Data Replication

1. Why Distribute Data?

Distributing data across multiple machines or locations provides several advantages:

1. **Scalability & High Throughput**
    - As data volumes grow, a single machine cannot handle all read/write requests efficiently.
    - Distributing data across multiple machines allows the system to handle increased loads.
2. **Fault Tolerance & High Availability**
    - If one or more machines fail, the application continues functioning with minimal disruption.
    - Redundancy ensures no single point of failure.
3. **Low Latency (Faster Performance)**
    - When users are geographically dispersed, data replication allows them to access nearby servers instead of waiting for a distant central server.

Challenges of Distributed Data

1. **Consistency**: Ensuring all replicas are updated properly.
2. **Application Complexity**: Applications must handle reading and writing across multiple machines.

2. Vertical vs. Horizontal Scaling

Vertical Scaling (Scaling Up)

- **Shared Memory Architecture**: A single, centralized server with expandable resources.
- **Shared Disk Architecture**: Multiple machines access a single storage system over a network.
- **Limitations**:
    - Expensive at large scales.
    - Limited by hardware constraints.

Horizontal Scaling (Scaling Out)

- **Shared Nothing Architecture**: Each node operates independently with its own CPU, memory, and storage.
- **Commodity Hardware**: Uses cheaper, distributed machines instead of a single powerful one.
- **Better for Distributed Applications**: Reduces contention and improves availability.

3. Data Replication vs. Data Partitioning

Replication	Partitioning
Copies of the same data exist on multiple nodes.	Data is divided into subsets and stored on different nodes.
Increases redundancy and fault tolerance.	Improves load balancing and query performance.
Ensures availability if a node fails.	Each partition contains only a portion of the dataset.

4. Strategies for Data Replication

Distributed databases typically adopt one of three strategies:

1. Single Leader Model

- **All writes go through a single leader.**
- **Leader propagates updates to followers.**
- Followers process instructions and apply changes.
- **Clients can read from either leader or followers.**

Pros: ✔ Ensures strong consistency when configured synchronously.
✔ Well-supported by relational (MySQL, PostgreSQL) and NoSQL (MongoDB) databases.

Cons: ✖ Leader failure disrupts writes.
✖ Replication lag between leader and followers.

2. Multiple Leader Model

- **Each node can act as a leader and process writes.**

- Leaders synchronize changes with each other.

Pros: ✔ Better write availability (no single point of failure).
✔ Works well for geographically distributed databases.

Cons: ✖ Risk of conflicting writes.
✖ Higher complexity in conflict resolution.

3. Leaderless Replication

- **No dedicated leader**—writes can go to any node.
- Uses **quorum-based** consistency (e.g., require writes to be acknowledged by a majority of nodes).

Pros: ✔ Highly available.
✔ No single failure point.

Cons: ✖ Risk of stale reads if nodes are inconsistent.
✖ Complex conflict resolution.

5. How Replication Works

Synchronous vs. Asynchronous Replication

- **Synchronous Replication**
    - Leader waits for acknowledgment from all followers before confirming the write.
    - Guarantees strong consistency.
    - Slower, as all nodes must respond before proceeding.
- **Asynchronous Replication**
    - Leader processes the write and sends updates to followers without waiting.
    - Faster but can cause **replication lag** (some followers may have outdated data).

Replication Lag

- **The time delay between a write on the leader and when followers update.**
- More severe in **asynchronous replication**.

Trade-offs: ✔ **Synchronous Replication**: Ensures strong consistency but slows performance.
✔ **Asynchronous Replication**: Increases availability but sacrifices consistency.

6. Handling Leader Failures

What happens when the leader node fails?

1. **Electing a New Leader**
   - Use a **consensus algorithm** (e.g., Raft, Paxos) to elect a new leader.
   - Some systems use a **controller node** to appoint the next leader.
2. **Handling Incomplete Writes**
   - If **asynchronous replication** is used, some writes may be lost.
   - Should the system **recover lost writes** or **discard them**?
   - **"Split Brain" Problem**: If the old leader recovers, two leaders may conflict.

7. Read Consistency in Replicated Databases

Since replication causes data propagation delays, different consistency models ensure proper reading order:

1. Read-After-Write Consistency

- Ensures a client sees their own writes immediately.
- Useful for applications like social media posts or user profile updates.

Implementation Methods:

1. Always read from the leader for recently modified data.
2. Temporarily switch to leader-based reads for recently updated data.

Challenge:
✖ Followers may be geographically closer, but forcing reads from the leader increases latency.

2. Monotonic Read Consistency

- Guarantees that **a user will never read an older version of data after seeing a newer version.**
- Prevents users from seeing "fluctuating" data.

Example:
If a user refreshes a dashboard, they should not see an older version after seeing a newer one.

3. Consistent Prefix Reads

- Ensures that **reads happen in the correct order.**
- Without it, users may see updates out of sequence.

Example:

- A banking app logs two transactions:
    - $100 Deposit → $50 Withdrawal
- Without **Consistent Prefix Reads**, a user might see the withdrawal first, making the account appear overdrawn.