Main Themes and Key Concepts:

1. Benefits and Challenges of Distributed Data:

- **Benefits:**
  Distributing data provides several advantages:
  - **Scalability / High Throughput:** Distributing data enables systems to manage increased data volumes and read/write demands that exceed the capabilities of a single machine. This is essential when "data volume or read/write load grows beyond the capacity of a single machine."
  - **Fault Tolerance / High Availability:** Distributed systems ensure that applications remain operational even if some machines fail. This is crucial for situations where "the application must continue working even if one or more machines go down."
  - **Latency:** Performance improves for geographically dispersed users by making data more accessible to them. This is vital when providing "fast performance" for users "in different parts of the world."
- **Challenges:**
  Distributing data comes with challenges:
  - **Consistency:** Ensuring consistency across multiple replicas is challenging, as "updates must be propagated across the network."
  - **Application Complexity:** In a distributed system, managing data reads and writes often falls on the application layer.

2. Scaling Architectures:

- The document compares various scaling architectures:
  - **Vertical Scaling (Shared Memory & Shared Disk):** These offer some fault tolerance (hot-swappable components in shared memory) or are suitable for high-read workloads (shared disk for data warehouses), but face limitations in scalability due to their centralized nature and issues like contention and locking overhead. Vertical scaling can also be costly, as shown by an example of a "$78,000/month" AWS instance.
  - **Horizontal Scaling (Shared Nothing):** This architecture, where "each node has its own CPU, memory, and disk," is ideal for geographic distribution and uses "commodity hardware." Coordination is managed at the application layer.

3. Replication vs. Partitioning:

- The document distinguishes between replication and partitioning:
  - **Replication:** This involves keeping identical copies ("replicates have the same data as the main dataset") across multiple nodes.
  - **Partitioning:** In this case, the dataset is divided into subsets ("partitions have a subset of the data") and distributed across nodes.

4. Common Replication Strategies:

- Distributed databases typically employ one of the following strategies:
    - **Single Leader Model:** In this model, all write operations are directed to a single leader node, which then propagates changes to follower nodes. Clients can read from either the leader or the followers. This is described as a "very common strategy" used by relational databases like "MySQL, Oracle, SQL Server, PostgreSQL" and NoSQL databases such as "MongoDB, RethinkDB (for real-time web apps), Espresso (LinkedIn)," as well as messaging brokers like "Kafka, RabbitMQ."
    - **Multiple Leader Model:** This allows writes to be accepted by multiple leader nodes, which must coordinate to resolve conflicts (this is implied but not detailed in the notes).
    - **Leaderless Model:** In this model, any replica can accept writes, with mechanisms like quorums ensuring consistency (this is also implied but not detailed).

5. Methods for Propagating Replication Information:

- Several methods are available for propagating write operations from the leader to followers:
    - **Statement-based Replication:** This method sends SQL commands (INSERT, UPDATE, DELETE) to replicas. While simple, it is "error-prone due to non-deterministic functions like `now()`, trigger side-effects, and challenges with concurrent transactions."
    - **Write-ahead Log (WAL):** This transmits a byte-level log of every change. However, "both the leader and followers must implement the same storage engine, making upgrades challenging."
    - **Logical (Row-based) Log:** For relational databases, this method logs "inserted rows, modified rows (before and after), and deleted rows." It is "decoupled from the storage engine, making it easier to parse."
    - **Trigger-based Replication:** Changes are logged in a separate table when a trigger fires. It is "flexible for application-specific replication but can also be more error-prone."

6. Synchronous vs. Asynchronous Replication:

- The trade-offs between consistency guarantees and performance are explored:
    - **Synchronous Replication:** In this approach, the leader waits for acknowledgment from one or more followers before confirming the write to the client. The document notes that "the leader waits for a response from the follower."
    - **Asynchronous Replication:** Here, the leader does not wait for acknowledgment from the followers. This improves availability and reduces write latency but risks data inconsistency.

7. Leader Failure Challenges:

- If the leader node fails, several challenges arise:
    - **Leader Election:** A new leader must be elected, potentially using a "consensus strategy (e.g., based on who has the most updates?) or appointing a new leader via a controller node."
    - **Client Reconfiguration:** Clients must be updated to send write operations to the new leader.
    - **Data Loss (in Asynchronous Replication):** If asynchronous replication is used, the new leader may not have received all writes from the failed leader. Decisions must be made on how to "recover the lost writes" or "discard" them.
    - **Split Brain:** Recovering the old leader may result in a "split brain" scenario, where two nodes simultaneously believe they are the leader and conflict arises.
    - **Leader Failure Detection:** Detecting a leader failure requires setting an "optimal timeout," which can be difficult to determine.

8. Replication Lag and Consistency:

- **Replication Lag:** This is the delay between a write on the leader and its reflection on the followers.
    - Synchronous replication can slow down writes and increase system brittleness with more followers.
    - Asynchronous replication maintains availability but may result in delayed or eventual consistency, referred to as the "inconsistency window."
- **Read-After-Write Consistency:** Ensures that after a write, a user's subsequent reads reflect the change:
    - Method 1: Always read from the leader for modifiable data.
    - Method 2: Dynamically switch to reading from the leader for "recently updated" data (e.g., within one minute of the last update).
- **Monotonic Read Consistency:** Prevents users from reading older data after having seen newer data in successive reads from different followers. It helps avoid "monotonic read anomalies," ensuring that when a user reads multiple times, they will not encounter outdated data.
- **Consistent Prefix Reads:** Guarantees that if a series of writes occurs in a specific order, any reader will observe them in the same order. This is important because "reading data out of order" can happen if different partitions replicate data at different rates. The guarantee ensures that "if a sequence of writes happens in a certain order, any reader will see them appear in the same order."