1. Benefits of Distributing Data

- **Scalability / High Throughput**: Handle growing data or R/W load.
- **Fault Tolerance / High Availability**: The system keeps working despite machine failure.
- **Latency**: Global users get faster access.

2. Challenges in Distributed Data

- **Consistency**: Updates must sync across nodes.
- **Application Complexity**: Devs must manage distributed reads/writes.

3. Vertical Scaling

Shared Memory Architectures

- Centralized servers with some fault tolerance (e.g., hot-swappable parts)

Shared Disk Architectures

- Machines share disk via fast network
- Good for high-read workloads (e.g., Data Warehousing)
- Poor for high-write loads (locking contention)

4. Horizontal Scaling (Shared Nothing Architecture)

- Each node has its own CPU, memory, and disk.
- Commodity hardware across geographic locations.
- Communication happens over the network at the app layer.

5. Replication vs Partitioning

- **Replication**: Copies of the full dataset.
- **Partitioning**: Subsets of the dataset distributed across nodes.

6. Replication Strategies

- **Single Leader Model**: All writes go to the leader; followers replicate from it.
- **Multiple Leader Model**: Multiple nodes accept writes (conflict-prone).
- **Leaderless Model**: No central writer node.

Used in:

- Relational DBs: MySQL, Oracle, PostgreSQL, SQL Server
- NoSQL DBs: MongoDB, RethinkDB, Espresso
- Messaging systems: Kafka, RabbitMQ

7. How Replication Info Is Transmitted to Followers

- **Statement-based Replication**: Sends SQL statements like INSERT, UPDATE, DELETE to replicas. Simple but error-prone due to use of non-deterministic functions (e.g., `NOW()`), side effects from triggers, and concurrency issues.
- **Write-ahead Log (WAL)**: Sends a low-level byte log of changes. Requires leader and followers to use the same storage engine. Makes upgrades harder.
- **Logical (Row-based) Log**: Sends actual row changes (inserted, updated, deleted rows), decoupled from storage engine, and easier to parse.
- **Trigger-based Replication**: Uses database triggers to log changes into a separate table. Offers flexibility but increases complexity and risk of errors.

8. Synchronous vs Asynchronous Replication

- **Synchronous Replication**: Leader waits for confirmation from followers before committing a write. Ensures consistency but can slow down the system.
- **Asynchronous Replication**: Leader commits immediately without waiting for followers. Improves speed but risks temporary inconsistency.

9. Leader Failure Challenges

- Selecting a new leader: requires consensus strategy or controller node.
- Clients need to know where to redirect their writes.
- Asynchronous replication risks missing recent writes—how do we recover them?
- Preventing split-brain scenarios where two leaders accept conflicting writes.
- Tuning leader failure detection is difficult—timeouts must be chosen carefully.

10. Replication Lag

- The delay between a leader committing a write and a follower reflecting it.
- **Synchronous**: Lag makes writes slower and system more fragile with more followers.
- **Asynchronous**: Better availability but delayed consistency—this delay is known as the **inconsistency window**.

11. Read-After-Write (RAW) Consistency

- Users should see their own recent writes immediately (e.g., Reddit comment shows up for you right away).
- Other users seeing the update can wait a bit.

Strategies:

1. Always read recently modified data from the leader.
2. Temporarily switch to leader for reads right after a write (e.g., for one minute).

Trade-off: May have to route requests away from nearby followers to more distant leaders.

12. Monotonic Read Consistency

- Prevents a user from seeing older data after they've already seen newer data.
- Ensures a user’s experience progresses in logical order.

13. Consistent Prefix Reads

- Guarantees that users see writes in the order they were applied.
- Prevents out-of-order reads that can happen when different partitions replicate at different rates.