Replicating Data

Distributing Data – Benefits:

- **Scalability & High Throughput**:
  - Distributed databases handle significantly more data and increased read/write loads than a single machine could manage effectively.

- **Fault Tolerance & High Availability**:
  - Ensures continuous operation even if one or more nodes fail, thus increasing the reliability of the overall system.

- **Latency Reduction**:
  - Data replication allows for geographically distributed nodes, enabling users worldwide to experience low-latency performance by accessing data from servers closest to them.

Distributed Data – Challenges:

- **Consistency**:
  - Data updates must propagate across multiple nodes in a network. Ensuring all nodes remain synchronized in a distributed environment is complex.

- **Application Complexity**:
  - Distributed systems require applications to handle intricacies of managing read and write operations across multiple nodes, often transferring this complexity to the application logic itself.

Vertical Scaling – Shared Memory Architectures:

- Features a centralized server location.
- Limited fault tolerance via "hot-swappable" components allowing parts to be replaced without downtime.
- All CPUs have direct access to a shared memory pool, simplifying data management.

Vertical Scaling – Shared Disk Architectures:

- Multiple machines are interconnected through a fast network, accessing a

shared disk storage.
- While enabling good read scalability, it encounters write scalability limits due to contention and locking mechanisms.
- Suitable for data warehouse systems that primarily read rather than write.

AWS EC2 Pricing – Oct 2024 (Example):

- Illustrates the high costs associated with vertical scaling approaches, emphasizing potential monthly costs of over $78,000 for very powerful instance types.
- Serves as motivation for considering alternative, more cost-effective scaling methods.

Horizontal Scaling – Shared Nothing Architectures:

- Each node independently manages its CPU, memory, and storage without shared resources.
- Application-level coordination manages communication and data synchronization between nodes.
- Geographical distribution is common, leveraging commodity hardware to keep costs manageable and scalability high.

Data Replication vs. Partitioning:

- **Replication** involves copying and maintaining database copies across multiple nodes, each replica maintaining an identical dataset.
- **Partitioning** (sharding) splits data into different subsets, each stored on separate nodes. Each node manages a unique subset of the data.

Replication:

- Essential for improving availability and fault tolerance. If one node goes offline, other replicas continue serving data seamlessly.

Common Replication Strategies:

- **Single Leader**:
  - One node handles all writes; replication streams updates to follower nodes.

- **Multiple Leader**:
  - Multiple nodes accept writes independently, potentially requiring complex synchronization.

- **Leaderless**:
  - No designated leader; every node can handle read and write operations, with consistency managed via consensus algorithms.

Leader-Based Replication:

- All client writes are directed to the leader node, which then replicates these updates to follower nodes.
- Followers process these instructions, maintaining synchronization.
- Clients perform read operations from either leader or followers, optimizing load distribution and response times.

Replication Methods:

- **Statement-based**: Replicates SQL commands (INSERT, UPDATE, DELETE).
- **Write-ahead Log (WAL)**: Logs every database change as low-level byte streams.
- **Logical (row-based)**: Replicates row-level changes explicitly.
- **Trigger-based**: Uses database triggers to initiate replication operations.

Synchronous vs Asynchronous Replication:

- **Synchronous**:
  - Leader waits for follower acknowledgment before completing the write, ensuring high consistency but sacrificing performance and latency.

- **Asynchronous**:
  - Leader completes write immediately without waiting for followers, prioritizing

speed and performance, but introducing potential lag ("inconsistency window").

Leader Failure Challenges:

- Requires mechanisms to quickly and reliably elect a new leader.
- Consensus strategies or controller nodes manage new leader selection.
- Complexity arises when handling data consistency and write losses, particularly with asynchronous replication.
- Risk of "split-brain" conditions, where multiple nodes mistakenly believe they're leaders.

Replication Lag:

- The time delay before writes on the leader propagate to followers.
- Increased lag with synchronous replication makes the system slower as followers increase.
- Asynchronous replication provides better performance but sacrifices immediate consistency, creating an "inconsistency window."

Read-after-Write Consistency:

- Ensures a user sees their updates immediately after committing changes (common requirement in user-facing applications).

Methods:
- **Always read from the leader** for modifiable data.
- **Dynamic reads**: Temporarily route reads to leader immediately after writes for consistency.

Challenges of Read-after-Write Consistency:

- Routing reads to distant leaders undermines geographic latency advantages originally gained from follower nodes.

Monotonic Read Consistency:

- Avoids anomalies where users see older data after previously viewing updated data.
- Ensures sequential consistency in multiple reads from distributed followers.

Consistent Prefix Reads:

- Guarantees that sequences of writes seen by users maintain the same order they were originally committed, despite replication delays across distributed systems.
- Important to maintain logical consistency from the user's perspective, even if full global consistency isn't achievable.