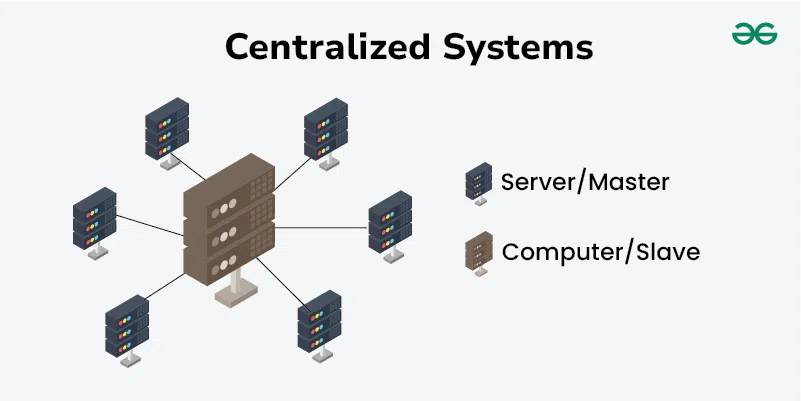
# TECHNICAL TERMS

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. |

### Network Partition vs Disaster

### CAP THEOREM

CAP Theorem (Consistency, Availability, Partition Tolerance)

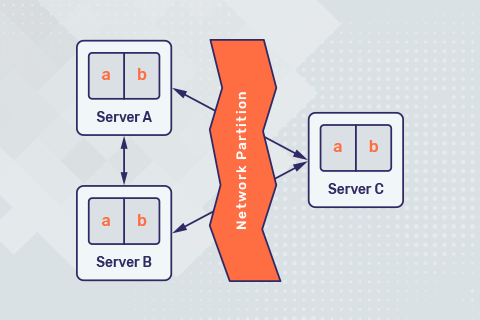
**Definition:**  
The CAP theorem, proposed by Eric Brewer, states that in a **distributed system**, you can achieve only **two** out of the following **three** guarantees at any time:

* **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 |

### 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)

* Ensures that a transaction is either **fully completed** or **fully rolled back** if any part of it fails.
* 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.
* In SQL, use **BEGIN TRANSACTION** to start, **COMMIT** to save changes, **ROLLBACK** to undo in case of failure
* 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 |

### 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**