BIG DATA ANALYTICS (CS-431)

Dr. Sriparna Saha
Associate Professor

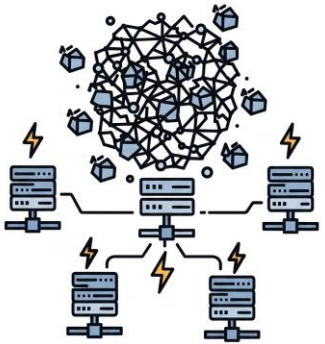
Website: https://www.iitp.ac.in/~sriparna/
Google Scholar: https://scholar.google.co.in/citations?user=Fj7jA_AAAAAJ&hl=en
Research Lab: SS_Lab
Core Research AREA: NLP, GenAI, LLMs, VLMs, Multimodality, Meta-Learning, Health Care, FinTech, Conversational Agents

TAs: Sarmistha Das, Nitish Kumar, Divyanshu Singh, Aditya Bhagat, Harsh Raj

BIG DATA STORAGE TECHNOLOGY

Why Distributed Systems for Big Data?

Modern applications generate data at a scale that a single server cannot handle. Big Data storage relies on **distributed systems** a network of interconnected computers to achieve:



- **Horizontal Scalability:** Instead of buying a more powerful server (vertical scaling), you can add more commodity servers to the cluster to handle increased load. This is far more cost-effective and flexible.
- **High Availability & Fault Tolerance:** If one node (server) in the system fails, the others can continue operating, ensuring the application remains available to users.

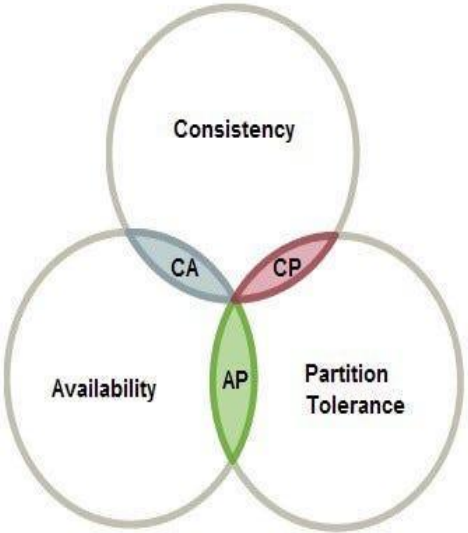
The Core Challenge: Data Consistency

In a distributed system, you have multiple copies of the same data on different machines. This raises a critical question:

If you write data to Node A, and then immediately read it from Node B, are you guaranteed to see the new data?

Answering this question is the central challenge of distributed data design. The **CAP Theorem** provides a foundational framework for thinking about this problem.

The CAP Theorem: An Introduction



CAP Theorem is a fundamental principle for distributed systems.

It states that it is **impossible** for a distributed data store to

simultaneously provide more than **two** of following three guarantees

- **Consistency (C):** Every read operation gets the most recent data or an error.
- **Availability (A):** Every request receives a non-error response,

  without the guarantee it's the most recent data.

- **Partition Tolerance (P):** The system continues to function despite network failures that split the system into multiple partitions.

'C': Consistency

Consistency (in the CAP context) means **strong consistency** or **linearizability**. It guarantees that every operation appears to have taken place instantaneously. All nodes in the system have the same view of the data at the same time.

- **Analogy:** Think of a bank account balance. If you deposit $100 at an ATM, a consistent system ensures that a simultaneous query from the mobile app will reflect that new balance immediately. There is only one "truth" for the balance at any given moment.

A': Availability

Availability means the system is always ready to accept requests. For every request made, a client will get a response, even if some nodes are down.

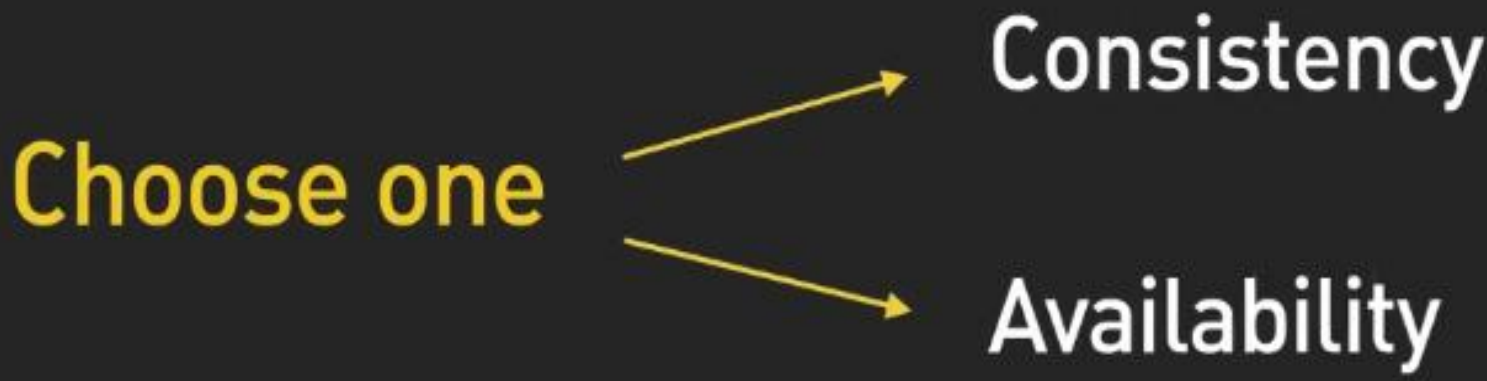
- **Important Nuance:** This response isn't guaranteed to be the most recent version of the data. The system prioritizes responding over ensuring the data is up-to-date.
- **Analogy:** When you "like" a post on a social media platform, the system immediately registers your like and updates the counter on your screen. It remains "available" to you, even if it takes a few seconds for that "like" to be visible to all your friends globally.

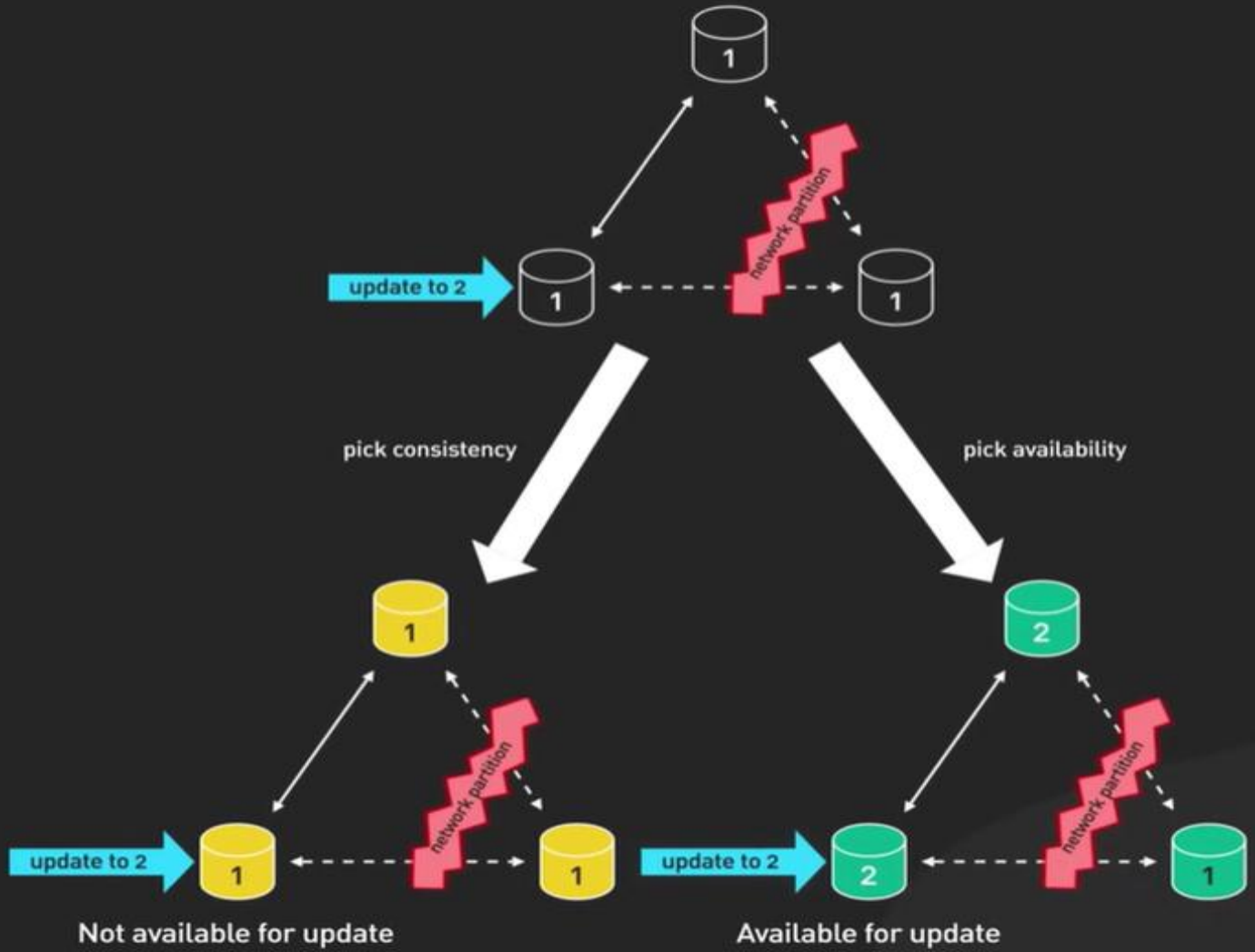
'P': Partition Tolerance

Partition Tolerance is the ability of the system to continue operating even when there is a communication break (a "network partition") between nodes.

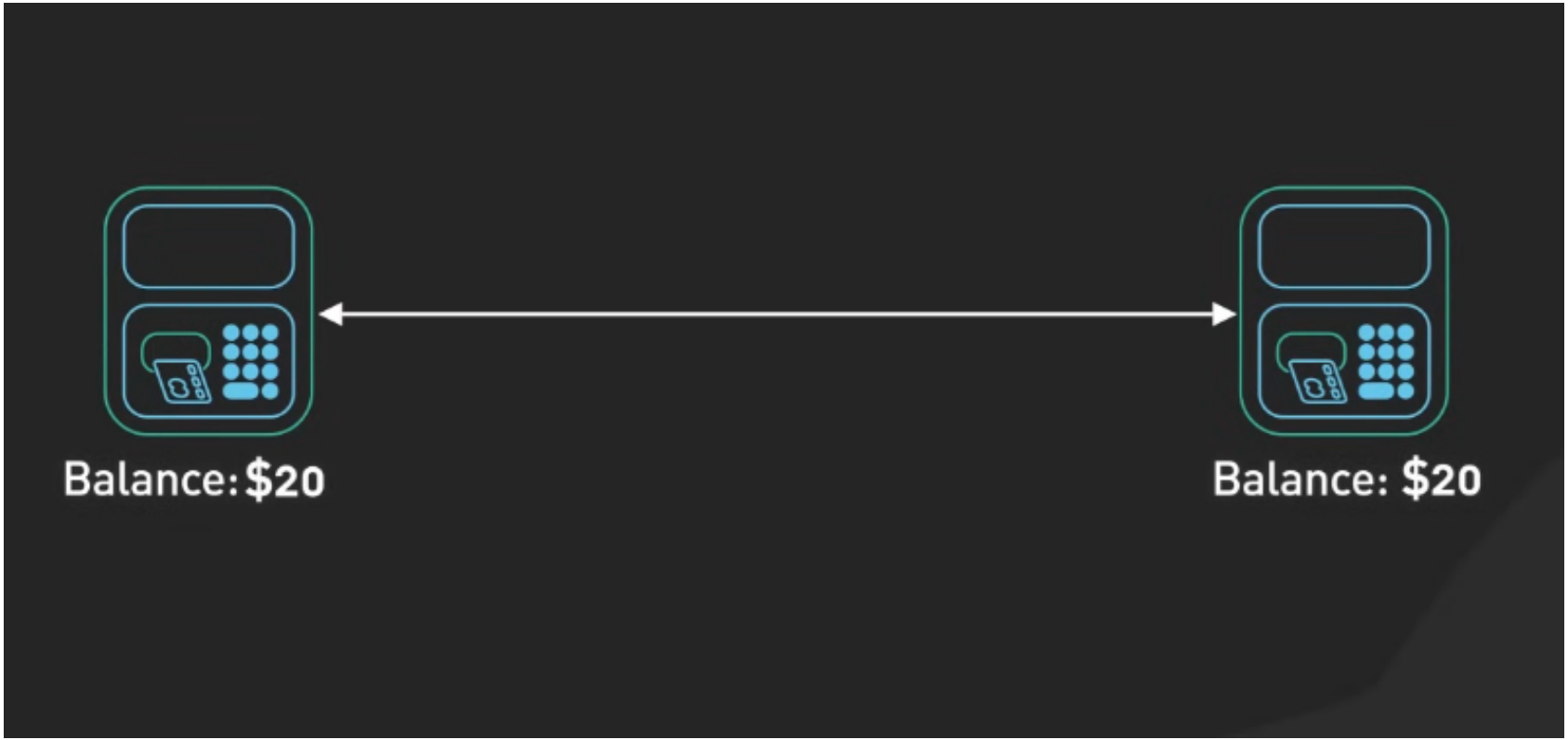
- **Why It's Non-Negotiable:** In any large-scale system that communicates over a network (like the internet or within a data center), failures are inevitable. Routers can fail, cables can be cut. You must design for failure.
- Therefore, in modern distributed systems, **Partition Tolerance (P) is a mandatory requirement.** The real choice is not *if* you'll have P, but what you'll sacrifice when a partition occurs.

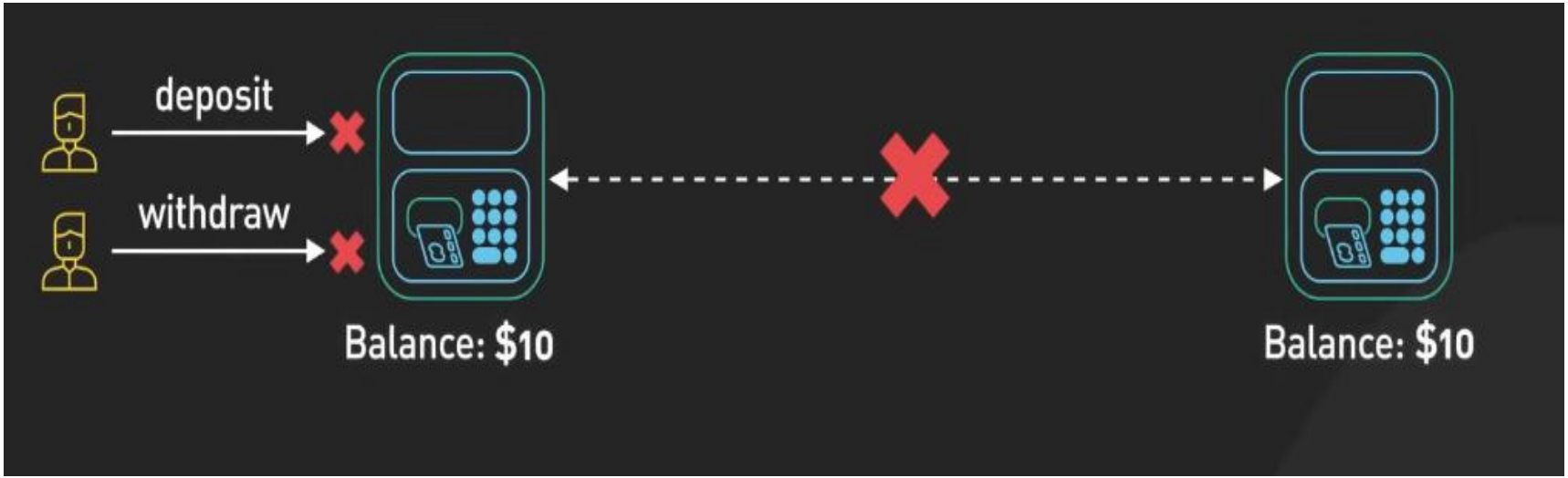
If partition happened:




1
update to 2
1
network partition
1
pick consistency
pick availability
1
2
update to 2
1
network partition
1
update to 2
2
network partition
1
Not available for update
Available for update

Example


Balance: $20
Balance: $20



In Case of Consistency

In Case of Availability

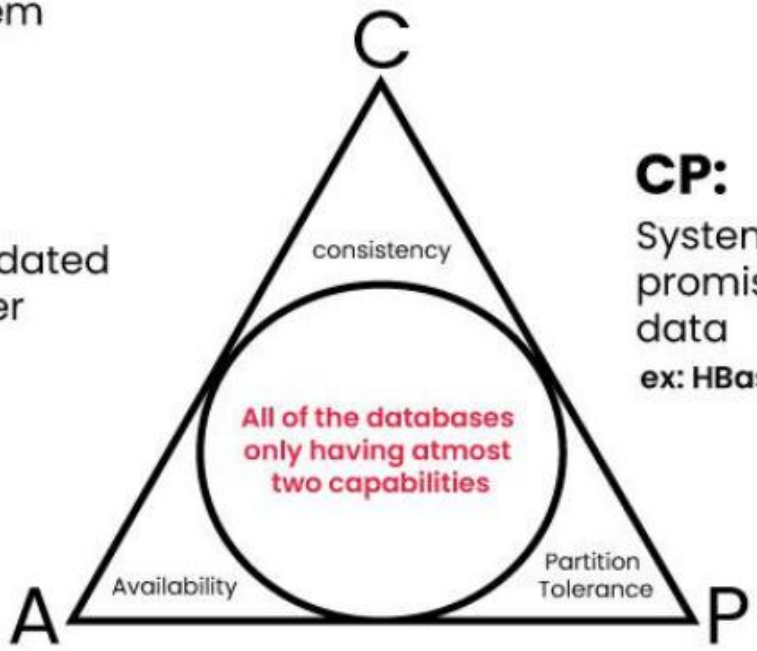
The CAP Theorem' s Inevitable Trade-Off

Since we must assume network partitions (P) will happen, the theorem forces a difficult choice when a partition occurs:

- **Do you sacrifice Consistency for Availability (AP)?** When nodes can't communicate, do you allow them to operate independently? This keeps the service online, but their data might diverge, becoming inconsistent.
- **Do you sacrifice Availability for Consistency (CP)?** When nodes can't communicate, do you shut down the affected part of the system to prevent inconsistent data from being written or read? This ensures data is correct, but the service becomes unavailable.

Trade-off in the CAP Theorem

CA:
system respond last updated data and promise higher availability
ex:RDBMS, PostgreSQL, etc



CP:
System can be distributed and promise to respond last updated data
ex: HBase, MongoDB, Redis

AP: System can be ditributed and promise to has high availability
ex: Cassandra, CouchDB, Riak, Voldemort, DynamoDB, etc

Common CAP Theorem Misconceptior

Cap Theorem

It's important to clarify what the CAP Theorem does and doesn't say.

- **Myth**: "You have to pick two out of three."
  - **Reality**: Because partition tolerance (P) is a given in any distributed system, the choice is almost always between consistency (C) and availability (A) *during a partition*.
- **Myth**: "Systems are either CP or AP."
  - **Reality**: Many systems are configurable. An administrator or developer can often tune the system's behavior, choosing different trade-offs for different operations.
- **Myth**: "The choice is binary."
  - **Reality**: Consistency is a spectrum. Systems can offer many levels of consistency, from very strong to very weak, each with different performance characteristics.

CAP Theorem in the Real World

CP (Consistency/Partition Tolerance)

- **MongoDB**: By default, it prioritizes strong consistency. During a partition, if a master node is on the minority side, it becomes read-only, sacrificing write availability.
- **HBase**: Built on HDFS and ZooKeeper, it guarantees strong consistency for reads and writes.

AP (Availability/Partition Tolerance)

- **Cassandra**: Designed from the ground up for massive availability and scalability. It allows for tunable consistency but defaults to an AP model, never sacrificing a write.
- **DynamoDB**: A classic AP system that prioritizes availability and performance, offering eventually consistent reads by default.

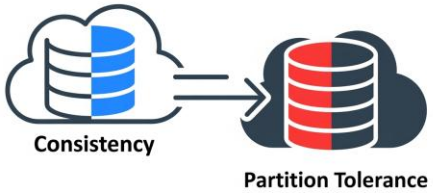
CA (Consistency/Availability)

- **PostgreSQL/Oracle**: In a single-instance setup, they are a perfect CA system. Distributed versions must make a CP or AP choice.

CP Systems (Consistency + Partition Tolerance)

When faced with a partition, a CP system chooses to preserve consistency.

- **Behavior:** The system will block or return an error for requests on the minority side of the partition to prevent data "drift." It waits until the partition is resolved and data is re-synchronized before becoming fully operational again.
- **Best For:** Systems where data accuracy is non-negotiable, such as banking systems, e-commerce stock management, and reservation systems.
- **Examples:** MongoDB, HBase, Redis.



AP Systems (Availability + Partition Tolerance)

When faced with a partition, an AP system chooses to preserve availability.

- **Behavior:** All nodes remain online and continue to serve requests using the best data they have locally. This can lead to clients seeing stale data. The system relies on mechanisms to resolve inconsistencies once the partition heals. This is the foundation of **eventual consistency**.
- **Best For:** Applications where uptime is critical and some temporary data inconsistency is acceptable.
- **Examples:** Amazon DynamoDB, Cassandra, CouchDB.

CA Systems (Consistency + Availability)

A CA system can only exist if you can guarantee there will **never** be a network partition.

- **Reality Check:** This is only possible in tightly controlled, single-node systems or specialized, fault-tolerant hardware clusters.
- **Relevance:** Traditional single-server relational databases (like a standard PostgreSQL or MySQL instance) operate as CA systems. They are consistent and available but have no concept of network partitions because all data resides on one machine. They are not considered distributed systems in the CAP sense.

The Consistency Spectrum

Consistency isn't just a simple choice between "strong" and "not strong." There is a wide spectrum of models that offer different trade-offs between performance and correctness.

- **Strong Consistency:** The most rigid and safest model.
- **Eventual Consistency:** The most flexible model, with many variations.
- **In-between Models:** Causal Consistency, Read-Your-Own-Writes Consistency, Session Consistency, etc.

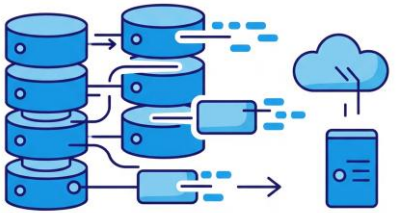
Choosing the right point on this spectrum is a key architectural decision.

Eventual Consistency

Eventual consistency is a foundational concept for AP systems. It provides a simple but powerful guarantee:

"If no new updates are made to a given data item, eventually, all accesses to that item will return the last updated value."

- **The "Eventually" Window:** This is the period of inconsistency. The system's goal is to minimize this window, but it accepts that it will exist.
- **Benefit:** This model allows for massive scalability and high availability, as writes can be accepted on any node without waiting for confirmation from others.



Eventual Consistency

Challenges of Eventual Consistency

While powerful, eventual consistency introduces complexity that developers must handle:

- **Stale Reads**: A user might read data that is outdated. For example, reading a product's price that was updated a few seconds ago on another server.
- **Write Conflicts**: Two users in different regions might update the same piece of data (e.g., a shared document) at the same time. When the partition heals, the system has a conflict. Who wins?
- **Increased Application Logic**: The application code, not just the database, must be aware of potential inconsistencies and be designed to handle them gracefully. This can make development more challenging.

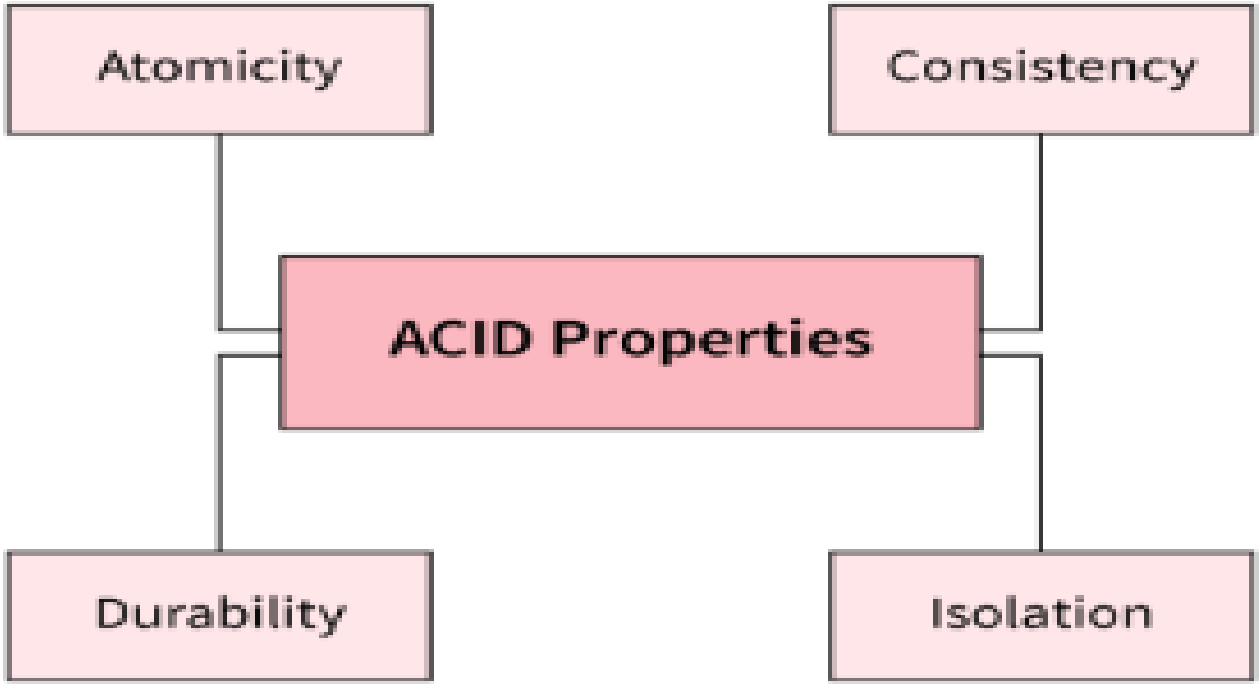
Consistency Trade-Offs in the Real World

The right model depends entirely on the business requirement for a specific piece of data.

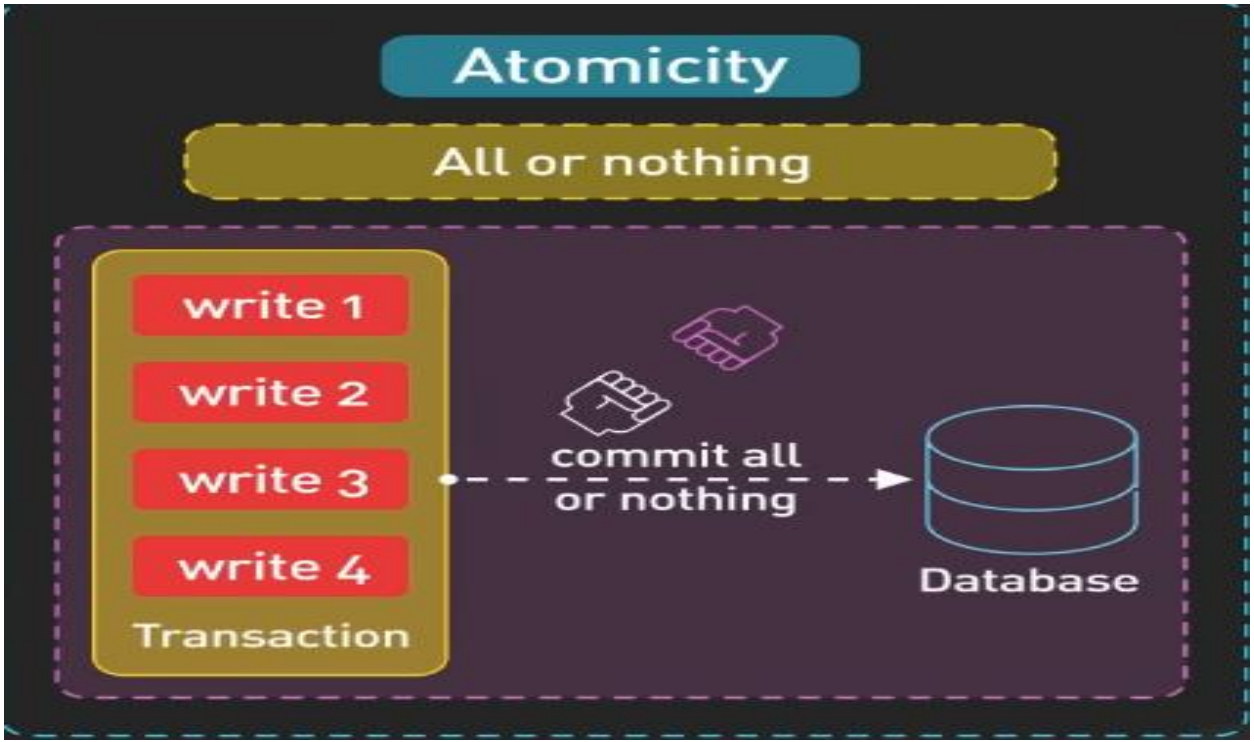
- **Strong Consistency (CP):**
  - **Use Case:** An e-commerce shopping cart checkout. You must have a perfectly consistent view of inventory and payment status.
  - **Risk of AP:** Selling an item that just went out of stock.
- **Eventual Consistency (AP):**
  - **Use Case:** The "likes" count on a social media post. It's acceptable if the count is slightly delayed or temporarily different for users in different regions.
  - **Risk of CP:** The "like" button becomes unavailable during a network issue, leading to poor user experience.

ACID and BASE Design Philosophies

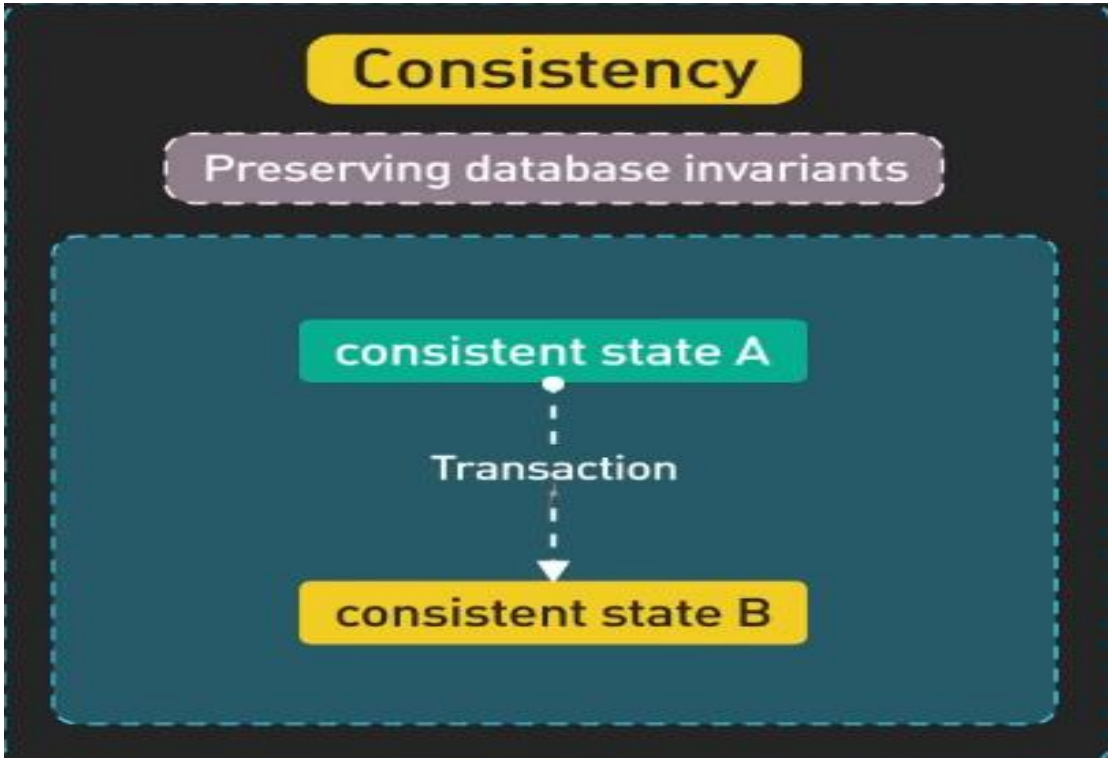
ACID Model


Atomicity
Consistency
ACID Properties
Durability
Isolation

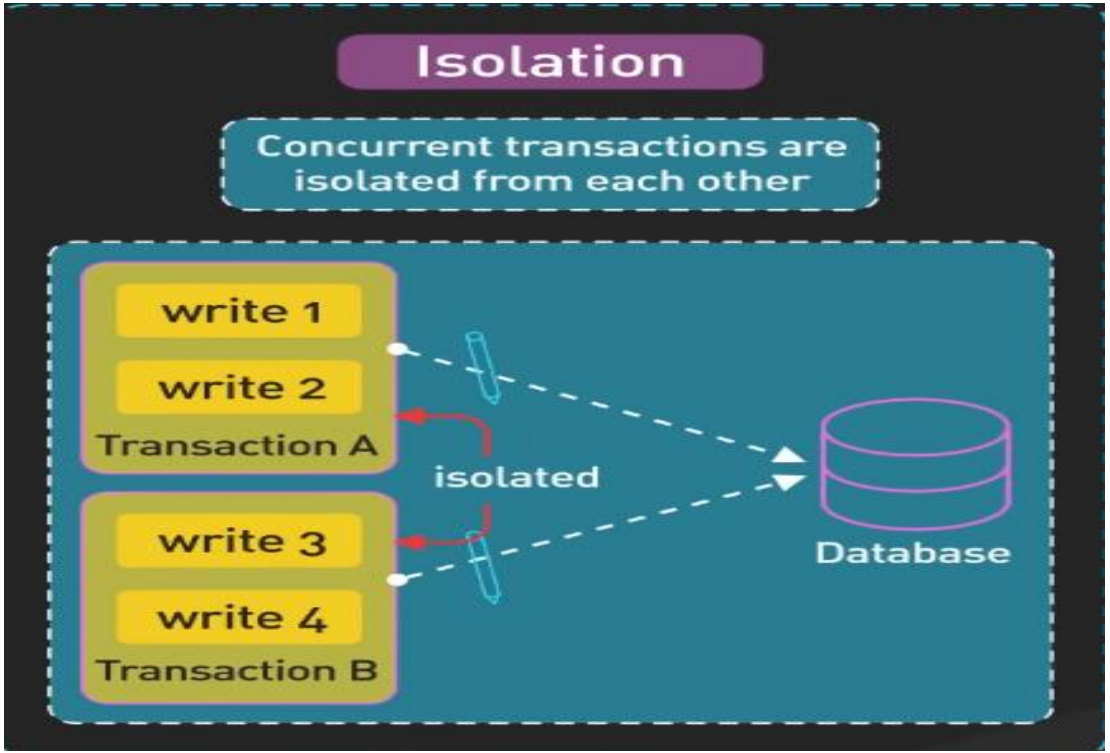
Atomicity


Atomicity
All or nothing
write 1
write 2
write 3
write 4
Transaction
commit all
or nothing
Database

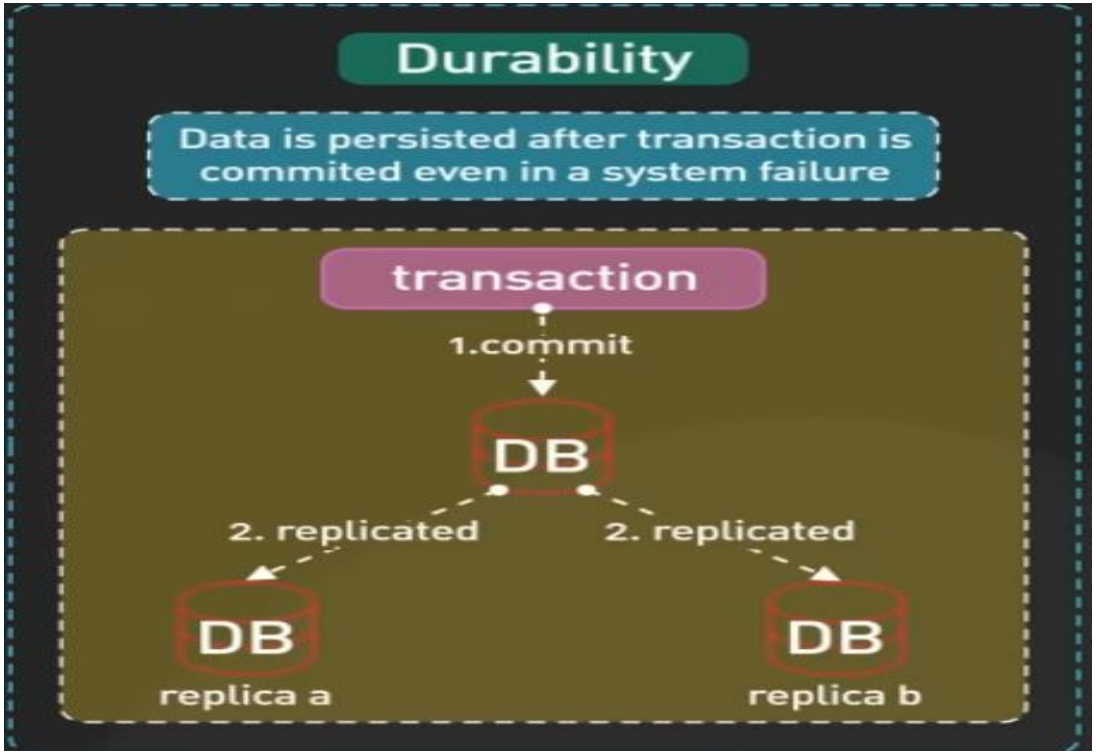
Consistency



Isolation


Isolation
Concurrent transactions are isolated from each other
write 1
write 2
Transaction A
isolated
write 3
write 4
Transaction B
Database

Durability

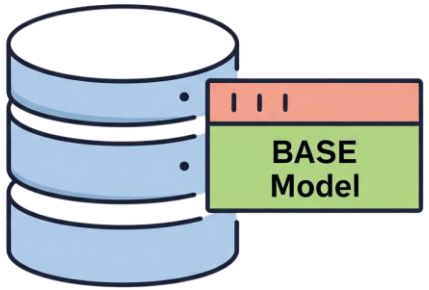

Durability
Data is persisted after transaction is commited even in a system failure
transaction
1.commit
DB
2. replicated
2. replicated
DB
replica a
DB
replica b

ACID Model

Common in relational databases (SQL), ACID provides a set of transactional guarantees.

- **Atomicity:** "All or nothing." A transaction (e.g., transferring money) either completes fully or fails entirely, leaving the database unchanged. It never ends up in a partial state.
- **Consistency:** "The data is always valid." Any transaction will bring the database from one valid state to another, enforcing all rules and constraints.
- **Isolation:** "Transactions don't interfere." Concurrent transactions produce the same result as if they were executed one after another.
- **Durability:** "Once it's saved, it stays saved." After a transaction is committed, it is permanently stored and will survive system crashes or power outages.

BASE Model



Common in many NoSQL databases, BASE offers a different set of promises for a distributed world.

- **Basically Available:** The system prioritizes availability, as described in the CAP theorem. It will always return a response.
- **Soft State:** The state of the system may change over time, even without new writes, as consistency propagates through the nodes. The data is not rigid.
- **Eventually Consistent:** As explained before, the system will eventually converge on a consistent state once writes stop. It doesn't happen immediately, but it will happen.

ACID vs. BASE: A Detailed Comparison

Feature	ACID (e.g., PostgreSQL)	BASE (e.g., Cassandra)
Focus	Strong Consistency	High Availability
Approach	Pessimistic (locks data to ensure correctness)	Optimistic (assumes conflicts are rare)
Data Model	Typically structured (Schema-on-write)	Typically unstructured/flexible (Schema-on-read)
Scalability	Vertical (bigger server)	Horizontal (more servers)
Use Case	Financial transactions, systems of record	Social media feeds, IoT data, analytics

ACID vs. BASE: The Scalability Dimension

A core difference between the philosophies is *how* they handle increased load.

- **ACID / Relational Databases: Vertical Scaling ("Scale-Up")**
  - **Method**: When the database is overloaded, you replace it with a single, more powerful (and much more expensive) server with more CPU, RAM, and faster storage.
  - **Limitation**: There is a physical and financial limit to how big a single machine can get.
  - **Analogy**: Swapping your car's engine for a bigger one.
- **BASE / NoSQL Databases: Horizontal Scaling ("Scale-Out")**
  - **Method**: When the database is overloaded, you add more cheap, commodity servers to the cluster. Load is distributed across the new servers.
  - **Advantage**: Theoretically limitless scalability.
  - **Analogy**: Adding more cars to your fleet.

Practical Scenarios for ACID

Choose an ACID-compliant system when the cost of inconsistency is extremely high.

- **Financial Ledgers:** Every debit must have a corresponding credit. Atomicity and consistency are paramount.
- **Airline Reservation Systems:** You cannot sell the same seat twice. Isolation ensures that two people trying to book the last seat don't both succeed.
- **HR and Payroll Systems:** Employee salary and status information must be accurate and durable.

Practical Scenarios for BASE

Choose a BASE system when the cost of unavailability is extremely high.

- **Social Media Feeds:** A user would rather see a slightly out-of-date feed than an error message. High availability for reading and writing is key.
- **IoT Sensor Data:** A system collecting data from millions of sensors must be able to ingest writes constantly. It's acceptable if analytics queries run on slightly delayed data.
- **User Session Data:** Storing user preferences or shopping cart items (before checkout) requires high availability but can tolerate minor delays in propagation.

NewSQL: The Best of Both Worlds?

A new category of databases, called **NewSQL**, aims to bridge the gap between traditional SQL and NoSQL.

- **Goal**: To provide the horizontal scalability and high availability of NoSQL systems (like BASE) while retaining the strong consistency and ACID guarantees of traditional relational databases.
- **How**: They achieve this through innovative architectures, often using consensus algorithms like Paxos or Raft to ensure all nodes agree on the state of the data before committing a transaction.
- **Examples**: CockroachDB, TiDB, Google Spanner. They represent the cutting edge of distributed database technology.