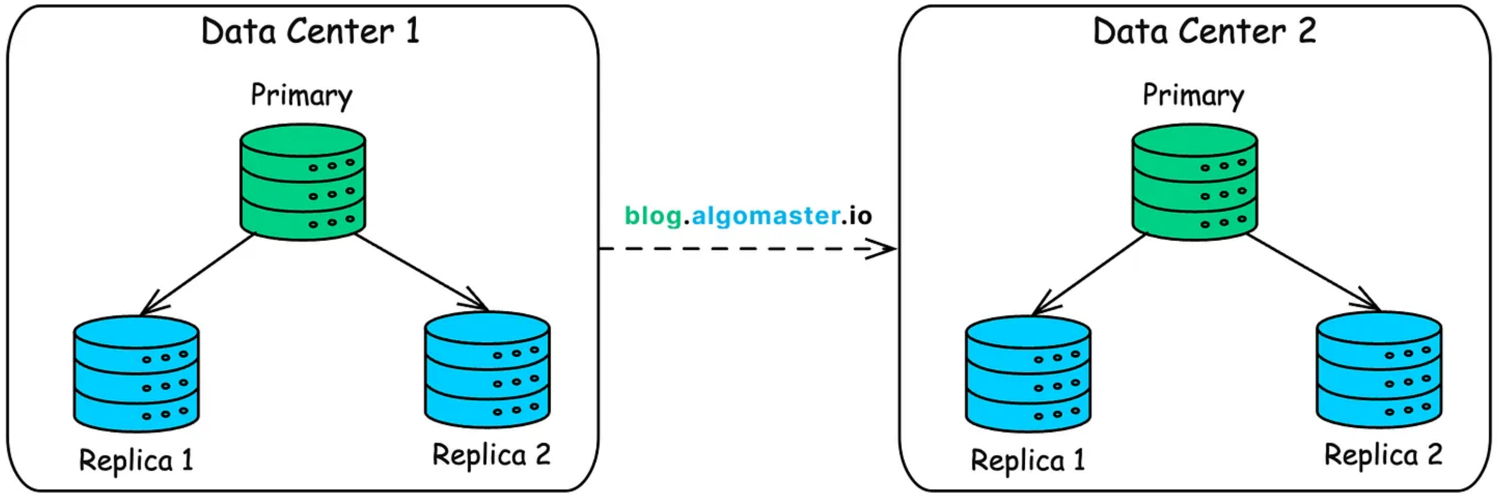
Strong vs. Eventual Consistency

ASHISH PRATAP SINGH
JUN 03, 2025

In today's distributed systems, data is almost never stored in a single place. It's replicated across **multiple servers**, often spread across **data centers around the world** to ensure high availability, fault tolerance, and performance.



This global distribution enables scalability but it comes with a critical challenge:

> **How do we ensure every user and system component sees a consistent and accurate view of the data, especially when updates are happening all the time?**

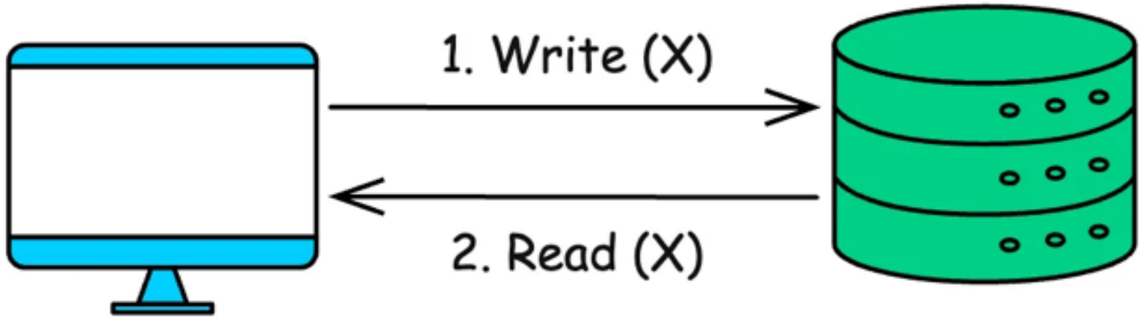
This is where **consistency models** come into play. They define the rules for how and when changes to data become visible across the system.

Two of the most widely discussed models are:

- **Strong Consistency** – where everyone sees the latest value, always.
- **Eventual Consistency** – where updates eventually propagate, but temporary inconsistencies are allowed.

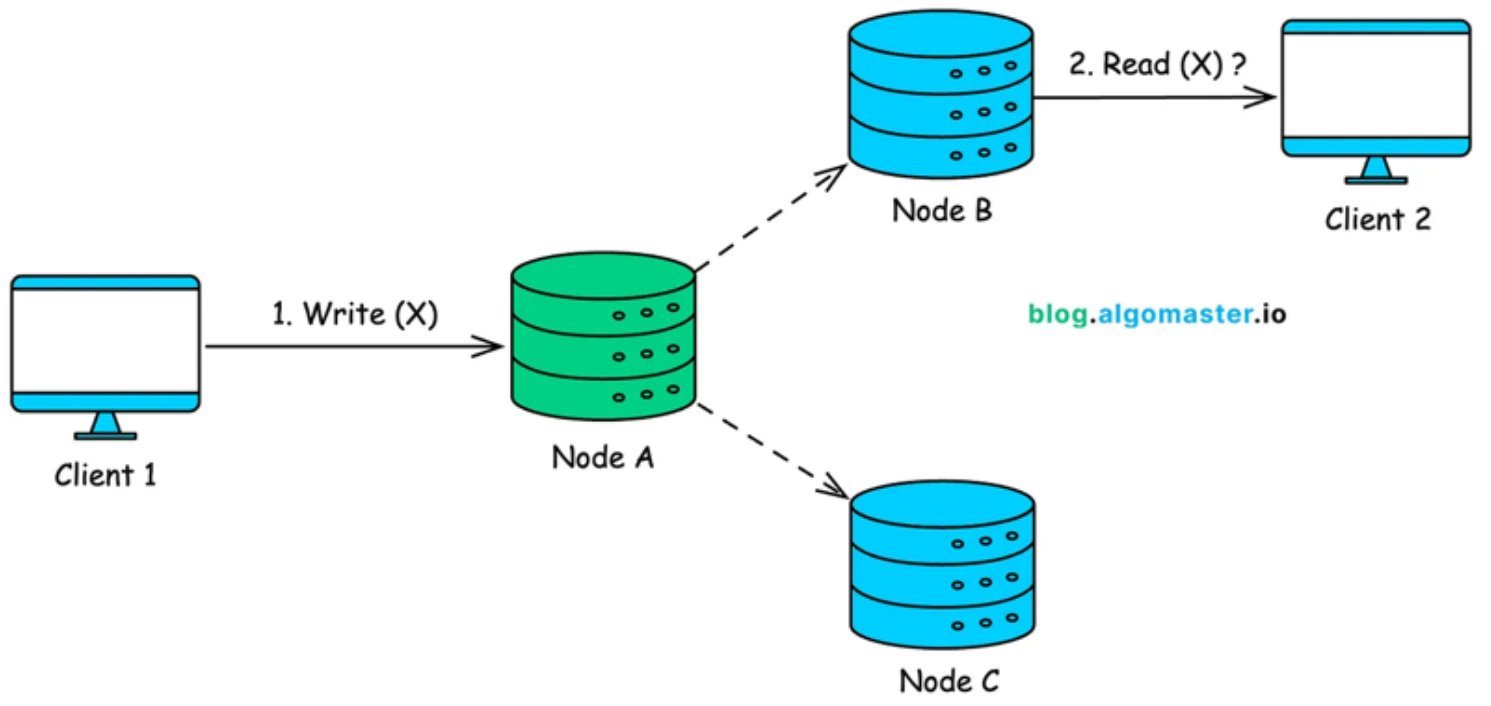
In this article, we'll break down these two models, explore their trade-offs, when to use each, and how to choose the right model depending on your application's goals.

What is Data Consistency in Distributed System?



In a **single-server system**, consistency is straightforward: when you write data, any subsequent read returns the most recent value. There's only one copy of the data.

But in **distributed systems**, where data is replicated across multiple nodes (for **availability**, **fault tolerance, or low latency**), things become more complex.



Imagine you write an update to **Node A**. That change takes time to **propagate** to **Node B** and **Node C**. During this **replication lag**, different nodes may temporarily hold **inconsistent versions** of the same data.

Consistency Models define the **rules and guarantees** a system provides about:

- **When** an update becomes visible
- **Where** it becomes visible (to which users or replicas)

- **In what order** operations are seen across the system

In essence, consistency models answer the question:

> **“If I write a piece of data, when and under what conditions will others (or even I) see that new value?”**

Strong Consistency

Strong consistency guarantees that once a write is successfully completed, **any read operation from any client or replica will reflect that write or a newer one**.

This means that the system always returns the most up-to-date value, regardless of where the read is performed.

It behaves as if there is a **single, globally synchronized copy of the data**, and all operations occur in a clear and consistent global order.

How It Works

To achieve strong consistency, the system performs **coordinated communication between replicas** before confirming a write:

1. A client sends a write request, such as updating a value in the database.
2. The primary node (e.g., Node A) propagates this write to a group of replicas.

3. Each replica (e.g., Node B, Node C) applies the write and sends an acknowledgment.
4. Only after receiving acknowledgments from all (or enough) replicas does the system confirm the write as complete to the client.
5. From that point onward, any read will return the updated value.

This behavior is made possible through **consensus algorithms** that ensure all replicas agree on the order of operations. Common protocols used include Paxos and Raft.

Pros

- **Simpler Application Logic**: You don't need to worry about stale data or implementing custom conflict resolution.
- **Predictable Behavior**: Easy to reason about; data reads always reflect the latest confirmed writes.
- **Data Integrity**: Ensures the highest level of data integrity and reliability.

Cons

- **Latency Overhead**: Writes (and sometimes reads) may be slower because they require coordination between nodes, often across regions.

- **Reduced Availability**: During network partitions or node failures, the system may reject requests to preserve consistency (as per the **CAP theorem**).
- **Complex Infrastructure**: Implementing strong consistency at scale requires sophisticated protocols and distributed coordination mechanisms.

When to Use Strong Consistency

Strong consistency is the right choice when your application needs **immediate correctness and absolute accuracy**.

It is especially important when inconsistencies could result in lost data, incorrect decisions, or broken trust such as:

- **Banking and financial systems**: Balances and transactions must always be accurate.
- **Inventory management**: You can't afford to sell the same item to two people.
- **Distributed locking**: Locks must be exclusive and reflect the latest state.
- **Unique ID generation**: Duplicate IDs must be prevented across replicas.

Eventual Consistency

Eventual consistency is a **weaker consistency model** that guarantees all replicas in a distributed system will **converge to the same value**, **eventually** as long as no new updates are made.

In simpler terms:

> If you stop writing to a piece of data, and wait long enough, **everyone will eventually see the same** (latest) **version of that data.**

There's **no guarantee about how soon** this convergence happens. In the meantime, different replicas may serve **different versions** of the data leading to temporary inconsistencies.

This model is often used in distributed systems where **high availability** and **partition tolerance** are prioritized over immediate consistency.

How It Works

1. A client sends a **write request** to a node, such as updating a data item on Node A.
2. Node A **acknowledges the write immediately**, without waiting for other replicas to be updated.
3. The updated value is **asynchronously propagated** to other replicas, such as Node

B and Node C.

4. While this propagation is in progress, **reads from other replicas** may return stale or outdated data.
5. Once all replicas have received and applied the update, the system is **fully consistent again**.

This replication process allows the system to **stay highly available and responsive**, even in the presence of network partitions or server failures.

A temporary inconsistency is acceptable in many scenarios, especially when the data being read is **not critical to business correctness** and **a slight delay in consistency does not harm the user experience**.

Example

Let’s say you're using a social media platform and you update your profile picture.

- You immediately see the new photo on your profile.
- A friend on the other side of the world might still see your old photo for a few seconds due to **replication lag**.
- After the system finishes syncing, everyone sees the updated photo.

This temporary inconsistency is acceptable because **the correctness of the system doesn't depend on everyone seeing the same thing at the exact same moment**.

Pros

- **Low Latency**: Writes don’t need to wait for global coordination; responses are fast.
- **High Availability**: Nodes can accept reads and writes independently even during network partitions.
- **Great for Scalability**: Ideal for large-scale, globally distributed systems that need to stay responsive under heavy load.

Cons

- **Temporary Staleness:** Clients may read outdated data until replicas are fully synchronized.
- **Increased Application Complexity**: Developers must handle inconsistency in code especially in read-after-write scenarios.
- **Potential for Conflicts**: If multiple replicas accept concurrent writes, you need a conflict resolution strategy (e.g., Last Write Wins, CRDTs).

When to Use Eventual Consistency

Eventual consistency is a good fit for **applications that require high availability and can tolerate temporary inconsistencies**. It is especially effective when systems are distributed globally or need to operate at massive scale.

Common use cases include:

- **Social media metrics:** Like counts, shares, and view counters can tolerate brief inconsistencies without impacting user experience.
- **Product view counts or analytics:** Tracking page visits, clicks, or user events can tolerate minor delays in consistency.
- **Recommendation systems:** Personalized suggestions such as products, movies, or articles do not require real-time accuracy and benefit from fast, large-scale reads.
- **DNS (Domain Name System):** DNS records are cached at multiple layers worldwide. Slight delays in propagation are acceptable and help maintain global availability.
- **Content delivery networks (CDNs):** Static assets like images, stylesheets, and scripts are served from edge locations. Updates propagate gradually to balance performance and consistency.
- **Shopping cart:** Eventual consistency is suitable when users add items from different devices. Conflict resolution strategies (such as merging or last-write-

wins) help maintain a coherent experience.

Variations and Client-Centric Models

It's important to note that "eventual consistency" is a broad term. There are stronger forms of eventual consistency that provide better guarantees:

- **Causal Consistency:** If operation A causally precedes operation B (e.g., B reads a value written by A), then all processes see A before B. Operations that are not causally related can be seen in different orders.
    - **Example:** In a comment thread, replies should appear after the comment they respond to even if the system is eventually consistent overall.
- **Read-Your-Writes Consistency:** After a client performs a write, any subsequent reads by that same client will always reflect the write (or a newer version). Other clients may still see stale data.
    - **Example:** You update your profile bio. When you refresh the page, your new bio appears immediately even if it takes a few seconds for others to see it.
- **Monotonic Reads Consistency:** If a client reads a value, any future reads by the same client will return the same or a newer value. The client will never see an

older version of the data.

    - **Example:** You see your post has 10 likes. After refreshing, you might see 10 or 11 likes but never fewer than 10.

- **Monotonic Writes Consistency:** Writes from the same client are executed in the order they were issued by that client.
    - **Example:** You post two comments: “Hello” followed by “World.” Other users will always see “Hello” before “World.”

Even in an **eventually consistent system**, applying **client-centric guarantees** helps preserve a sense of order, responsiveness, and trust for individual users.

Choosing the Right Consistency Model

There's no "best" consistency model. The right choice depends heavily on your application's specific requirements:

1. Data Criticality

How critical is it that all users see the most up-to-date, correct data at all times?

- High (e.g., financial transactions, inventory management)
  → **Use strong consistency.**
- Lower (e.g., social likes, view counts, analytics)
  → **Eventual consistency is often sufficient.**

2. User Experience Expectations:

Will users notice or care about stale data? Can the UI manage it gracefully?

For systems where users expect immediate feedback, you can often use:

- **Optimistic UI** updates (e.g., show the change immediately, then sync)
- **Loading indicators** (e.g., "Syncing..." or "Updating...")

Strong consistency reduces confusion, but **eventual consistency** can be acceptable with the right UX design.

3. Performance (Latency) Requirements

How important is low latency for reads and writes?

- **Eventual consistency** often delivers **faster response times**, since writes don't block on global coordination.

- **Strong consistency** adds overhead especially in geo-distributed systems.

4. Availability Requirements

Can the system tolerate downtime or errors during network partitions?

- **Strong consistency** may sacrifice **availability** to preserve correctness (as per the CAP theorem).
- **Eventual consistency** allows systems to **remain available** even during partial failures or network splits.

5. Scalability Needs

Does the system need to support massive scale across regions or data centers?

- **Eventually consistent systems** scale more easily especially for **read-heavy workloads.**
- **Strong consistency** may limit scaling options due to coordination overhead.

6. Development Complexity

How much complexity are you willing to handle at the application layer?

- **Strong consistency** simplifies application logic. Developers can assume they're always reading the latest state.

- **Eventual consistency** requires handling: stale data, idempotent operations and conflict resolution.

6. Conflict Resolution

What happens if two users write conflicting data at the same time?

In **eventually consistent systems**, concurrent writes to different replicas must be reconciled when replicas sync.

Common strategies:

- Last Write Wins (LWW)
- CRDTs (Conflict-Free Replicated Data Types)
- Custom merge logic based on application semantics

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more such content every week.

If you have any questions or suggestions, leave a comment.

This post is public so feel free to share it.

P.S. If you're enjoying this newsletter and want to get even more value, consider becoming a paid subscriber.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

There are group discounts, gift options, and referral bonuses available.

Checkout my Youtube channel for more in-depth content.

Follow me on LinkedIn and X to stay updated.

Checkout my GitHub repositories for free interview preparation resources.

I hope you have a lovely day!

See you soon,

Ashish

101 Likes • 7 Restacks

Discussion about this post

Comments Restacks

Write a comment...

DP Bhatt 5 Jun

Liked by Ashish Pratap Singh

Amazing post. Thank you

LIKE (3) REPLY SHARE