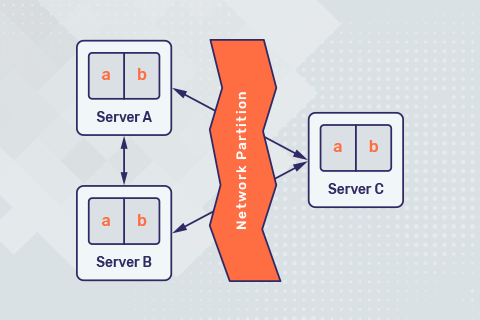
**Definition:**  
The CAP theorem, proposed by Eric Brewer, states that in a **distributed system**, you can achieve only **two** out of the following **three** guarantees at any time:

* **Consistency (C)** – Every read request returns the most recent write or an error (no stale data).
* **Availability (A)** – Every request gets a response (no system downtime), but it might return stale data(outdated, inaccurate, or irrelevant information).
* **Partition Tolerance (P)** – The system continues to function despite network partitions (i.e., some nodes cannot communicate).

Since network failures are **unavoidable**, distributed systems must choose between **Consistency and Availability**. When a network partition occurs, a distributed system must prioritize between consistency and availability. Depending on the use case, a system might favour availability to ensure continuous operation, or consistency to maintain data integrity. **CA databases don’t exist** in distributed environments because partition tolerance is mandatory.

### Network Partition in Distributed Systems

A **network partition** in a distributed system occurs when the network splits into two or more isolated groups due to communication failures. This means that some nodes cannot communicate with others, leading to potential **data inconsistency, availability issues, or even system failures**.



#### Causes of Network Partition in Distributed Systems

1. **Hardware Failures**

* **Router/Switch Failure** – If a network switch or router crashes, some nodes may lose connectivity.
* **Cable Disconnection** – Physical damage to fibre optics or Ethernet cables can break communication links.
* **Power Failures** – If a data centre or a region experiences a power outage, some nodes may become unreachable

1. **Software Issues**

* **Firewall Misconfigurations** – Security rules may unintentionally block traffic between nodes.
* **Bugs in Network Protocols** – A faulty TCP/IP stack or incorrect routing can lead to communication failures.
* **DNS Failures** – If nodes rely on DNS for discovery and it goes down, they may fail to locate each other.

1. **Network Congestion & Latency Issues**
   * **High Traffic Load** – Too many requests can overload network links, leading to dropped packets.
   * **Packet Loss & Retransmission Delays** – If packets are frequently lost or delayed, some nodes might become unreachable.
2. **Geo-Distributed Systems & Cloud Outages**
   * **Data Centre Failures** – If a cloud region goes offline (e.g., AWS region outage), nodes in that region become isolated.
   * **Inter-Region Latency** – Long-distance communication between cloud regions may cause temporary partitions.
3. **Security Attacks**
   * **DDoS (Distributed Denial of Service) Attacks** – Malicious actors can overload network infrastructure, disrupting communication.
4. **Software Updates & Version Mismatches**
   * **Incompatible Protocol Versions** – If different nodes run incompatible versions of a communication protocol, they may fail to talk.
   * **Service Restarts & Rolling Updates** – If updates cause inconsistent service availability, temporary partitions may form.
5. **Natural Disasters & External Factors**
   * **Earthquakes, Floods, or Fires** – Damage to data centres or network infrastructure can lead to partitions.
   * **Undersea Cable Cuts** – If a submarine cable is damaged, entire regions may become disconnected.
6. **Network Configuration Changes**
   * **Improper Load Balancer Settings** – Load balancers might unintentionally route traffic away from healthy nodes.
   * **Faulty Network Updates** – Pushing incorrect network rules or firewall settings may cut off connectivity.

#### Impact of Network Partition on Distributed Systems

When a network partition occurs, it can significantly affect the reliability, consistency, and availability of a distributed system. The impact depends on how the system is designed to handle partitions (CAP theorem: Consistency, Availability, Partition Tolerance).

1. **Data Inconsistency**
   * If different partitions continue operating independently, they may make conflicting updates.

**Example**: In a banking system, one partition processes a withdrawal while another does not see the updated balance, leading to incorrect account states.

1. **Reduced System Availability**
   * Some services may become unreachable if a partition cuts off critical nodes.

**Example**: In a microservices-based system, if the authentication service is isolated, users may be unable to log in.

1. **Increased Latency**
   * If a system attempts to reroute traffic through backup nodes or retry failed requests, response times may increase.

**Example**: In a cloud-based system, if a partition causes traffic to be redirected to another region, latency increases.

1. **Potential Data Loss or Overwrites**
   * If a partitioned node temporarily stores updates and later rejoins, newer data from another partition might overwrite its changes.

**Example**: In an eventually consistent database, updates made in an isolated partition may be lost when synchronization occurs.

1. **Failure in Consensus Algorithms**
   * Consensus algorithms are designed to enable a collection of distributed machines to work together as a coherent group, even in the presence of failures and outages**.**
   * Distributed consensus protocols (e.g., Paxos, Raft) may stall if they cannot reach a quorum.

**Example**: A Raft-based system that loses a majority of nodes in a partition will be unable to make progress.

1. **Cascading Failures** 
   * When a partition prevents one service from functioning, dependent services may also fail.

**Example**: A network partition in an e-commerce platform's payment service could prevent order processing, leading to lost transactions.

1. **Split-Brain Syndrome**
   * When multiple partitions assume they are the primary leader, leading to conflicting decisions.

**Example**: In a leader-based system (e.g., Apache ZooKeeper, Kafka), two partitions may elect separate leaders, causing inconsistencies.

1. **Delayed or Lost Updates**
   * Writes performed in one partition may not sync to the other until the partition heals.

**Example**: In a NoSQL database like Cassandra, writes may be stored in one partition and only later reconciled when the network recovers.

#### How Network Partitions Relate to the CAP Theorem?

The CAP theorem states that a **distributed system** can provide only two out of the following three guarantees simultaneously

* **Consistency:** All nodes see the same data at the same time.
* **Availability:**Every request receives a response, without guarantee that it contains the most recent data.
* **Partition Tolerance:**The system continues to operate despite network partitions.

When a network partition occurs, a distributed system must prioritize between consistency and availability. Depending on the use case, a system might favour availability to ensure continuous operation, or consistency to maintain data integrity.

#### ****Network Partition Detection in Distributed Systems****

Detecting network partitions is **crucial** to ensuring that distributed systems handle failures effectively. Since partitions can cause **data inconsistencies, availability issues, or split-brain scenarios**, systems must detect them early and take appropriate action.

1. **Network Monitoring & Logging**

Systems use **network monitoring tools** to detect sudden drops in connectivity.

**How It Works:**

* Use monitoring tools like **Prometheus, Grafana, Nagios, and Splunk** to track network behaviour.
* Set up **alerts** for sudden node disconnections or unusual traffic patterns.

**Examples:**

* **Netflix’s Chaos Monkey** tests partition handling by simulating failures.
* **AWS CloudWatch** monitors network health in cloud environments.

**Pros:**

* Helps debug network issues in real-time.

**Cons**:

* Detection is passive; needs manual intervention to react.

1. **Heartbeat Mechanism (Failure Detection)**

Nodes in a distributed system periodically send **heartbeat messages** to confirm their availability.

**How It Works:**

* Each node sends a **heartbeat** to its peers or a central monitor (e.g., ZooKeeper, etcd).
* If a node **does not respond within a timeout**, it is considered **suspect** or failed.
* If multiple nodes in a region are unreachable, a **network partition is inferred**.

**Examples:**

* Apache ZooKeeper uses **ZAB (ZooKeeper Atomic Broadcast) protocol** for leader election with heartbeats.
* Apache Cassandra and Kubernetes use **Gossip Protocols** for failure detection.

**Pros:**

* Lightweight and fast detection.

**Cons**

* False positives due to temporary network delays.

1. **Gossip Protocols (Decentralized Detection)**

A decentralized approach where nodes share information about other nodes' health.

**How It Works:**

* Each node **randomly selects peers** and exchanges status updates.
* If a node is marked **unreachable by multiple peers**, a partition is suspected.
* Nodes eventually **converge on a common view** of the partition.

**Examples:**

* **Apache Cassandra, Amazon DynamoDB** use gossip-based failure detection.
* **Consul** (by HashiCorp) uses gossip to detect failures in service discovery.

**Pros:**

* Scales well for large systems

**Cons:**

* Slower convergence in detecting partitions.

1. **Consensus Protocols (Leader-Based Detection)**

Consensus algorithms like **Raft, Paxos, and ZooKeeper's ZAB** detect partitions when leader nodes lose quorum.

**How It Works:**

* A leader regularly checks if it can communicate with the majority of nodes (**quorum**).
* If it loses quorum, it **steps down** to prevent split-brain.
* Other nodes attempt to **elect a new leader**.

**Examples:**

* **Raft-based systems (e.g., etcd, Consul)** detect partitions via leader timeouts.
* **Apache Kafka** detects partitioned brokers via ZooKeeper.

**Pros:**

* Prevents data inconsistencies (split-brain).

**Cons:**

* Can cause temporary unavailability during re-elections.

1. **Timeout-Based Detection**

Nodes mark others as unreachable if they fail to respond within a given timeframe.

**How It Works:**

* Each request sent between nodes has a timeout.
* If the response is not received before the timeout, the node is marked as unreachable.
* If multiple nodes become unreachable simultaneously, a partition is assumed.

**Examples**:

* Raft and Paxos use timeouts to detect leader failures.
* Database systems like MongoDB detect node failures using response timeouts.

**Pros**: Simple to implement.

**Cons**: Can be affected by temporary slowdowns (false positives).

1. **Fencing Tokens & Lease Mechanisms**

Fencing prevents split-brain scenarios by ensuring that only one partition can modify data.

**How It Works:**

* A leader or primary node acquires a lease (time-limited token).
* If the leader loses contact with the majority, it cannot renew the lease.
* Another partition cannot act as the leader until the previous lease expires.

**Examples**:

* ZooKeeper uses fencing tokens to prevent multiple leaders.
* Google Spanner uses TrueTime to synchronize node status.

**Pros**:

* Strong protection against split-brain issues.

**Cons**:

* Adds extra coordination overhead.

#### ****Strategies for Handling Network Partitions in Distributed Systems****

Network partitions are inevitable in distributed systems, so designing for them is crucial. The strategy chosen depends on the system's requirements for **Consistency (C), Availability (A), and Partition Tolerance (P)** as per the **CAP theorem**. Below are key strategies for mitigating network partitions:

## ****Quorum-Based Approaches**** (Used in CP or AP Systems)

Quorum-based strategies ensure that a minimum number of nodes agree on operations to maintain consistency. In a distributed database, a write operation requires acknowledgment from a majority of replicas (quorum). If a network partition occurs, only the partition with the majority can process write operations, preventing data inconsistencies.

**Read & Write Quorums** – Require a certain number of nodes to acknowledge a read or write before committing.

**Example**: Apache Cassandra uses N = W + R (where N = total replicas, W = write quorum, R = read quorum).

**Advantages**: Prevents stale reads and data loss.  
**Disadvantages**: Reduces availability when quorum isn't met.

1. **Eventual Consistency** (Used in AP Systems)

Allow nodes to process operations independently during a partition and resolve inconsistencies later. Eventual consistency models allow temporary inconsistencies during network partitions, resolving them once the partitions heal. This approach is common in systems like Cassandra and DynamoDB, where the priority is to ensure availability and eventual reconciliation of data.

**Conflict Resolution via Vector Clocks** – Track causality of updates to merge conflicting writes.

**CRDTs (Conflict-Free Replicated Data Types)** – Use mathematical models to ensure conflict-free merging.

**Example**: Amazon DynamoDB and Riak use eventual consistency with conflict resolution.

**Advantages**: High availability, no service disruption.

**Disadvantages**: Temporary inconsistencies may occur.

1. **Leader Election & Failover Mechanisms**

Ensure that only one leader is active during partitions to avoid split-brain scenarios.

**Raft & Paxos Consensus Protocols** – Ensure a single leader per partition.

**Failover to a Backup Leader** – If a leader is isolated, elect a new one safely.

**Example**: Apache Kafka ensures a leader is always available for partitions.

**Advantages**: Prevents conflicting writes.

**Disadvantages**: May result in downtime during leader re-election.

1. **Partition Detection & Fencing**

Detect network partitions early and prevent conflicting operations.

**Heartbeats & Failure Detection** – Nodes periodically check each other's health.

**Lease Mechanisms** – Limit leader validity time to prevent multiple leaders

**Example**: Apache ZooKeeper uses fencing to prevent split-brain conditions.

**Advantages**: Avoids inconsistency due to multiple leaders.

**Disadvantages**: Requires extra coordination overhead.

1. **Multi-Region & Redundant Networks**

Design systems to work across multiple data centers to minimize partition effects.

**Geo-Distributed Replication** – Replicate data across different locations.

**Smart Load Balancing** – Redirect requests to healthy nodes.

**Example**: Google Spanner synchronizes distributed nodes with atomic clocks.

**Advantages**: Improves fault tolerance.

**Disadvantages**: Increases latency due to cross-region communication.

1. **Graceful Degradation & Read-Only Mode**

Allow systems to function in a limited mode instead of failing completely.

**Fallback to Read-Only Mode** – If write operations are unsafe, allow only reads.

**Degraded Service Modes** – Reduce non-critical features during partition.

**Example**: Banking systems may allow balance checks but disable transactions.

**Advantages**: Improves user experience during failures.

**Disadvantages**: Not always applicable for all services.

1. **Hybrid Approach: CAP Theorem Trade-Offs**

Choose a balance between Consistency (C), Availability (A), and Partition Tolerance (P) based on business needs.

**CP Systems** (e.g., Zookeeper, etcd) – Favor consistency, but may become unavailable.

**AP Systems** (e.g., DynamoDB, Cassandra) – Favor availability with eventual consistency.

**Example**: Banking applications favour CP, while social media platforms favour AP.

**Advantages**: Customizable trade-offs for different use cases.

**Disadvantages**: No perfect solution—must choose between consistency or availability.

## ****Key Takeaways****

✅ **Network Partitioning is Unavoidable**

* Hardware failures, software bugs, congestion, or cloud outages can cause partitions.
* Designing for **failure tolerance** ensures system resilience.

✅ **CAP Theorem Defines Trade-offs**

* You **cannot** have **Consistency (C), Availability (A), and Partition Tolerance (P) together**.
* Choose **CP** (e.g., financial systems) or **AP** (e.g., social media, e-commerce) based on needs.

✅ **Detection & Handling are Crucial**

* Use **heartbeats, timeouts, gossip protocols, and consensus algorithms** to detect partitions early.
* Implement **quorum-based writes, leader election, fencing tokens, and read replicas** to mitigate impact.

✅ **Split-Brain is a Serious Issue**

* If two partitions think they are the primary, data corruption can occur.
* Use **lease mechanisms, fencing tokens, and quorum checks** to prevent this.

✅ **Choose the Right Strategy**

* **Critical data systems** (e.g., banking, stock trading) prioritize **strong consistency (CP)**.
* **Scalable, highly available apps** (e.g., Netflix, Amazon) favor **eventual consistency (AP)**.
* **Hybrid approaches** (e.g., Google Spanner) use **synchronized clocks + quorum-based writes**.

## ****Final Thought: Design for Failure, Not Just Success****

## **A well-designed distributed system doesn’t just function when everything works—it remains stable even when failures happen.**

### Single-Leader Applications (Leader-Follower Architecture)

A **single-leader architecture** (also called **Leader-Follower** or **Primary-Replica**) is a **distributed system pattern** where one **Leader (Primary)** node manages updates, and multiple **Followers (Replicas)** replicate the data.

💡 **Key Idea:**

* **Leader (Primary):** Handles all **writes** and **synchronizes** data to followers.
* **Followers (Replicas):** Only handle **read** requests.

**🛠 Where is it Used?**

✅ **Databases (Replication)** → MySQL, PostgreSQL, MongoDB Replication  
✅ **Distributed Systems** → Apache Kafka, Raft Consensus  
✅ **Load Balancing & Failover Systems**

## ****📌 Single-Leader (Leader-Follower) vs. Master-Slave Architecture****

Both **Single-Leader (Leader-Follower)** and **Master-Slave** architectures have a similar concept where one node is responsible for coordination while others replicate data. However, they have **key differences** in terms of **control, failover, data synchronization, and modern usage.**

**Master-Slave is becoming obsolete** due to the **centralized failure risk** and **scalability issues**.  
✅ **Single-Leader (Leader-Follower) is still widely used** but evolving into **Multi-Leader and Leaderless models** for higher availability (e.g., DynamoDB, Cassandra).

 **Master-Slave** → **Used for task delegation & execution.**

 **Leader-Follower** → **Used for data replication & distributed consensus.**

 **Leader-Follower is more fault-tolerant & scalable**, whereas **Master-Slave has a single failure point.**

### Database Architecture Patterns

## ****Single-Leader (Leader-Follower) Architecture****

🔹 **Concept:** One **leader (primary)** handles writes, and **followers (replicas)** handle reads.  
🔹 **Use Cases:** High read scalability, database replication.  
🔹 **Examples:** MySQL Replication, PostgreSQL Replication, Redis Primary-Replica.  
✅ **Pros:** Scalable reads, strong consistency.  
❌ **Cons:** Write bottleneck, replication lag.

## ****Multi-Leader (Multi-Master) Architecture****

🔹 **Concept:** Multiple **leaders** handle both **reads and writes**, syncing data across instances.  
🔹 **Use Cases:** Geo-distributed databases, multi-datacenter setups.  
🔹 **Examples:** Amazon Aurora, MongoDB, Active-Active MySQL.  
✅ **Pros:** High availability, better write scalability.  
❌ **Cons:** Conflict resolution is complex.

## ****Leaderless (Peer-to-Peer) Architecture****

🔹 **Concept:** No single leader; all nodes can handle **reads & writes**.  
🔹 **Use Cases:** Distributed NoSQL databases, high availability.  
🔹 **Examples:** DynamoDB, Cassandra, Riak.  
✅ **Pros:** No single point of failure, highly available.  
❌ **Cons:** Eventual consistency, complex coordination.

## ****Sharded (Partitioned) Architecture****

🔹 **Concept:** Data is **split (sharded)** across multiple databases based on a key (e.g., user ID).  
🔹 **Use Cases:** Large-scale applications, horizontal scalability.  
🔹 **Examples:** MongoDB Sharding, MySQL Partitioning, Elasticsearch.  
✅ **Pros:** Infinite scalability, avoids single-node overload.  
❌ **Cons:** Cross-shard queries are complex.

## ****CQRS (Command Query Responsibility Segregation)****

🔹 **Concept:** **Writes (commands)** and **reads (queries)** use **separate databases**.  
🔹 **Use Cases:** High-performance systems, event sourcing.  
🔹 **Examples:** Event-Driven Microservices, Banking Systems.  
✅ **Pros:** Optimized for performance, scalable.  
❌ **Cons:** Data synchronization complexity.

### CAP Classification of Distributed Databases (Multi-Region Deployment)

In a **distributed system** spanning multiple regions, **network partitions (P) are inevitable** due to communication delays, failures, or latency across regions.

* This means that every distributed database must choose between **Consistency (C) and Availability (A)** while tolerating partitions (P).
* **CA databases don’t exist** in distributed environments because partition tolerance is mandatory.

#### How to Make Azure Cosmos DB AP or CP?

Azure Cosmos DB allows you to **tune the CAP behaviour** using its **Consistency Levels**.

* To make Cosmos DB **AP**, use **Eventual Consistency**.
* To make Cosmos DB **CP**, use **Strong Consistency**.

#### ****Steps to Make Cosmos DB AP (Availability + Partition Tolerance)****

* **Choose an "Eventual" or "Session" Consistency Mode**
  + **Eventual Consistency**: Maximizes availability; reads might not reflect the latest writes.
  + **Session Consistency**: Ensures per-session consistency but allows eventual consistency across sessions.
* **Enable Multi-Region Writes**
  + This ensures that writes can happen in any region without waiting for global synchronization.
* **Replication Strategy**:
  + Set up **multi-region replicas** to handle partition failures.

#### ****Steps to Make Cosmos DB CP (Consistency + Partition Tolerance)****

* **Choose "Strong Consistency"**
  + This ensures that every read returns the most recent write globally.
* **Disable Multi-Region Writes**
  + If strong consistency is required, **only one write region** should exist.
* **Replication Strategy**:
  + Use a single primary write region and **multiple read replicas**.

#### How Azure Cosmos DB Internally Handles AP (Availability + Partition Tolerance)

When configured for **AP mode**, Cosmos DB prioritizes **availability** and **partition tolerance**, meaning:  
✅ The system remains available even when network failures occur.  
✅ Reads and writes can happen in any region without waiting for synchronization.  
❌ Some inconsistency might be present across regions due to eventual consistency.

#### Internal Mechanisms That Enable AP Behaviour in Cosmos DB

##### ****Multi-Region Writes (Enables High Availability)****

* In **AP mode**, Cosmos DB allows **multi-region writes**, meaning **any replica can accept writes**.
* Writes are replicated **asynchronously** across regions to ensure availability.
* If a region goes down, another region can continue accepting writes **without downtime**.

**Example:**  
If a user in **India** writes data, another user in **USA** might see a slightly outdated version of that data (due to async replication).

##### ****Asynchronous Replication (Ensures Partition Tolerance)****

* Instead of waiting for confirmation from all replicas, **writes are propagated asynchronously**.
* This means that in case of network failure (partitioning), the database remains **available**, but the latest data may not be immediately visible everywhere.

**Example:**  
If a write is made in **Asia** and a read is requested in **Europe**, the read might return **slightly stale data** if replication is delayed.

##### ****Conflict Resolution (Handles Write Conflicts Across Regions)****

When multiple regions allow writes, **conflicts** can occur. Cosmos DB provides **three conflict resolution strategies**:

| **Strategy** | **How It Works** | **Use Case** |
| --- | --- | --- |
| **Last Writer Wins (Default for AP Mode)** | The most recent write (based on timestamp) is kept. | Best for cases where newer data is usually correct. |
| **Custom Resolution (User-Defined Functions - UDFs)** | Developers write logic to merge conflicting changes. | Needed for complex business logic. |
| **Manual Resolution** | Conflicts are logged, and apps handle resolution. | When human intervention is needed. |

**Example:**

* User 1 in **Japan** updates a profile name to "Alice".
* User 2 in **USA** updates the same profile name to "Alicia".
* Since both writes happen at the same time, Cosmos DB picks the **latest timestamp** as the final value.

##### ****Consistency Levels (Tuning AP Behavior)****

Cosmos DB provides **five consistency levels**, and the **weaker the consistency, the stronger the AP behaviour**. **For a pure AP system**, you should use **Eventual Consistency** (default) or **Session Consistency**.

| **Consistency Level** | **CAP Category** | **How It Affects AP** |
| --- | --- | --- |
| **Strong** | CP | Disables AP behaviour (waits for full sync). |
| **Bounded Staleness** | CP/AP | Ensures partial consistency but allows some lag. |
| **Session** | AP | Each user session is consistent but may differ across sessions. |
| **Consistent Prefix** | AP | Maintains write order but allows lag. |
| **Eventual** | AP | Maximizes availability with possible stale reads. |

**Example:**

If a user writes **"X = 5"** in **Asia**, another user in **USA** might still see **"X = 3"** until replication is complete.

##### Automatic Failover & Global Distribution (Ensures High Availability)

 Cosmos DB has **multi-region failover** so that if one region **fails**, another region automatically takes over.

 Read operations are served from the closest available region.

 Ensures high availability **even in case of entire region failure**.

**Example:**

If the **West US** region goes down, Cosmos DB automatically routes requests to **East US** **without downtime**.

#### ****Summary: How Cosmos DB Implements AP Behaviour Internally****

| **Feature** | **How It Supports AP?** |
| --- | --- |
| **Multi-Region Writes** | Allows writes from any region without blocking. |
| **Asynchronous Replication** | Ensures availability, tolerates network failures. |
| **Conflict Resolution** | Handles write conflicts without blocking availability. |
| **Eventual Consistency** | Maximizes AP by allowing stale reads. |
| **Automatic Failover** | Ensures uptime even during region failures. |

#### ****When to Use Cosmos DB in AP Mode?****

* **Global Applications:** Social media feeds, IoT, gaming leaderboards.
* **High-Availability Services:** Applications that **must stay online**, even if some data is slightly outdated.
* **Caching & Recommendation Engines:** Where slight inconsistency is acceptable.

#### How Azure Cosmos DB Internally Handles CP (Consistency + Partition Tolerance)

When configured for **CP mode**, Cosmos DB prioritizes **consistency** and **partition tolerance**, meaning:  
✅ Data is always consistent across all replicas.  
✅ Reads always return the latest committed write.  
✅ Network failures may **temporarily block writes** to maintain consistency.  
❌ Availability might be affected during partitions

#### Internal Mechanisms That Enable CP Behaviour in Cosmos DB

##### ****Single-Region Writes (Strong Consistency Requirement)****

* In **CP mode**, Cosmos DB enforces **single-region writes** to **avoid conflicts**.
* This means all writes must go through a **designated leader region** before propagating.
* **Synchronous replication** ensures data consistency before acknowledging a write.

**Example:**  
If a user in **India** writes data, another user in **USA** cannot read outdated data. The read must wait until replication completes.

##### ****Synchronous Replication (Ensures Consistency)****

* Writes are **only confirmed after being replicated** to all configured regions.
* Cosmos DB uses a **quorum-based approach** to ensure a majority of replicas acknowledge the write.

**Example:**

* A **banking system** where a balance update must be immediately reflected everywhere.
* If **User A (USA)** transfers money, **User B (India)** must see the updated balance instantly.

##### ****Strong Consistency Level (Enforcing CP)****

Cosmos DB provides **five consistency levels**, but **only "Strong Consistency" enforces CP**: **For a true CP system, you must configure Cosmos DB with "Strong Consistency".**

| **Consistency Level** | **CAP Category** | **Behaviour** |
| --- | --- | --- |
| **Strong (CP Mode)** | CP | Reads always return the latest committed write (global consistency). |
| **Bounded Staleness** | CP/AP | Reads lag behind by a defined time interval. |
| **Session** | AP | Each user session sees consistent data, but others may see different versions. |
| **Consistent Prefix** | AP | Write order is preserved, but data may be stale. |
| **Eventual** | AP | Maximizes availability but allows stale reads. |

**Example:**  
If a write happens in **Europe**, a read in **Japan** **must wait** until the change is **replicated and confirmed**.

##### ****Quorum-Based Replication (Ensuring Writes Are Committed)****

* Cosmos DB uses **quorum-based replication** in CP mode.
* A write is considered successful only when **a majority of replicas acknowledge it**.

**Example:**

* If a write is sent to **3 replicas**, it must be **acknowledged by at least 2 replicas** before being committed.
* If network failure prevents consensus, **writes may be blocked** to maintain consistency.

##### ****Failover Handling (Consistency Over Availability)****

* If the **leader region fails**, Cosmos DB **elects a new leader**.
* Until a new leader is elected, **writes are blocked** to prevent inconsistency.

**Example:**

* If **West US** is the leader and it goes down, a new leader is chosen in **East US**.
* Until failover completes, **writes are paused** to ensure strong consistency.

#### ****Summary: How Cosmos DB Implements CP Behaviour Internally****

| **Feature** | **How It Supports CP?** |
| --- | --- |
| **Single-Region Writes** | Ensures only one leader handles writes. |
| **Synchronous Replication** | Writes are confirmed only after reaching all replicas. |
| **Strong Consistency Level** | Guarantees that reads always return the latest data. |
| **Quorum-Based Replication** | Prevents conflicts by requiring majority agreement. |
| **Failover Mechanism** | Prevents writes until a new leader is established. |

#### ****When to Use Cosmos DB in CP Mode?****

* **Banking & Financial Systems:** Transactions **must be consistent** (no stale reads).
* **Inventory & Order Processing:** Prevents duplicate orders due to race conditions.
* **Healthcare & Critical Systems:** Medical records **must remain accurate** at all times.

#### ****Final Thought: AP vs. CP in Cosmos DB****

| **Mode** | **AP (Availability)** | **CP (Consistency)** |
| --- | --- | --- |
| **Replication** | **Asynchronous** | **Synchronous** |
| **Write Handling** | Multi-master | Single leader |
| **Read Consistency** | Eventual | Strong |
| **Partition Handling** | Stays available | May block writes |
| **Use Case** | Social media, IoT | Banking, Healthcare |

### What is ****Split Brain**** in Distributed Systems?

**Split Brain** is a condition in a **network partition** where **two or more parts of the system continue operating independently**, **believing they are the only active part**, which leads to **conflicts or data inconsistency**.

#### ****Why It Happens:****

In a network partition, if both sides of the system **lose communication**, but don't realize it, they keep making **conflicting updates**.

#### 🛒 ****Retail Example (Azure):****

You have a distributed retail app with:

* **Order service in East US**
* **Inventory service in West Europe**
* They use **Cosmos DB multi-region** with active-active writes

#### 🔁 ****Scenario:****

* Network partition occurs between East US and West Europe
* Both services still work **independently**
* East US sells 10 units of a product
* West Europe also sells the **same 10 units**, unaware of East US
* Result: **Overselling** and **inventory mismatch**

This is a **split brain** condition — both regions think they have the full picture.

#### ⚠️ ****Problems Caused:****

* **Data conflicts**
* **Lost updates**
* **Inconsistent state**
* **Revenue and trust loss** in retail

#### 🛡️ ****How to Handle Split Brain in Azure:****

| **Technique** | **Purpose** |
| --- | --- |
| **Quorum-based writes** (e.g., Cosmos DB consistency levels) | Ensure majority agreement before accepting writes |
| **Leader election** (e.g., Service Fabric, etcd) | Only one active writer at a time |
| **Manual conflict resolution policies** | Define rules for resolving conflicts (e.g., "latest write wins") |
| **Graceful degradation** | Temporarily restrict some operations in degraded mode |

### ACID Properties in Databases

ACID properties ensure the reliability and integrity of database transactions. ACID stands for **Atomicity, Consistency, Isolation, and Durability**

##### 1. ****Atomicity**** (All or Nothing)

1. Ensures that a transaction is either **fully completed** or **fully rolled back** if any part of it fails.
2. Example:
   * Consider a bank transfer where ₹5000 is transferred from Account A to Account B.
   * The transaction consists of two steps:
     1. Deduct ₹5000 from Account A.
     2. Add ₹5000 to Account B.
   * If step 2 fails after step 1, Atomicity ensures that the deduction from Account A is rolled back, maintaining data integrity.
3. In SQL, use **BEGIN TRANSACTION** to start, **COMMIT** to save changes, **ROLLBACK** to undo in case of failure
4. Spring Boot achieves atomicity using **@Transactional annotation**. If any part of the transaction fails, everything is rolled back.

##### 2. ****Consistency**** (Valid State)

* Ensures that the database transitions from **one valid state to another** without violating constraints.
* Example:
  + If a database has a constraint that a student's age cannot be negative, inserting a student record with age -5 will be rejected to maintain consistency.
* In SQL , Use **primary keys**, **foreign keys**, **unique constraints**, and **check constraints** to prevent invalid data
* In Spring Boot , Use JPA annotations to enforce constraints. @**Column**(nullable = false) → Prevents null values , @**Size**(min = 3, max = 50) → Restricts string length ,@**Min**(0) → Ensures non-negative value

##### 3. ****Isolation**** (Concurrent Transactions)

* Ensures that multiple transactions executing **simultaneously do not interfere** with each other.
* Example:
  + Suppose two users are booking the last available train ticket at the same time.
  + Isolation ensures that only **one user successfully books the ticket**, preventing double-booking.
* In SQL, Use **transaction isolation levels** to manage concurrent transactions

Different isolation levels in SQL:

**READ UNCOMMITTED**: Allows dirty reads (least safe)

**READ COMMITTED**: Prevents dirty reads but allows non-repeatable reads

**REPEATABLE READ**: Prevents dirty and non-repeatable reads but allows phantom reads

**SERIALIZABLE:** Ensures full isolation (most strict)

Example: Set the required isolation level using **SET TRANSACTION ISOLATION LEVEL REPEATABLE READ** before running a transaction

* In Spring boot , Use @**Transactional**(isolation = **Isolation.READ\_COMMITTED**)

##### ****4. Durability**** (Permanent Changes)

* Ensures that once a transaction is committed, it is **permanently stored**, even if the system crashes.
* Example:
  + After a user successfully places an online order, even if the system crashes, the order remains recorded in the database.
* In SQL , Enable Database Transaction Logs. Use **Write-Ahead Logging (WAL)** where changes are first written to a log before being applied to the database. Example: In PostgreSQL, enable WAL logging by setting **wal\_level = logical** in configuration
* Ensure Spring Boot Uses Transaction Logs. Configure **Hibernate flush mode** to **AUTO** (default) to commit changes properly

**spring.jpa.properties.hibernate.connection.provider\_disables\_autocommit = false**

#### ****When to Use ACID Properties?****

* **Banking Systems:** Ensures no money is lost in transactions.
* **E-commerce:** Prevents double purchases of the same item.
* **Stock Trading:** Ensures trade records remain valid.
* **Healthcare & Government Systems:** Maintains integrity and accuracy.

#### ****How to Use ACID Properties?****

* **Atomicity:** Use **transactions** (BEGIN TRANSACTION, COMMIT, ROLLBACK).
* **Consistency:** Define **constraints** (Primary Key, Foreign Key, CHECK).
* **Isolation:** Use **transaction isolation levels** (Read Uncommitted, Read Committed, Repeatable Read, Serializable).
* **Durability:** Enable **logging & backup mechanisms (**Enable database **replication** (Master-Slave setup)

#### ****Advantages of ACID:****

* **Data Integrity** – Ensures consistency.
* **Reliability** – Transactions are durable and recoverable.
* **Concurrency** **Control** – Prevents conflicts in multi-user environments.
* **Fault** **Tolerance** – System failures don’t cause data corruption.

#### ****Disadvantages of ACID:****

* **Performance Overhead** – More locking and logging can slow down performance.
* **Scalability Issues** – Difficult to scale horizontally (NoSQL databases are preferred for large-scale applications).
* **Complex Implementation** – Requires careful design and tuning.

### ****Isolation Levels in a Database****

In a multi-user database system, multiple transactions run concurrently. **Isolation levels** define how much one transaction is isolated from others. If isolation is too low, transactions can interfere with each other, leading to issues like **dirty reads, non-repeatable reads, and phantom reads**. If isolation is too high, performance might be affected.

##### ****1****. READ UNCOMMITTED

* Transactions can read **uncommitted** changes made by other transactions.
* This may result in **dirty reads** (reading data that might be rolled back).
* **Fastest but least safe** isolation level.

## ****Example Scenario****

1. Transaction A updates an account balance from **₹1000 to ₹5000** but has not yet committed.
2. Transaction B reads the balance and sees **₹5000**.
3. Transaction A **rolls back**, but Transaction B already used the incorrect data.

**Issue:** Users might see uncommitted (and potentially incorrect) data.

##### ****2. READ COMMITTED (Default in Most Databases)****

* Prevents **dirty reads** by ensuring transactions only read **committed data**.
* **Non-repeatable reads and phantom reads** can still happen.
* **Fixes dirty** reads but allows **non-repeatable reads** (values can change between two reads).

## ****Example Scenario****

1. Transaction A reads a **₹1000** balance.
2. Meanwhile, Transaction B updates the balance to **₹5000** and commits.
3. If Transaction A reads again, it sees **₹5000** (data changed mid-transaction).

##### ****3. REPEATABLE READ****

* Prevents **dirty reads and non-repeatable reads**.
* Ensures that if a transaction **reads** the same row twice, it **gets the same value**.
* **Phantom reads** (new rows appearing) can still happen.

## ****Example Scenario****

1. Transaction A reads a **₹1000** balance.
2. Transaction B updates it to **₹5000** and commits.
3. Transaction A **reads the balance again**, but still sees **₹1000** (repeatable read).

##### ****4. SERIALIZABLE (Highest Isolation)****

* **Prevents all three issues:** dirty reads, non-repeatable reads, and phantom reads.
* Achieved by **locking the entire table** or **executing transactions sequentially**.
* **Most secure but worst performance** (low concurrency).
* Ensures maximum consistency, but might slow down performance due to locking.

## ****Example Scenario****

1. Transaction A reads all account balances.
2. **While Transaction A is running, no other transaction can modify or insert new rows.**
3. This prevents **phantom reads**, ensuring complete data consistency.

#### ****Comparison of Issues in Different Isolation Levels****

| **Isolation Level** | **Prevents Dirty Read** | **Prevents Non-Repeatable Read** | **Prevents Phantom Read** | **Use Case** |
| --- | --- | --- | --- | --- |
| READ UNCOMMITTED | ❌ No | ❌ No | ❌ No | Fast reads, but risky |
| READ COMMITTED | ✅ Yes | ❌ No | ❌ No | General-purpose, good balance |
| REPEATABLE READ | ✅ Yes | ✅ Yes | ❌ No | Prevents most issues, but allows phantom reads |
| SERIALIZABLE | ✅ Yes | ✅ Yes | ✅ Yes | Highest consistency, but slowest |

#### ****When to Use Each Isolation Level****

| **Use Case** | **Recommended Isolation Level** |
| --- | --- |
| High-speed applications where dirty reads are acceptable | Read Uncommitted |
| General-purpose applications (default level in most DBs) | Read Committed |
| Financial applications requiring stable reads | Repeatable Read |
| Banking, accounting, or government transactions | Serializable |

### ****Understanding Dirty Reads, Non-Repeatable Reads, and Phantom Reads****

When multiple transactions execute concurrently, data inconsistencies can occur. These are classified as **Dirty Reads, Non-Repeatable Reads, and Phantom Reads** based on the type of inconsistency.

##### ****1. Dirty Read (Reading Uncommitted Data)****

* A **dirty read** occurs when a transaction reads data that has been modified by another transaction but **not yet committed**.
* If the modifying transaction **rolls back**, the first transaction will have read incorrect data.

**Example Scenario**

#### ****Step 1: Transaction A (Updating but Not Committed Yet)****

BEGIN TRANSACTION;

UPDATE accounts SET balance = 5000 WHERE id = 1; -- Changed from 1000 to 5000

-- Transaction A has not yet committed

#### ****Step 2: Transaction B (Reads the Uncommitted Value)****

SELECT balance FROM accounts WHERE id = 1;

-- Transaction B sees balance = 5000

#### ****Step 3: Transaction A Rolls Back****

ROLLBACK;

-- Balance goes back to 1000

* **Issue:** Transaction B read 5000, which never really existed after rollback.

##### ****2. Non-Repeatable Read (Different Values in Same Transaction)****

* A **non-repeatable read** occurs when a transaction reads the same row twice and gets different values because another transaction **updated and committed** in between.

## ****Example Scenario****

#### ****Step 1: Transaction A Reads Data****

BEGIN TRANSACTION;

SELECT balance FROM accounts WHERE id = 1;

-- Transaction A sees balance = 1000

#### ****Step 2: Transaction B Updates and Commits****

BEGIN TRANSACTION;

UPDATE accounts SET balance = 5000 WHERE id = 1;

COMMIT;

#### ****Step 3: Transaction A Reads the Data Again****

SELECT balance FROM accounts WHERE id = 1;

-- Transaction A now sees balance = 5000 (changed!)

* **Issue:** The data changed **within the same transaction**, causing inconsistency.

##### ****3. Phantom Read (New Rows Appearing in Same Transaction)****

* A **phantom read** occurs when a transaction reads a set of rows twice and gets **different numbers of rows** because another transaction **inserted new rows** in between.

## ****Example Scenario****

#### ****Step 1: Transaction A Reads a List of Accounts with Balance > 2000****

BEGIN TRANSACTION;

SELECT \* FROM accounts WHERE balance > 2000;

-- Transaction A gets 5 rows

#### ****Step 2: Transaction B Inserts a New Row and Commits****

BEGIN TRANSACTION;

INSERT INTO accounts (id, balance) VALUES (6, 3000);

COMMIT;

#### ****Step 3: Transaction A Reads the Same Query Again****

SELECT \* FROM accounts WHERE balance > 2000;

-- Transaction A now gets 6 rows instead of 5 (phantom row appeared!)

* **Issue:** The **number of records** changed within the same transaction.

#### ****Comparison of Dirty, Non-Repeatable, and Phantom Reads****

| **Issue Type** | **Cause** | **Data Read is Uncommitted?** | **Row Count Changes?** | **Fix with Isolation Level** |
| --- | --- | --- | --- | --- |
| **Dirty Read** | Another transaction updated but did not commit | Yes | No | Read Committed |
| **Non-Repeatable Read** | Another transaction committed an update | No | No | Repeatable Read |
| **Phantom Read** | Another transaction committed an insert/delete | No | Yes | Serializable |

### Race Condition

A **race condition** in a backend system occurs when two or more processes access shared data concurrently, and the outcome depends on the timing or order of execution. In retail systems, this is especially critical in scenarios like **shopping carts, inventory management, and checkout processes**, where accurate and consistent state is crucial.

**💡 Example: Race Condition in a Retail Cart System**

Imagine two users (or the same user from two devices) try to **purchase the last item of a product** at the same time:

1. The item’s stock is **1**.
2. Two requests come in **almost simultaneously** to add the product to their carts or proceed to checkout.
3. Both requests read the stock as **1**.
4. Both decide it's available and reduce it by 1.
5. Result: Stock becomes **-1**. Over-selling happens.

#### Solutions to Handle Race Conditions

#### ****1. Database Transactions with Locking****

Use **pessimistic locking** or **optimistic locking** to ensure consistent access.

##### Pessimistic Locking (SELECT FOR UPDATE)

BEGIN TRANSACTION;

SELECT stock FROM product WHERE id = 101 FOR UPDATE;

-- Check stock and reduce

UPDATE product SET stock = stock - 1 WHERE id = 101;

COMMIT;

# In Spring Boot (with JPA):

**@Lock(LockModeType.PESSIMISTIC\_WRITE)**

@Query("SELECT p FROM Product p WHERE p.id = :id")

Product findProductForUpdate(@Param("id") Long id);

* Ensures no other transaction can read/update until this is complete.
* Good for low-concurrency systems.

##### Optimistic Locking

Add a version field to the entity.

@Version

private Integer version;

Spring Data JPA throws OptimisticLockingFailureException if another transaction has modified the entity in between.

* Ideal when collisions are rare.
* Retry logic is needed on conflict.

#### 2. Synchronized Block in Distributed Locks

<https://www.youtube.com/watch?v=7eV4nib3Cm8>

#### ****3. Queueing (Serializing Requests)****

Push order requests into a **queue** (e.g., Kafka or RabbitMQ). Process them **one-by-one** in a consumer.

* Ensures serialized processing.
* Excellent for high-scale and consistency.

##### Optimistic Locking

Optimistic Locking is a **concurrency control mechanism** where you:

* Read a record.
* Modify and try to save it.
* If another transaction modified it in the meantime (based on a **version** field), an **OptimisticLockingFailureException** is thrown.

It does **not lock the DB row** like pessimistic locking — instead, it assumes that **collisions are rare** and handles them **only if they occur**.

🔁 Every time the entity is updated, version is incremented. If two transactions try to update the same row, only one will succeed.

| **id** | **name** | **stock** | **version** |
| --- | --- | --- | --- |
| 1 | Laptop | 3 | 1 |

|  |
| --- |
| @Builder  public class Product {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY)  private Long id;  private String name;  private int stock;  @Version  private int version;  }  @Transactional  public String checkout(Long productId, int quantity) {  try {  Product product = productRepository.findById(productId)  .orElseThrow(() -> new RuntimeException("Product not found"));  if (product.getStock() < quantity) {  return "Insufficient stock";  }  product.setStock(product.getStock() - quantity);  // Save triggers version check  productRepository.save(product);  return "Checkout successful";  } catch (OptimisticLockException | org.springframework.orm.ObjectOptimisticLockingFailureException e) {  return "Checkout failed due to concurrent update. Please try again.";  }  } |

## Summary Comparison

| **Feature** | **Optimistic Locking** | **Pessimistic Locking** |
| --- | --- | --- |
| **Locking strategy** | No actual DB lock; checks version during update | Acquires a DB-level lock on the row |
| **Concurrency assumption** | Conflicts are rare | Conflicts are likely |
| **Performance** | High (no locks until conflict occurs) | Lower (blocks other access until lock is released) |
| **Use case** | High-read, low-write systems | High-contention, critical-update scenarios |
| **Failure mode** | OptimisticLockException during save | Other threads/transactions are **blocked** or **deadlock** |
| **Retry logic** | Often needed | Not needed (but may block/wait longer) |

### BASE Properties in Databases

#### What is BASE?

* **BASE** stands for **Basically Available, Soft state, and Eventually consistent**.
* It is an alternative to the **ACID** model used in NoSQL and distributed databases.
* Unlike ACID, which ensures strong consistency, BASE **sacrifices strong consistency for availability and performance**.

#### ****BASE Properties Explained****

| **Property** | **Description** |
| --- | --- |
| **Basically Available** | The system guarantees availability even if some nodes fail. |
| **Soft State** | Data may change over time due to replication and eventual consistency. |
| **Eventually Consistent** | The system guarantees that data will become consistent over time but not immediately. |

##### ****1. Basically Available (High Availability)****

* Ensures that the system is **always available**, even if some data is not immediately up-to-date.

## ****Scenario:**** Social Media Posts

## A social media app ensures that a user can post updates instantly.

## If one database node is down, the post is stored in another node and synchronized later.

* **Advantage:** Users never see downtime.
* **Disadvantage:** Posts might take time to appear on all devices.

##### ****2. Soft State (Allowing Temporary Inconsistency)****

* The state of the database may change over time, even without user input, due to eventual consistency mechanisms.
* Replicas may temporarily hold different versions of data until synchronization occurs.
* Example:
  + In a distributed database, a user updates their profile picture. The change may take a few seconds to propagate across all servers.
  + Different servers may show different profile pictures temporarily, but they will eventually synchronize.

##### ****3. Eventually Consistent (Final Data Consistency)****

* Guarantees that, over time, all updates will be reflected across the system, but **immediate consistency is not required**.
* Example:
  + Social media posts might not be visible to all followers instantly, but after a short delay, everyone sees the same data.

#### ****When to Use BASE?****

* **Social Media Apps** → Speed is more important than accuracy (Facebook, Twitter).
* **E-Commerce Platforms** → Shopping cart updates can be eventual (Amazon, Flipkart).
* **Streaming Services** → Video recommendations don't need immediate consistency (Netflix, YouTube).
* **IoT & Big Data** → Handling massive volumes with distributed processing.

#### ****Advantages and Disadvantages of BASE****

## ****Advantages****

* **High Availability** → Always allows read/write, even if some nodes fail.
* **Scalability** → Handles large data loads efficiently.
* **Fault Tolerance** → Works well in distributed environments.
* **Better Performance** → Reduces database locks and speeds up transactions.

## ****Disadvantages****

* **Eventual Consistency** → No guarantee of immediate accuracy.
* **Complex Data Handling** → Developers must manage stale data issues.
* **Not Ideal for Banking** → Critical applications require strong consistency.

#### Final Thoughts

* **BASE is ideal for NoSQL databases** where **performance and availability** are more important than strict consistency.
* **ACID is necessary for SQL databases** where **data integrity** is crucial.
* **Choosing BASE vs. ACID depends on business needs**:
* Banking? → **Use ACID**
* Social Media? → **Use BASE**

### Ratelimiter

**Rate Limiting** is a technique used to **control the number of requests a client can make** to a server or service in a given time window. It helps to prevent the overloading of servers by limiting the number of requests that can be made in a given time frame.It helps to prevent a high volume of requests from overwhelming a server or API.

A diagram of a server

AI-generated content may be incorrect.

#### Why is Rate Limiting Needed?

| **Scenario** | **Reason** |
| --- | --- |
| **Prevent Abuse** | Stop users/bots from hammering your service (DoS, brute-force, abuse). |
| **Protect Backend Systems** | Avoid overloading downstream systems (DBs, APIs, queues). |
| **Fair Usage Enforcement** | Ensure fair resource allocation across clients or tenants. |
| **Cost Management in Cloud (e.g. Azure)** | Reduce operational costs by restricting overuse. High-frequency access increases your costs. Services like Azure Event Hubs, Cosmos DB, or API calls are often **billed per use**.  Rate limiting in Cosmos DB primarily revolves around **Request Units (RUs)**. |
| **Improve Reliability and User Experience** | Instead of letting a system go down under load   * Rate limit responses can return:   + 429 Too Many Requests   + With a **Retry-After** header |

#### Where to place the Rate Limiter - Client Side or Server Side ?

A rate limiter should generally be implemented on the **server side** rather than on the client side.

Placing the **rate limiter on the server side** is essential for **security, trust, and system protection**. Let’s break this down clearly:

##### 1. ****Security and Trust****

* You **can't trust the client** — it can be:
  + Hacked
  + Tampered with
  + Automated (bots, scrapers, etc.)

✅ Server-side rate limiting **enforces** rules regardless of who or what the client is.

🔎 Example: A user disables the client-side JavaScript and sends 1000 requests using a script — only the server can stop them.

##### 2. ****Prevents Abuse, Bots, and DDoS Attacks****

* Malicious users or bots can **bypass client-side limits** easily.
* Only server-side logic can reliably:
  + Detect unusual traffic patterns
  + Block abusive IPs or users
  + Throttle suspicious activities

✅ Server-side rate limiting is a **defensive layer** to protect your API and backend.

##### 3. ****Centralized Control and Visibility****

* On the server, you can:
  + Set global or per-user/IP limits
  + Monitor who is consuming how much
  + Analyze logs to identify abuse

✅ Server-side gives you **full observability** and **auditing capability**.

##### 4. ****Ensures Fair Usage (Especially in Multi-User Systems)****

* In SaaS or B2B platforms (e.g., retail chains), each user or store should get:
  + Fair access to shared resources
  + Isolation from noisy neighbors

✅ Server-side rate limiting lets you **limit per API key, token, or tenant**.

Example: Store A and Store B should each get 100 req/min — not 10,000 and 0.

##### 5. ****Protects Backend Resources****

* Backend systems like:
  + **Databases (e.g., Cosmos DB)**
  + **Blob storage**
  + **Payment APIs**
  + Have limits or costs

If you don’t throttle at the server, a sudden spike can:

* Crash your system
* Exhaust your RU quota (in Cosmos DB)
* Drive up cloud costs

✅ Server-side rate limiting **saves you money** and **maintains reliability**.

## **6. Supports Multi-Tier or Subscription-Based Limits**

* You can offer:
  + Free users → 100 req/day
  + Premium users → 1000 req/min
* These checks **must happen on the server** where subscription and identity data lives.

✅ Server-side rate limiting enables **usage-based business models**.

## **7. Avoids Logic Duplication and Inconsistency**

* If you rely on clients to limit themselves:
  + Mobile, web, desktop clients all need duplicate logic
  + What if one forgets to implement it correctly?

✅ Server-side rate limiting is **single-source-of-truth enforcement**

#### Real-Life Use Case: Retail System

Imagine:

* Your retail company has 500 stores.
* Each store syncs product and planogram data to **Azure Cosmos DB** via a backend API.

If a few stores sync aggressively:

* Cosmos DB RUs get exhausted → 429 errors
* Other stores face failures
* Azure cost spikes

✅ Server-side rate limiting:

* Ensures each store gets fair access (e.g., 200 RUs/sec)
* Protects Cosmos from overload
* Keeps the system stable under high load

#### Algorithms to Design a Rate Limiter API

1. Fixed Window Counter
2. Sliding Window Log
3. Sliding Window Counter
4. Token Bucket
5. Leaky Bucket

#### Where to Place the Rate Limiter on the Server Side?

##### 1. ****API Gateway / Reverse Proxy Level****

## Description:

* Placed **before** requests reach your backend services.
* Best for coarse-grained throttling (e.g., per IP or API key).

## Example Technologies:

* **Azure API Management (APIM)**
* **Kong**, **NGINX**, **AWS API Gateway**, **Envoy**

## Example Use Case:

* Limit all clients to 1000 requests per minute.
* Block specific abusive IPs or tokens before backend is hit.

##### 2. ****Load Balancer / Ingress Controller Level****

## Description:

* Network-level request throttling or shaping.
* Mostly used in Kubernetes or microservice deployments.

## Example Technologies:

* **Azure Application Gateway**
* **NGINX Ingress Controller**
* **Istio**

## Use Case:

* Drop or delay excessive requests even before they hit the API layer.

##### 3. ****Middleware / Filter in Web Application (Spring Boot, etc.)****

## Description:

* Implements rate limiting as a filter, interceptor, or AOP advice.
* Can access headers, tokens, and request path.

## Use Case:

* Fine-grained per-user, per-endpoint logic.
* Integrates well with business rules.

## Pros:

* Custom logic per user/role/endpoint
* Uses session, token, or headers

## Cons:

* Still consumes minimal app resources before rejecting

@Component

public class RateLimitingFilter extends OncePerRequestFilter {

public void doFilterInternal(...) {

if (!rateLimiter.allow(userId)) {

response.setStatus(429);

return;

}

filterChain.doFilter(request, response);

}

}

#### 

##### 4. ****Service/Business Logic Layer****

## Description:

* Place rate limiter directly in your service methods.
* Good for feature-level or internal call throttling.

## Example:

public Product getProduct(String userId) {

if (!rateLimiter.allow(userId)) {

throw new TooManyRequestsException();

}

return db.fetchProduct();

}

## Use Case:

* Limit only expensive operations (e.g., search, sync, reporting)
* Different limits for different services (e.g., retail stores vs. admins)

##### 5. ****Database or Cache Layer (Advanced)****

## Description:

* Store rate limiting state in shared cache (e.g., Redis, Cosmos DB).
* Use Redis INCR with TTL for distributed rate limiting.

## Use Case:

* You have multiple instances of your app and need global counters.
* You need to persist rate state beyond memory.

## Example in Redis:

INCR user:123:requests

EXPIRE user:123:requests 60

#### Placement of Rate Limiter in Server-Side Architecture — Explained with a Retail Use Case

A company that provides APIs for **store managers, regional heads, and internal systems** to fetch inventory, sales, and planogram data using a retail backend system.

##### 💼 ****Retail Use Case Overview****

* Users: Store Managers, Area Heads, POS Terminals, Mobile Apps
* APIs:
  + /getInventory
  + /getStoreSales
  + /syncPlanogram
  + /getProductDetails

Each of these APIs can be subject to misuse or overuse, potentially crashing backend services or degrading performance for others.

##### 🔄 ****Flow with Rate Limiting at Different Server-Side Layers****

##### 1. ****Load Balancer (Comes First)****

**Role in Retail:**  
Distributes traffic to the appropriate region or data center (e.g., North India vs. South India retail zones).

**Example:**  
If a regional head for Chennai logs in, Azure Front Door routes them to the South India cluster.

**Rate Limiting:**

* Mostly **not done here**.
* Focus is on routing, not user-specific logic.
* Can help throttle sudden traffic surges.

##### 2. ****API Gateway (Azure API Management)****

**Role in Retail:**  
Handles global rules, IP-based throttling, and API key validation.

You can apply IP throttling rules like:

* Limit each IP to **100 requests per minute**
* If exceeded, respond with 429 Too Many Requests

**Use Case:**

* Store-level API key → 1000 req/min
* Regional head API key → 5000 req/min

**Why Here?**

* Protects backend from bots or accidental loops.
* Cost-effective since fewer requests hit the app.

##### 3. ****Middleware/Filter Layer in Spring Boot (Application)****

**Role in Retail:**  
Add per-user or per-role limits using headers or tokens.

**Example in Spring Boot:**

if (!rateLimiter.allow(userId)) {

throw new RateLimitException();

}

**Use Case:**

* Store manager: max 100 req/min to /getInventory
* POS terminal: max 5 syncs/hour for /syncPlanogram

**Why Here?**

* You know the user’s role and store.
* Logic can be dynamic (e.g., VIP stores may have higher limits).

##### 4. ****Service Layer (Business Logic)****

**Role in Retail:**  
Limit only expensive or business-critical methods.

**Use Case:**

* Limit /syncPlanogram to 3 times/day per store.
* Restrict report generation API to once every 5 minutes per user.

**Why Here?**

* Full access to business context
* Very fine-grained control

##### 5. ****Distributed Store (e.g., Redis)****

**Used for:**

* Multi-instance consistency
* Cross-region rate limits
* Persistent state even after server restarts

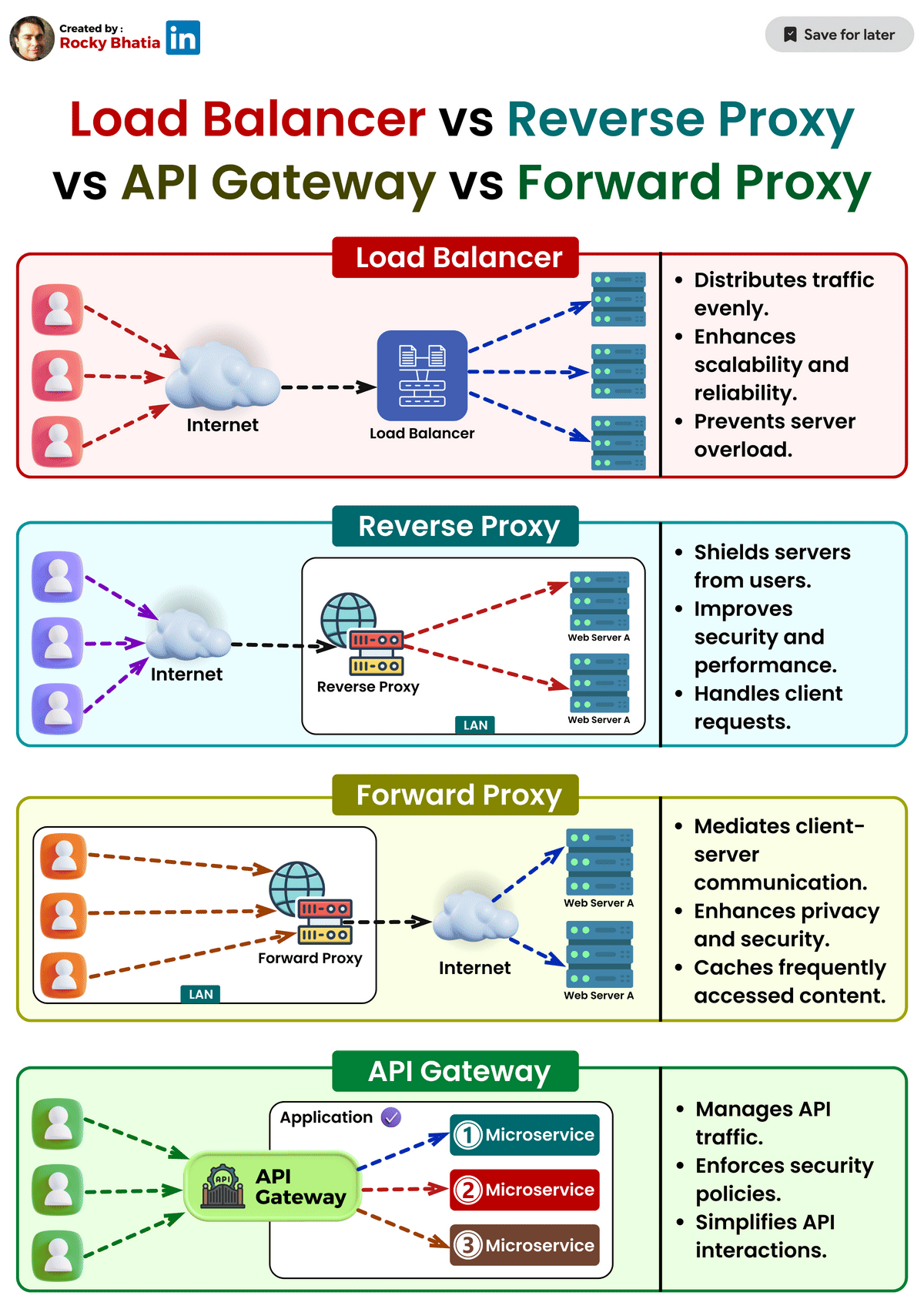
**Use Case:**

* Regional head accessing from multiple locations — counters must be shared.

#### Reference:

* <https://www.geeksforgeeks.org/how-to-design-a-rate-limiter-api-learn-system-design/>
* <https://medium.com/geekculture/system-design-design-a-rate-limiter-81d200c9d392>

### Load Balancer vs Forward Proxy vs API Gateway vs Reverse Proxy



### Load Balancer

A **Load Balancer** is a networking component or service that **distributes incoming network traffic across multiple backend servers or services** to ensure:

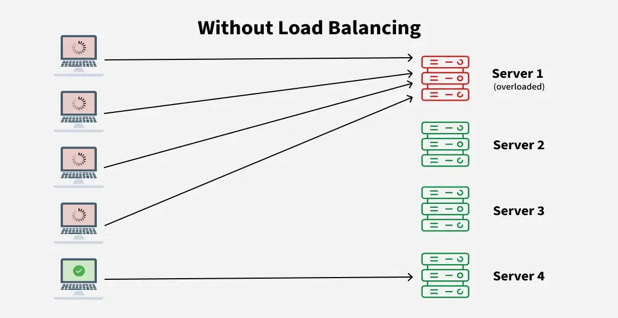
* High availability
* Reliability
* Performance
* Scalability

It acts as a **traffic manager**, sitting between the client and backend servers.

#### ****Why Use a Load Balancer?****

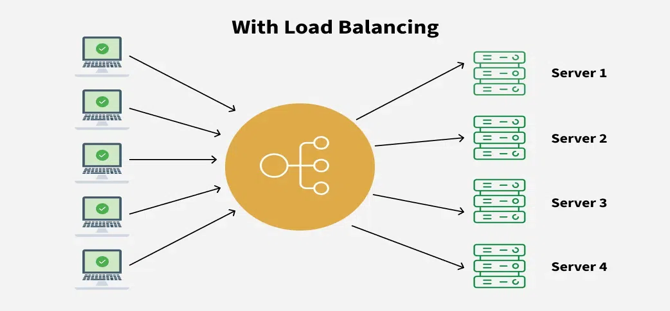
Without a load balancer:

* A single server handles all requests
* If that server crashes, the system becomes unavailable
* Traffic spikes can overwhelm it



With a load balancer:

* Requests are evenly distributed
* Traffic is managed dynamically
* Failures in one server do not affect availability



#### ****Problems Without a Load Balancer****

##### 1. ****Single Point of Failure (SPOF)****

* If your only server crashes or goes down, **your entire application becomes unavailable**.
* No redundancy means **zero fault tolerance**.

**Impact**: Downtime, lost revenue, and poor user experience.

##### 2. ****Server Overload****

* All traffic hits one server, leading to **high CPU, memory, and I/O usage**.
* Once resource limits are reached, the server slows down or crashes.

**Impact**: Sluggish performance, timeouts, or 500 errors.

##### 3. ****No Scalability****

* Adding more servers doesn’t help unless there’s a mechanism to **distribute traffic**.
* You can’t grow your app smoothly with user demand.

**Impact**: Stagnant architecture, poor UX under high load.

##### 4. ****Uneven Traffic Distribution****

* If multiple servers are manually configured with DNS round-robin or hardcoded IPs, **some servers may be overloaded** while others are underused.

**Impact**: Wasted resources and imbalanced performance.

##### 5. ****Difficult Maintenance****

* You can't easily **take a server down for maintenance** without causing disruption.
* There’s no automatic rerouting of traffic to healthy instances.

**Impact**: Risk of manual errors and downtime.

##### 6. ****No Health Monitoring****

* Without a load balancer, there's no automated way to check if a server is alive or healthy.
* Requests could be sent to **dead or unresponsive servers**.

**Impact**: Users may face intermittent failures.

##### 7. ****No SSL Offloading****

* Every server must handle **HTTPS encryption and decryption**, adding load.
* SSL certificates must be managed on each server individually.

**Impact**: Redundant configuration, increased CPU usage.

##### 8. ****No Session Persistence / Sticky Sessions****

* Without a centralized controller, **user sessions can jump between servers**, breaking login states or shopping carts.

**Impact**: Broken user experience.

#### Key Characteristics of Load Balancers

##### 1. ****Traffic Distribution****

* Evenly distributes incoming traffic across multiple backend servers.
* Prevents any single server from becoming a bottleneck.

##### 2. ****High Availability & Fault Tolerance****

* Continuously checks the health of servers.
* Automatically routes traffic away from unhealthy or down servers.
* Supports failover to ensure minimal downtime.

##### 3. ****Scalability****

* Allows horizontal scaling by adding/removing backend servers.
* Handles sudden traffic spikes without impacting performance.

##### 4. ****Health Checks & Monitoring****

* Periodically pings servers to ensure they are responsive.
* Only sends traffic to servers that are marked as healthy.

##### 5. ****Load Balancing Algorithms****

* Uses strategies like Round Robin, Least Connections, or IP Hashing to decide how to route traffic.

##### 6. ****Session Persistence (Sticky Sessions)****

* Optionally ensures that a user stays connected to the same backend server during a session.
* Useful for apps that store session data in memory.

##### 7. ****SSL Termination****

* Decrypts HTTPS traffic at the load balancer level before forwarding to servers.
* Reduces CPU load on backend servers.

##### 8. ****Layer 4 and Layer 7 Support****

* **Layer 4 (Transport Layer)**: Routes based on IP address and port (e.g., TCP, UDP).
* **Layer 7 (Application Layer)**: Routes based on HTTP methods, URL paths, headers, cookies, etc.

##### 9. ****Reverse Proxy Functionality****

* Often acts as a reverse proxy, sitting in front of servers to handle all incoming requests.
* Provides an abstraction layer and security buffer.

##### 10. ****Security Features****

* Protects backend servers from direct exposure to the internet.
* Supports DDoS mitigation, Web Application Firewalls (WAF), and access control lists.

##### 11. ****Global Load Balancing (Optional)****

* Routes traffic across multiple geographic regions.
* Uses geo-location and latency-based routing for global apps.

#### ****How a Load Balancer Works (Step-by-Step)****

##### ****1. Client Sends a Request****

* A user tries to access an application or website (e.g., example.com).
* The domain name resolves to the **IP address of the load balancer**.

##### ****2. Load Balancer Receives the Request****

* It acts as the **entry point** for incoming traffic.
* Can be physical (hardware), virtual (software), or cloud-based (e.g., AWS ALB/NLB, Azure LB).

##### ****3. Health Check of Backend Servers****

* Load balancer regularly pings backend servers to see if they are **healthy and responsive**.
* If a server is **unhealthy**, it’s **excluded** from traffic routing.

##### ****4. Routing Decision Based on Algorithm****

* Load balancer chooses a backend server based on configured algorithms:
  + **Round Robin**
  + **Least Connections**
  + **IP Hash**
  + **Weighted Distribution**

##### ****5. Forwards Request to Selected Server****

* The chosen server receives the request and processes it.
* The load balancer may:
  + Maintain **session persistence** (sticky session),
  + Decrypt SSL (if using SSL termination),
  + Apply routing rules (Layer 7 logic).

##### ****6. Server Sends Response****

* The selected backend server generates a response.
* Response is sent **back through the load balancer** to the client.

##### ****7. Repeat for Each Incoming Request****

* This process continues for every new client or request, dynamically choosing the best backend server.

A diagram of a computer network

AI-generated content may be incorrect.

#### ****Types of Load Balancers****

#### **Based on OSI Layer**

##### ****1. Layer 4 Load Balancer (Transport Layer)****

* Operates at the **TCP/UDP** level.
* Routes traffic based on **IP address and port number**.
* Doesn't inspect content of the request.
* **Faster and lightweight**, suitable for low-latency needs.

Examples: AWS NLB, Azure Load Balancer, HAProxy (TCP mode)

##### ****2. Layer 7 Load Balancer (Application Layer)****

* Operates at the **HTTP/HTTPS** level.
* Makes routing decisions based on **URL path, headers, cookies, etc.**
* Supports content-based routing, SSL termination, and WAF integration.

Examples: AWS ALB, Azure Application Gateway, NGINX, Traefik

## ****Based on Deployment/Implementation****

##### 3. ****Hardware Load Balancer****

* Physical appliances installed in data centers.
* High performance, often used in enterprises or legacy setups.
* Expensive and less flexible.

Examples: F5 BIG-IP, Citrix ADC (NetScaler)

##### 4. ****Software Load Balancer****

* Runs on general-purpose servers or containers.
* Highly flexible and configurable.
* Can be deployed in the cloud, on-premise, or hybrid.

Examples: NGINX, HAProxy, Envoy, Traefik

##### 5. ****Cloud Load Balancer****

* Fully managed by cloud providers.
* Scales automatically and integrates with cloud services.
* Supports advanced features like geo-routing, autoscaling, SSL offloading.

Examples:

* **AWS**: ALB, NLB, Global Accelerator
* **Azure**: Application Gateway, Azure Load Balancer
* **GCP**: Cloud Load Balancing

## ****Other Specialized Types****

##### 6. ****Global Load Balancer (Geo Load Balancer - GSLB)****

* Routes users to the nearest data center based on **geo-location, latency, or health**.
* Used in **multi-region deployments**.

Examples: AWS Route 53 (with Geo routing), Cloudflare Load Balancing

##### 7. ****DNS Load Balancer****

* Uses **DNS round-robin** or intelligent routing via DNS responses.
* Very simple but lacks health check precision.

Examples: AWS Route 53, Azure Traffic Manager

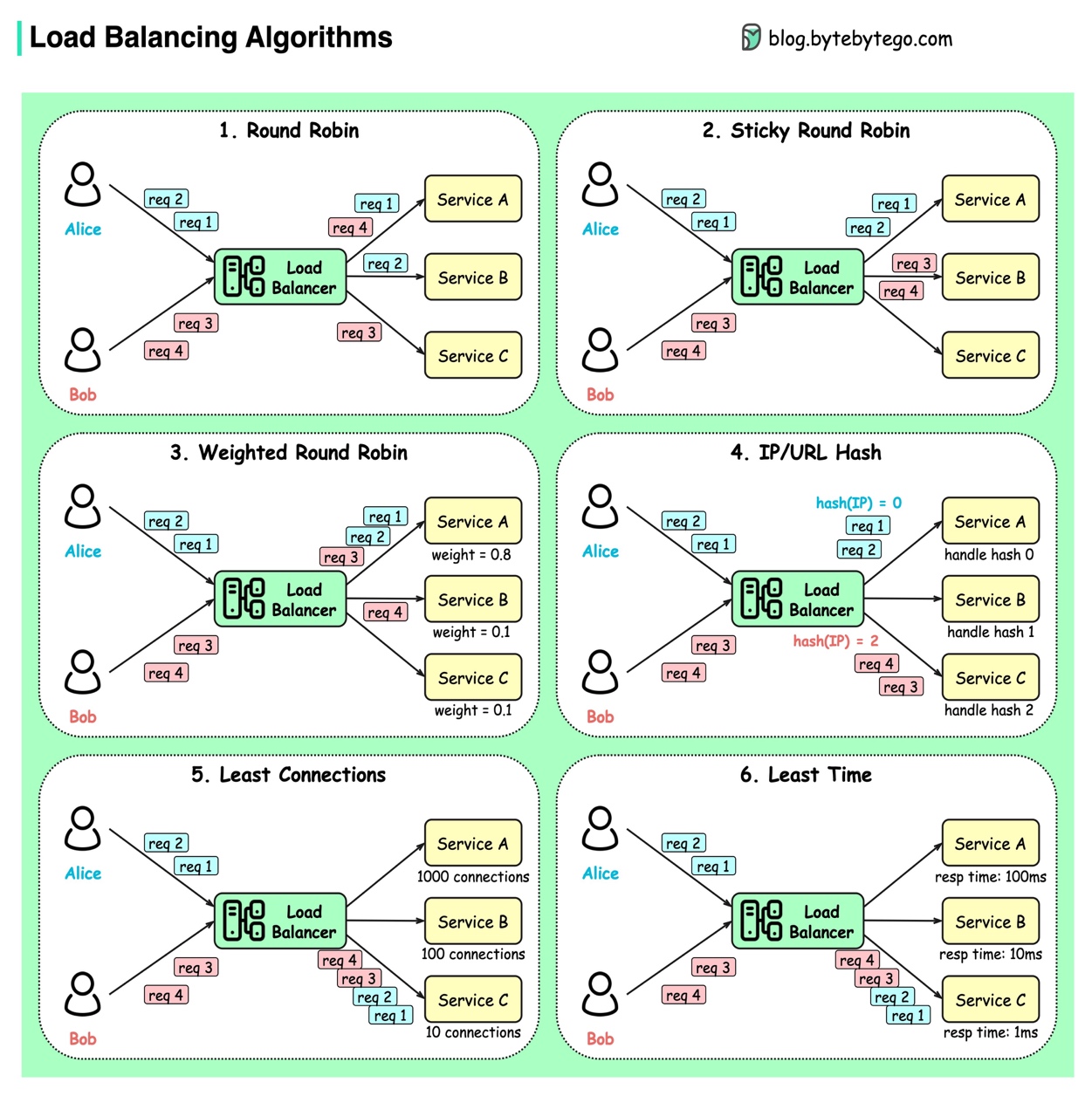
##### 8. ****Reverse Proxy Load Balancer****

* Acts as a **gateway that handles client requests and forwards them to backend servers**.
* Often used with SSL termination, caching, compression.

Examples: NGINX, Apache HTTP Server, Envoy

#### Load Balancing Algorithms

We need a [load-balancing algorithm](https://www.geeksforgeeks.org/load-balancing-algorithms/) to decide which request should be redirected to which backend server. The different system uses different ways to select the servers from the load balancer. Companies use varieties of load-balancing algorithm techniques depending on the configuration



## ****1. Static Load Balancing Algorithms****

These algorithms don't rely on system state (like CPU or memory). They are simple and fast.

##### 🌀 a. ****Round Robin****

* **How it works:** Cycles through backend servers in a circular manner.
* **Use Case:** Uniform tasks like serving static web pages.
* **Example:**  
  A retail website rotates traffic among 3 identical catalog APIs:  
  Request 1 → Server A, Request 2 → Server B, Request 3 → Server C...

##### ⚖️ b. ****Weighted Round Robin****

* **How it works:** Assigns weights to servers based on capacity. Higher-weight servers get more traffic.
* **Use Case:** When servers have different CPU/RAM.
* **Example:**  
  Server A (weight 3), B (weight 2), C (weight 1):  
  Traffic ratio will be 3:2:1.

## ****2. Dynamic Load Balancing Algorithms****

These make decisions based on real-time server or network load.

##### a. ****Least Connections****

* **How it works:** Routes new traffic to the server with the fewest active connections.
* **Use Case:** Good for long-lived connections (e.g., streaming, sessions).
* **Example:**  
  Checkout APIs with active sessions go to servers with less ongoing activity.

##### b. ****Least Response Time****

* **How it works:** Routes to the server that responds fastest.
* **Use Case:** Dynamic workloads where speed is crucial.
* **Example:**  
  Route customers to a pricing API that returns the response fastest at that moment.

##### c. ****Resource-Based / Load-Aware****

* **How it works:** Uses metrics like CPU, memory, or latency to route traffic.
* **Use Case:** Microservices with unpredictable workloads.
* **Example:**  
  Route large inventory update requests to servers with higher free memory.

## ****3. Adaptive & Custom Algorithms****

These are more intelligent, often involving hashing, persistence, or learning.

##### a. ****IP Hash****

* **How it works:** Hashes client IP to select a specific server.
* **Use Case:** Session stickiness (user always goes to the same backend).
* **Example:**  
  Loyalty program login routes the same customer to the same session server.

##### b. ****Consistent Hashing****

* **How it works:** Hashes a key (e.g., store ID, user ID) and maps it to a ring of servers.
* **Use Case:** Caching, distributed systems like NoSQL DBs or CDN.
* **Example:**  
  Store #5031 requests always go to the same product cache node.

##### c. ****AI/ML-Based Load Balancing****

* **How it works:** Predictive algorithms route based on traffic patterns, demand forecast.
* **Use Case:** Highly scalable retail platforms with variable workloads.
* **Example:**  
  During Black Friday, a smart LB predicts surges and reroutes dynamically before overload occurs.

#### Challenges of using Load Balancers

1. Single Point of Failure: Load balancers might create a single point of failure even though they improve fault tolerance. Issues with the load balancer itself could cause traffic distribution to be disrupted.
2. Complexity and Cost: High-quality load balancing solutions may be expensive, and load balancer implementation and management can be complicated. This covers load balancers for both software and hardware.
3. Configuration Challenges: Configuring load balancers correctly can be challenging, especially when dealing with complex application architectures or diverse server environments.
4. Potential for Overhead: Depending on the load balancing technique and configuration, there may be additional overhead in the form of delay and processing time, even though modern load balancers are designed to lessen this effect.
5. SSL Inspection Challenges: When SSL termination is performed at the load balancer, it may introduce challenges related to SSL inspection and handling end-to-end encryption.

Reference :

* <https://www.geeksforgeeks.org/what-is-load-balancer-system-design/>
* <https://www.geeksforgeeks.org/load-balancing-algorithms/>
* <https://www.cloudflare.com/en-gb/learning/performance/what-is-load-balancing/>

### API Gateway / Reverse Proxy /Forward Proxy

### Service Mesh

### Caching

**Caching** is a technique used to **store frequently accessed data in a temporary storage layer (cache)** so that future requests for that data can be served faster. Instead of fetching from a slow backend (like a database or external API), the data is fetched from the cache, improving performance and reducing load on primary systems.

## ****How Does Cache Work?****

1. **Check the cache**: When a request for data is made, the system first checks the cache.
2. **Cache hit**: If the data is found, it's returned immediately.
3. **Cache miss**: If not found, the system fetches it from the primary data source, returns it, and stores it in the cache for future use

## ****Types of Cache****

1. **In-memory cache**: Stores data in RAM (e.g., Redis, Memcached).
2. **Disk cache**: Stores on local disk for persistent caching.
3. **Distributed cache**: Shared cache across multiple servers (e.g., Redis Cluster).
4. **Application-level cache**: Local to the application instance (e.g., Guava, Caffeine).
5. **CDN (Content Delivery Network)**: Cache for static content like images and videos.

## ****Applications of Caching****

* **Database query caching**
* **API response caching**
* **Session storage**
* **Web page caching (CDN)**
* **DNS resolution caching**
* **Authentication/Authorization token caching**
* **E-commerce – product list/details, cart info, inventory data**

## ****Advantages of Using Caching****

1. ⚡ **Improves performance** (faster response times)
2. 🎯 **Reduces load** on backend servers and databases
3. 📉 **Decreases latency** and network traffic
4. 💸 **Saves cost** by reducing compute and DB usage
5. 🔄 **Enhances scalability**

## ****Disadvantages of Using Caching****

1. ❗ **Stale data** if cache is not properly invalidated
2. 🧠 **Complexity** in managing cache consistency
3. 📦 **Memory consumption** (especially for in-memory caches)
4. 🔄 **Eviction issues** – right data might get removed
5. 🧪 **Difficult to test** cache-dependent logic

## 🧹 ****Cache Invalidation Strategies****

Cache invalidation ensures that the cache doesn’t serve stale data.

For systems that use caching to improve performance, cache invalidation is essential. Data is temporarily kept for faster access when it is cached. However, the cached version goes out of date if the original data changes. In order to guarantee that users obtain the most recent information, cache invalidation techniques make sure that out-of-date records are either updated or deleted.

* Common strategies include time-based expiration, where cached data is discarded after a certain time, and event-driven invalidation, triggered by changes to the underlying data.
* Proper cache invalidation optimizes performance and avoids serving users with obsolete or inaccurate content from the cache.

## 🗑️ ****Eviction Policies of Caching (When Cache is Full)****

For caching systems to effectively manage their limited cache capacity, eviction policies are essential. An eviction policy decides which existing item to remove when the cache is full and a new item needs to be stored.

1. **LRU (Least Recently Used)**: Removes the least recently accessed data.
2. **LFU (Least Frequently Used)**:Removes items used the fewest times.
3. **FIFO (First-In First-Out)**:Removes oldest inserted item.
4. **Random Replacement**: Randomly removes an entry (used in Memcached).

##### Cache Invalidation and the Methods to Invalidate Cache

**Cache Invalidation** is the process of **removing or updating** stale or outdated data in the cache to maintain consistency with the source of truth (usually a database or external service).Without proper invalidation, caches may return **stale/incorrect data**, which can lead to serious bugs and user confusion.

## ****Why is Cache Invalidation Important?****

* Imagine an e-commerce system:
  + A product's price is updated in the DB, but the cache still returns the old price.
  + This causes **data inconsistency**, **wrong billing**, and a **bad user experience**.

Hence, cache must be invalidated properly when:

* Data is updated
* Data is deleted
* New data is added
* Cache entry expires (TTL)

## Cache Invalidation Methods (Strategies)

#### 1. ****Time-based Invalidation (TTL - Time to Live)****

**How it works**:

* When data is written to the cache, it's stored with an expiration time.
* After TTL expires, the cache evicts it and fetches fresh data on the next request.

**Pros:**

* Simple to implement
* Good for data that changes rarely or predictably

**Cons**:

* + May serve stale data during the TTL window

#### ****Write-through Cache Invalidation****

**How it works**:

* + **Every write** (insert/update/delete) is written to both **DB and cache simultaneously**.
  + Read operations are always from cache.

**Flow**:

User updates → Write to DB + Write to Cache → Future reads from cache

**Pros**:

* Always has fresh data in cache
* Easy to reason about

**Cons**:

* Slower writes (since both cache & DB are involved)
* Higher write latency

#### ****Write-behind (Write-back) Cache****

**How it works**:

* + Writes go to the **cache first**.
  + Cache **asynchronously updates** the DB after a delay or in batches.

**Flow**:

User updates → Write to cache → Async write to DB later

**Pros**:

* + Faster writes
  + Reduces DB write load

**Cons**:

* + Risk of **data loss** if cache crashes before syncing to DB
  + Requires careful handling of failure scenarios

#### Cache-aside (Lazy Loading) – ⭐ Most Common

* Also called **Lazy Cache**.
* Application **checks the cache first**:
  + If data is found → return.
  + If not → fetch from DB → update cache → return data.
* On write/update/delete: **invalidate** or **update** cache manually.

**Flow**:

Read:

if (cache has key)

return cache[key]

else

data = db.query()

cache.set(key, data)

return data

Write:

db.update()

cache.delete(key) // Invalidate manually

|  |
| --- |
| @Cacheable(value = "products", key = "#productId")  public Product getProductById(Long productId) {  return productRepository.findById(productId).orElse(null);  }  @CacheEvict(value = "products", key = "#productId")  public void updateProduct(Long productId, Product product) {  productRepository.save(product);  }  @Cacheable: Used during **read** to check cache and fallback to DB.  @CacheEvict: Explicitly clears cache during **update**.  → This is **Cache-aside pattern** with **explicit invalidation on write**. |

**Pros**:

* Cache only contains frequently accessed data
* Reduces memory footprint

**Cons**:

* Possible cache miss after data updates
* Risk of stale data if invalidation is missed

#### ****Explicit Invalidation (Manual)****

#### The application explicitly calls cache.invalidate(key) or cache.delete(key) when the data changes.

**Example (in Java/Spring Boot):**

@CacheEvict(value = "product", key = "#productId")

public void updateProduct(Long productId, Product product) {

productRepository.save(product);

}

**Pros**:

* Precise control
* Works well for targeted invalidations

**Cons**:

* Requires discipline to ensure every update is paired with a cache eviction

#### Event-driven Invalidation (Message Queue based)

When data is changed, a **message (event)** is sent to a pub-sub system (Kafka, RabbitMQ), and all caches subscribed to that topic **invalidate the relevant keys**.

**Flow**:

Update DB → Publish "product-updated-123" → Subscribers invalidate "product:123" in cache

**Pros**:

* Useful in distributed environments
* Keeps multiple caches in sync

**Cons**:

* Complex to implement
* Requires reliable messaging and listener setup

## Which Strategy to Use When?

| **Use Case** | **Recommended Strategy** |
| --- | --- |
| Frequently changing data | Cache-aside or Write-through |
| Rarely changing data | Time-based (TTL) |
| Highly read-heavy system | Write-through |
| Performance-critical writes | Write-behind |
| Multi-instance distributed cache | Event-driven |
| Small app with simple needs | Manual (explicit) + TTL |

## Summary Table

| **Strategy** | **Read Path** | **Write Path** | **Complexity** | **Consistency** |
| --- | --- | --- | --- | --- |
| TTL | Fast | No change | Easy | Weak |
| Write-through | Cache | DB + Cache | Medium | Strong |
| Write-behind | Cache | Cache → Async to DB | High | Weak |
| Cache-aside | DB if miss | DB + Manual invalidate cache | Medium | Medium |
| Explicit | Cache | Manual invalidate after DB | Low-Medium | Medium |
| Event-driven | Cache | DB + publish event | High | Strong |

#### ✅ ****Cache-aside vs Explicit****:

* ✅ **Cache-aside**: You **read** through cache, and **update/invalidate** on writes.
* ✅ **Explicit**: You only **manually evict/update** cache during **writes** — usually part of cache-aside, but can exist standalone in custom systems.

🔄 In most real-world apps, **cache-aside + explicit invalidation on write** is used together for correctness and simplicity.

##### Cache Eviction Policies

## What is a Cache Eviction Policy?

A **cache eviction policy** determines **which data to remove** from the cache **when the cache is full**, or when a cache entry is no longer valid or needed.

## ****Why Are Eviction Policies Important?****

* Cache memory is limited (e.g., Redis on a 2GB server).
* You cannot cache everything forever.
* To maintain **high hit rates**, you must evict the **least useful** data.

## ****Common Cache Eviction Policies****

#### 1. ****LRU (Least Recently Used)**** – ⭐ Most Common

* Removes the **least recently accessed** item.
* Based on the idea that "recently used = likely to be used again."

🛠 Example:

Cache: [A (5 mins ago), B (2 mins ago), C (1 min ago)]

New item D added → Evict A (oldest usage)

📈 Ideal for:

* Session management
* API response caching
* General-purpose caching

🔧 Tools:

* Redis (default eviction: volatile-lru)
* Java’s LinkedHashMap with access-order

#### 2. ****LFU (Least Frequently Used)****

* Removes the **least often accessed** item (low frequency).
* Prioritizes **hot items** that are accessed often.

🛠 Example:

Cache:

A (used 10 times)

B (used 3 times)

C (used 1 time)

→ Evict C

📈 Ideal for:

* Recommendation systems
* Product pages in e-commerce (frequently viewed products)

🔧 Redis has volatile-lfu and allkeys-lfu eviction strategies

#### 3. ****FIFO (First-In First-Out)****

* Evicts the **oldest inserted** item, regardless of access.

🛠 Example:

Inserted: A → B → C

→ Insert D → Evict A

📈 Ideal for:

* Queues
* **When freshness is more important than popularity**

❌ Cons:

* Might evict frequently used items if they were added earlier

#### 4. ****MRU (Most Recently Used)****

* Opposite of LRU: **Evicts the most recently accessed item**.
* Based on the assumption that recently used items are **less likely** to be reused soon.

🛠 Example:

A (used 10 mins ago), B (used 3 mins ago), C (just used)

→ Evict C (most recently used)

📈 Rarely used, but applicable when:

* Access patterns indicate you need old data more than recent data
* Undo operations in editors

#### 5. ****Random Eviction****

* Randomly selects an item to evict.

📈 Ideal for:

* Very simple systems
* When caching logic is stateless or eviction overhead must be low

🔧 Supported in Memcached and Redis (allkeys-random, volatile-random)

❌ Cons:

* Not optimized; may evict important data

#### 6. ****No Eviction (Fail on Full)****

* If the cache is full, **don’t store new data**.
* Read/write requests fail or bypass the cache.

📈 Useful in:

* Strict systems where consistency matters more than caching (e.g., financial systems)

🔧 Redis config: noeviction

## ****Redis Eviction Policy Modes****

Redis supports various eviction policies (set with maxmemory-policy):

| **Policy** | **Description** |
| --- | --- |
| volatile-lru | LRU for keys with expiry set   |  | | --- | | (LRU eviction applies only to keys that have an expiration (TTL) set) | |
| allkeys-lru | LRU for all keys (LRU eviction applies to **all keys**, **with or without TTL**) |
| volatile-lfu | LFU for keys with expiry set |
| allkeys-lfu | LFU for all keys |
| volatile-random | Random among expirable keys |
| allkeys-random | Random among all keys |
| volatile-ttl | Evict keys with nearest expiry |
| noeviction | Don’t evict, return error when full |

## Reference

* <https://www.geeksforgeeks.org/caching-system-design-concept-for-beginners/>
* <https://www.geeksforgeeks.org/cache-eviction-policies-system-design/>
* <https://origin.geeksforgeeks.org/system-design/cache-invalidation-and-the-methods-to-invalidate-cache/>
* <https://www.geeksforgeeks.org/system-design/cache-aside-pattern/>

Caching in a **distributed system** is a performance optimization technique where frequently accessed data is stored closer to the application layer to reduce latency, load on underlying systems (like databases), and improve response times. In distributed systems, caching becomes more complex due to scalability, consistency, fault tolerance, and coordination challenges.

## Why Use Caching in Distributed Systems?

* **Reduce Latency**: Serve data faster by avoiding calls to slower storage layers.
* **Improve Throughput**: Handle more requests per second.
* **Reduce Backend Load**: Decrease the number of calls to the database or external APIs.
* **Increase Availability**: Cached data can sometimes serve requests even if the main service is down.

## Types of Caches in Distributed Systems

| **Cache Type** | **Description** |
| --- | --- |
| **Client-side cache** | Cache located on the client (e.g., browser or mobile app).  - Browser Cache (HTTP caching, localStorage, sessionStorage) - Service Workers - IndexedDB - Apollo Client Cache (GraphQL) - Room (Android) |
| **Server-side (local) cache** | Cache lives in the server’s memory. Fast but not shared across servers.  - Guava Cache (Java) - Caffeine (Java) - Ehcache (local mode) - Spring @Cacheable with ConcurrentMapCache |
| **Distributed cache** | Cache is shared across nodes.  - **Redis** (Clustered) - **Memcached** - **Hazelcast** - **Apache Ignite** - **Ehcache with Terracotta** |

When your dataset size is small, it’s usually enough to keep all the cache data on one server.

But as the system gets **bigger**, the cache size also gets bigger and a **single-node cache** often falls short when scaling to handle millions of users and massive datasets.

In such scenarios, we need to **distribute** the cache data across multiple servers.

This is where **distributed caching** comes into play.

#### Reference

* <https://www.geeksforgeeks.org/caching-system-design-concept-for-beginners/>
* <https://blog.algomaster.io/p/distributed-caching>
* <https://www.geeksforgeeks.org/what-is-a-distributed-cache/>

### Distributed Caching

### CDN

### Sharding vs Partitioning

### Distributed Cart Design