# Replicating Data

## Introduction to Data Replication

Data replication involves creating and maintaining multiple copies of the same data across different locations. It's a key component of most modern distributed database systems.

## Benefits of Distributed Data Systems

The presentation outlines three main benefits of distributing data:

1. **Scalability/High Throughput**: When data volume or read/write operations exceed what a single machine can handle, distributing data across multiple machines becomes necessary. This is especially important for large-scale applications that serve millions of users simultaneously.

2. **Fault Tolerance/High Availability**: Distributed systems allow applications to continue functioning even if one or more machines fail. This resilience is critical for services that need to maintain continuous uptime (like banking systems, e-commerce platforms, etc.).

3. **Reduced Latency**: By placing data closer to users around the world, distributed systems can reduce the physical distance data must travel, significantly improving response times. This is why major tech companies have data centers on multiple continents.

## Challenges of Distributed Data

The presentation highlights two primary challenges:

1. **Consistency**: Updates must be propagated across the network to maintain consistency. This introduces complex synchronization problems and potential for data inconsistencies.

2. **Application Complexity**: The responsibility for properly reading and writing data in a distributed environment often falls to the application layer, increasing development complexity.

## Scaling Approaches

### Vertical Scaling

Vertical scaling refers to adding more resources (CPU, RAM, storage) to a single machine:

- **Shared Memory Architectures**: A centralized server with multiple CPUs accessing the same memory. While offering some fault tolerance through hot-swappable components, these systems remain centralized and have limited geographic distribution.

- **Shared Disk Architectures**: Multiple machines connected via a fast network to shared storage. This approach is limited by contention and locking overhead, making it more suitable for read-heavy workloads (like data warehouses) than write-heavy applications.

The presentation notes the high cost of vertical scaling, showing AWS EC2 pricing where high-end instances can cost over $78,000/month as of October 2024.

### Horizontal Scaling (Shared Nothing Architectures)

This approach involves adding more machines, each with its own resources:

- Each node has independent CPU, memory, and disk
- Nodes communicate via application layer protocols over conventional networks
- Can be geographically distributed
- Often uses commodity (lower-cost) hardware
- Much more cost-effective for scaling large systems

## Data Management Strategies

### Replication vs. Partitioning

The presentation distinguishes between two fundamental approaches:

1. **Replication**: Creating complete copies of data across multiple nodes. All replicas contain the same data as the main copy.

2. **Partitioning**: Dividing data into subsets across multiple nodes. Each partition contains only a portion of the complete dataset.

## Replication Strategies

Three common strategies for data replication:

1. **Single Leader Model**: One node (the leader) accepts all writes and distributes changes to followers. This is the most common approach.

2. **Multiple Leader Model**: Multiple nodes can accept writes, which are then synchronized with other nodes.

3. **Leaderless Model**: Any node can accept writes, with various mechanisms to resolve conflicts.

The single leader model is particularly widespread, used by:

- Relational databases: MySQL, Oracle, SQL Server, PostgreSQL
- NoSQL databases: MongoDB, RethinkDB, Espresso (LinkedIn)
- Messaging systems: Kafka, RabbitMQ

## How Replication Information is Transmitted

Four primary methods:

1. **Statement-based**: Sending SQL statements (INSERT, UPDATE, DELETE) to replicas. Simple but error-prone due to non-deterministic functions, trigger side-effects, and challenges with concurrent transactions.

2. **Write-ahead Log (WAL)**: Low-level byte-specific logs of all database changes. Efficient but requires the same storage engine across all nodes and complicates upgrades.

3. **Logical (row-based) Log**: Records of inserted, modified, and deleted rows. More flexible than WAL as it's decoupled from storage engine details.

4. **Trigger-based**: Changes logged via database triggers. Offers application-specific flexibility but is more prone to errors.

## Synchronous vs. Asynchronous Replication

- **Synchronous**: Leader waits for confirmation from followers before acknowledging a write is complete. Provides stronger consistency but reduces performance and availability.

- **Asynchronous**: Leader doesn't wait for follower confirmation. Improves performance and availability but risks data loss if the leader fails before changes propagate.

# Handling Leader Failures

Leader failure creates several challenges:

1. **Selecting a new leader**: May require consensus algorithms or external controller nodes.
2. **Reconfiguring clients**: Redirecting write traffic to the new leader.
3. **Handling incomplete replication**: If using asynchronous replication, determining how to handle writes that didn't reach all followers.
4. **Split brain problems**: Preventing scenarios where multiple nodes believe they are the leader.
5. **Failure detection**: Setting appropriate timeouts to accurately detect when a leader has failed.

# Replication Lag and Consistency Models

**Replication lag** refers to delays between when a write occurs on the leader and when it's reflected on followers.

Consistency models discussed include:

1. **Read-after-Write Consistency**: Ensuring users see their own writes immediately after making them. Implementation approaches include:
    - Always reading user-modifiable data from the leader
    - Dynamically routing requests to the leader for recently modified data
2. **Monotonic Read Consistency**: Ensuring a user never sees data going "backward in time" when making multiple reads. This prevents the scenario where a user first sees newer data then older data due to reading from different replicas.

3. **Consistent Prefix Reads**: Guaranteeing that related writes appear to users in the same order they were originally written. This addresses issues that can arise when different data partitions replicate at different rates.

# Implementation Tradeoffs

The presentation highlights the tension between:

1. Using followers to improve latency by being geographically close to users
2. The need to route certain reads to potentially distant leaders to maintain consistency guarantees