DS 4300

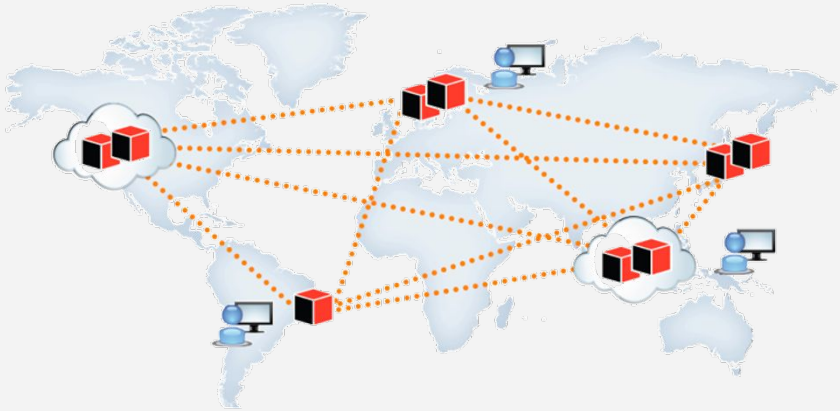
Replicating Data

Mark Fontenot, PhD
Northeastern University

Some material used with permission from Dr. Rachlin, with thanks!

Distributing Data - Benefits

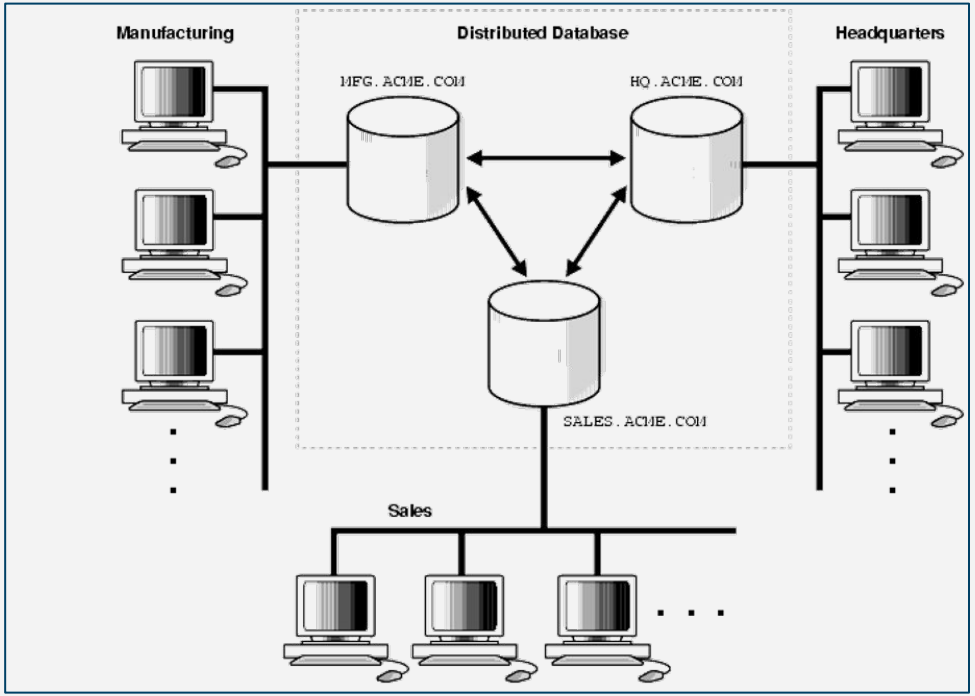
- **Scalability / High throughput**: Data volume or Read/Write load grows beyond the capacity of a single machine
- **Fault Tolerance / High Availability**: Your application needs to continue working even if one or more machines goes down.
- **Latency**: When you have users in different parts of the world you want to give them fast performance too



Distributed Data - Challenges

- **Consistency**: Updates must be propagated *across the network*.

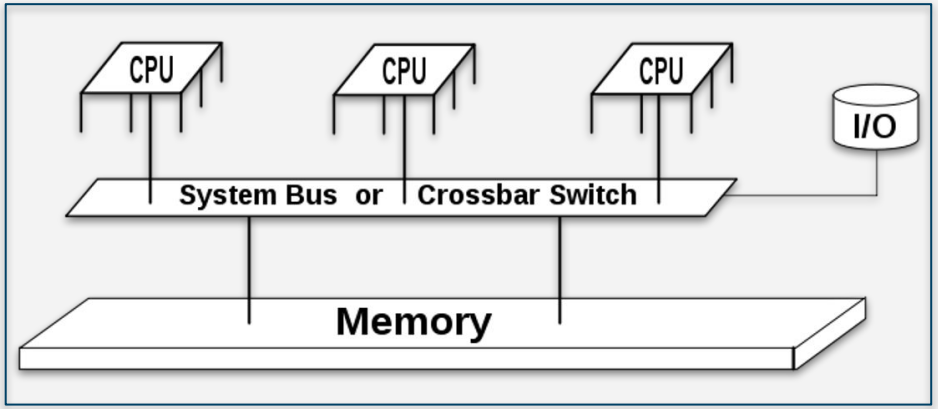
- **Application Complexity**: Responsibility for reading and writing data in a distributed environment often falls to the application.



Vertical Scaling - Shared Memory Architectures

- Geographically Centralized server

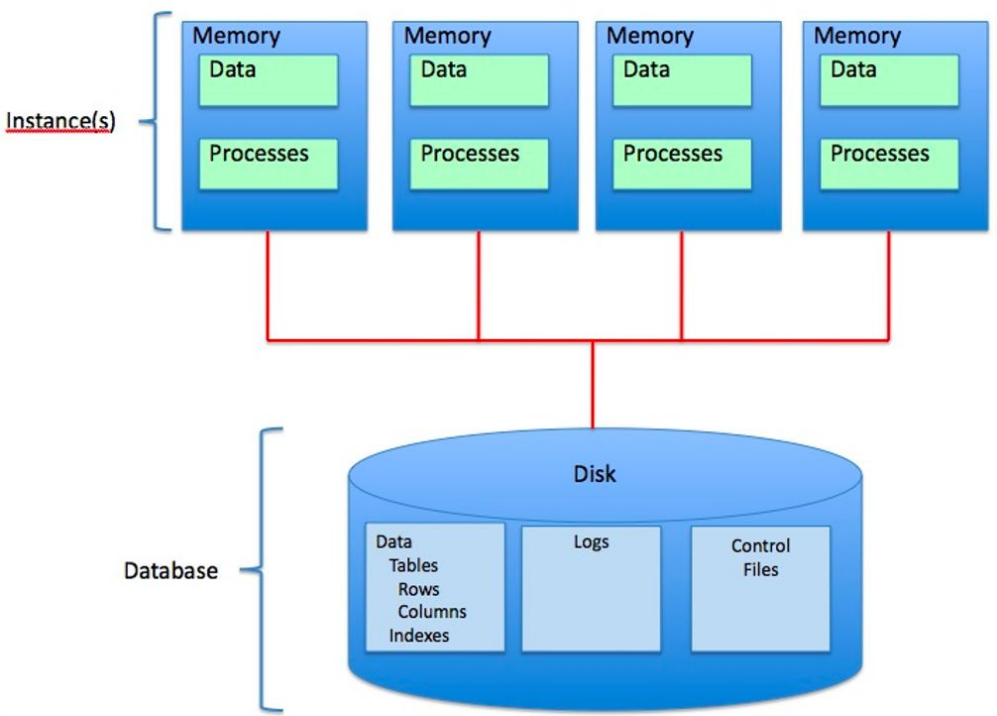
- Some fault tolerance (via hot-swappable components)



Vertical Scaling - Shared Disk Architectures

- Machines are connected via a fast network

- Contention and the overhead of locking limit scalability (high-write volumes) … BUT ok for Data Warehouse applications (high read volumes)



AWS EC2 Pricing - Oct 2024

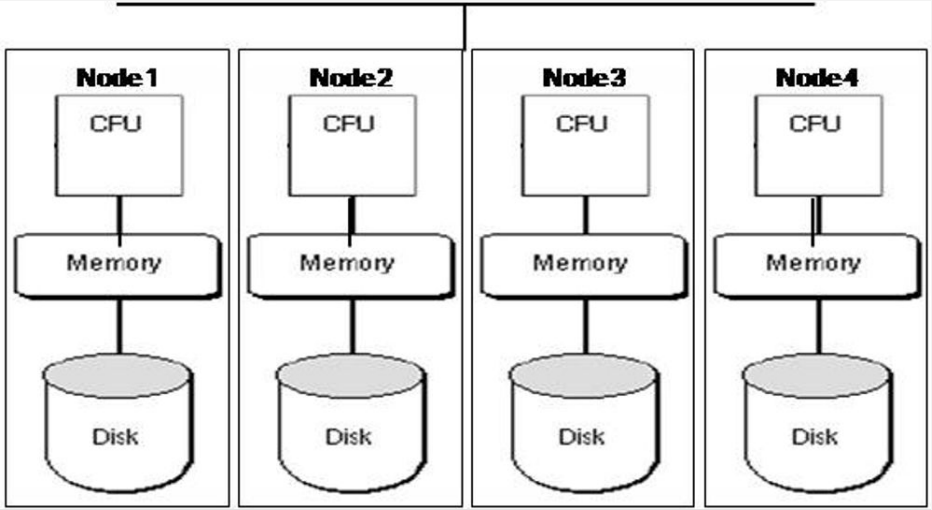
Instance name	On-Demand hourly rate	vCPU	Memory	Storage	Network performance
t4g.nano	$0.0042	2	0.5 GiB	EBS Only	Up to 5 Gigabit
t4g.micro	$0.0084	2	1 GiB	EBS Only	Up to 5 Gigabit
t4g.small	$0.0168	2	2 GiB	EBS Only	Up to 5 Gigabit
t3.medium	$0.0416	2	4 GiB	EBS Only	Up to 5 Gigabit
t3.large	$0.0832	2	8 GiB	EBS Only	Up to 5 Gigabit
t3.xlarge	$0.1664	4	16 GiB	EBS Only	Up to 5 Gigabit
t3.2xlarge	$0.3328	8		S Only	Up to 5 Gigabit
u-6tb1.112xlarge	$54.60	448		nly	100 Gigabit
u-9tb1.112xlarge	$81.90	448		nly	100 Gigabit
p5.48xlarge	$98.32	192	2048 GiB	8 x 3840 GB SSD	3200 Gigabit
u-12tb1.112xlarge	$109.20	448	12288 GiB	EBS Only	100 Gigabit

> $78,000/month

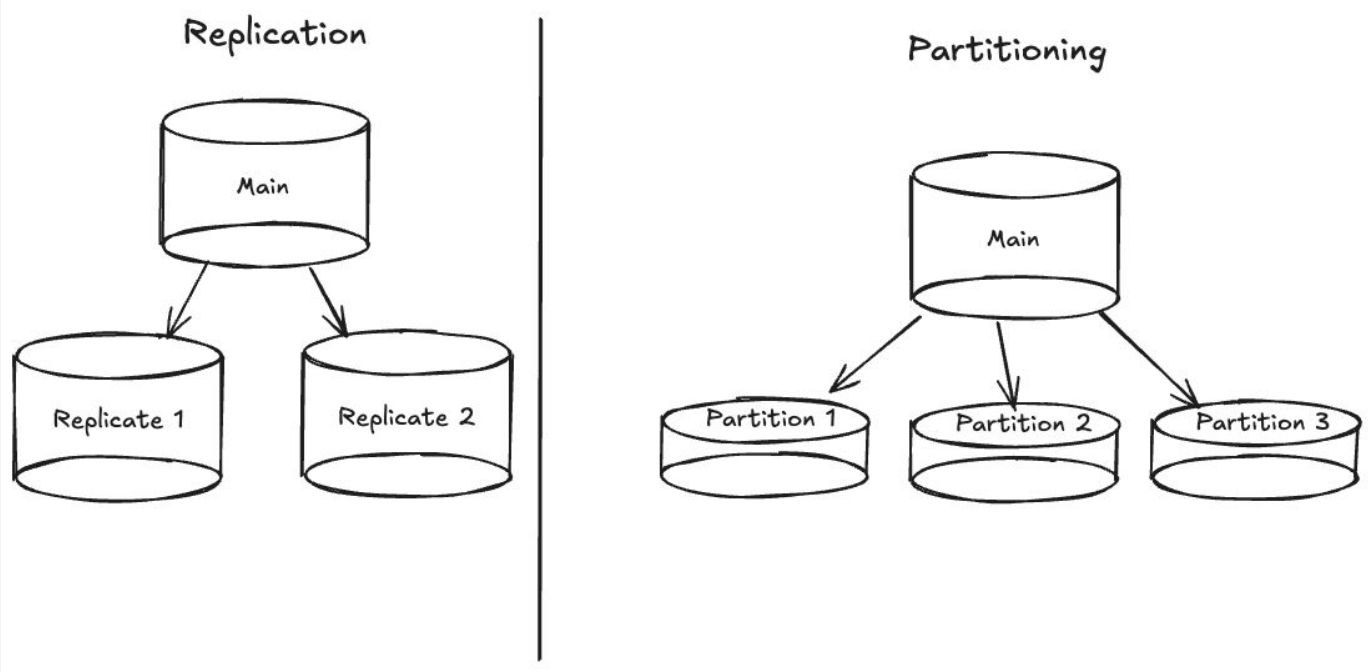
https://aws.amazon.com/ec2/pricing/on-demand/

Horizontal Scaling - Shared Nothing Architectures

- Each node has its own CPU, memory, and disk
- Coordination via application layer using conventional network
- Geographically distributed
- Commodity hardware



Data - Replication vs Partitioning



Replicates have same data as Main

Partitions have a subset of the data

Replication

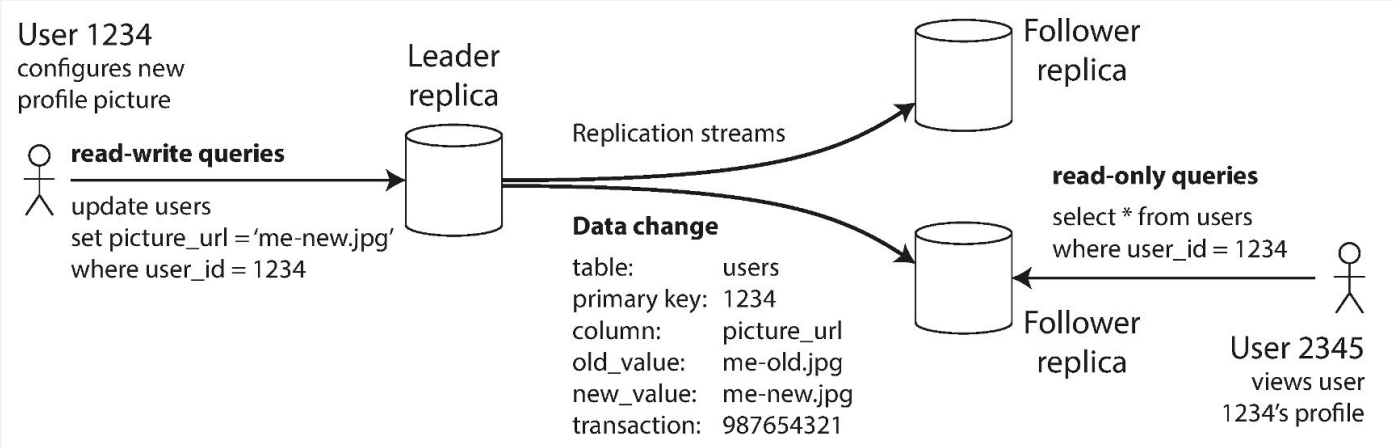
Common Strategies for Replication

- Single leader model
- Multiple leader model
- Leaderless model

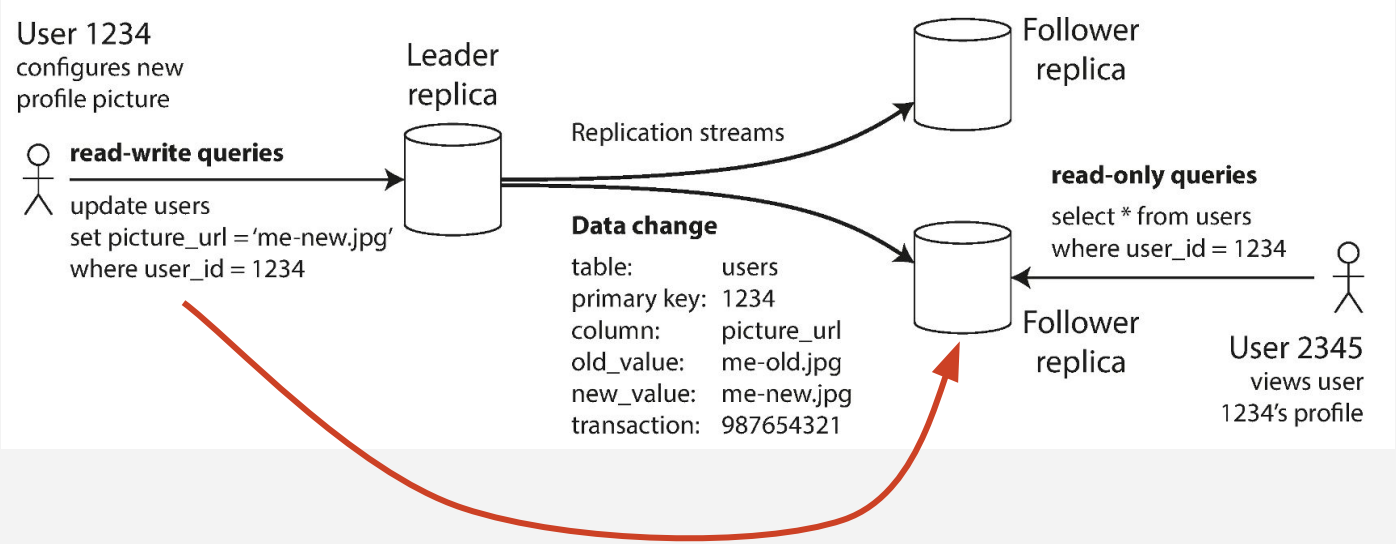
Distributed databases usually adopt one of these strategies.

Leader-Based Replication

- All writes from clients go to the leader
- Leader sends replication info to the followers
- Followers process the instructions from the leader
- Clients can read from either the leader or followers



Leader-Based Replication



This write could NOT be sent to one of the followers... only the leader.

Leader-Based Replication - Very Common Strategy

Relational:

- MySQL,
- Oracle,
- SQL Server,
- PostgreSQL

NoSQL:

- MongoDB,
- RethinkDB (realtime web apps),
- Espresso (LinkedIn)

Messaging Brokers: Kafka, RabbitMQ

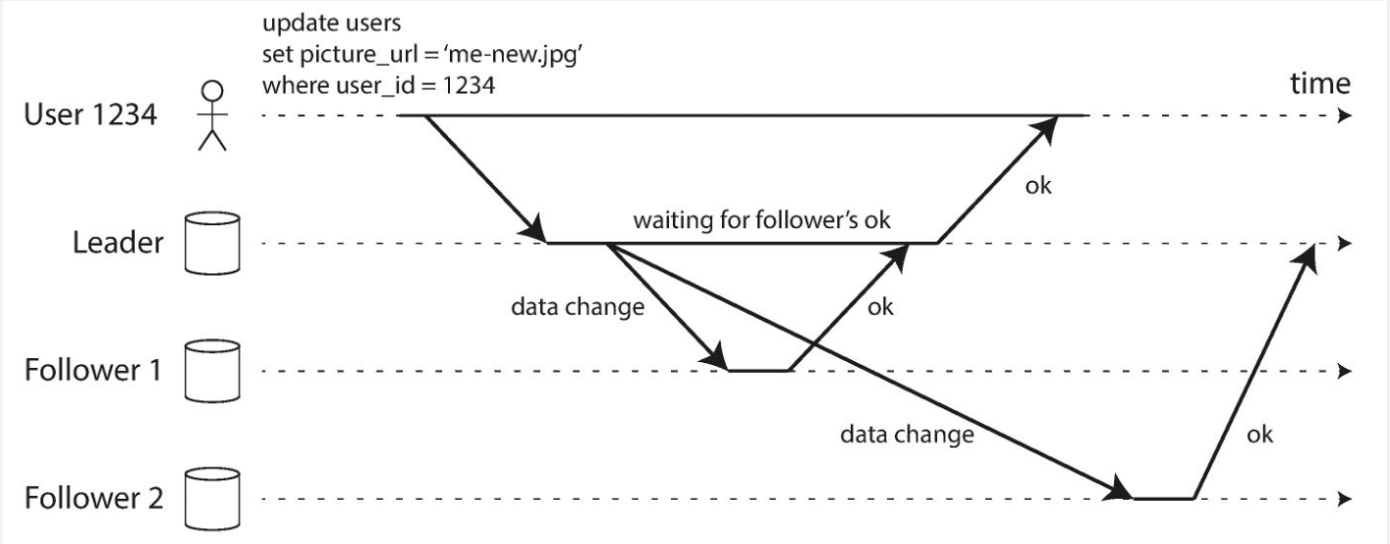
How Is Replication Info Transmitted to Followers?

Replication Method	Description
Statement-based	Send INSERT, UPDATE, DELETEs to replica. Simple but error-prone due to non-deterministic functions like now(), trigger side-effects, and difficulty in handling concurrent transactions.
Write-ahead Log (WAL)	A byte-level specific log of every change to the database. Leader and all followers must implement the same storage engine and makes upgrades difficult.
Logical (row-based) Log	For relational DBs: Inserted rows, modified rows (before and after), deleted rows. A transaction log will identify all the rows that changed in each transaction and how they changed. Logical logs are decoupled from the storage engine and easier to parse.
Trigger-based	Changes are logged to a separate table whenever a trigger fires in response to an insert, update, or delete. Flexible because you can have application specific replication, but also more error prone.

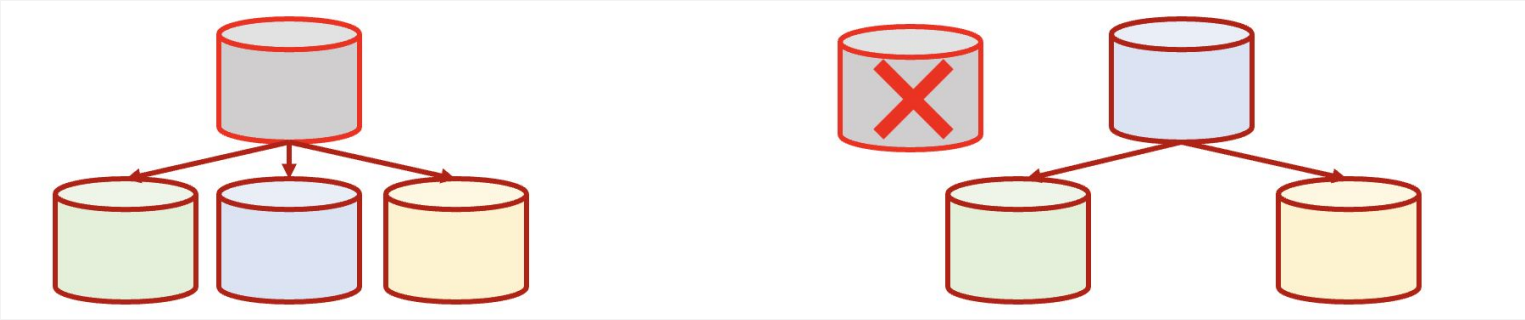
Synchronous vs Asynchronous Replication

Synchronous: Leader waits for a response from the follower

Asynchronous: Leader doesn't wait for confirmation.



What Happens When the Leader Fails?

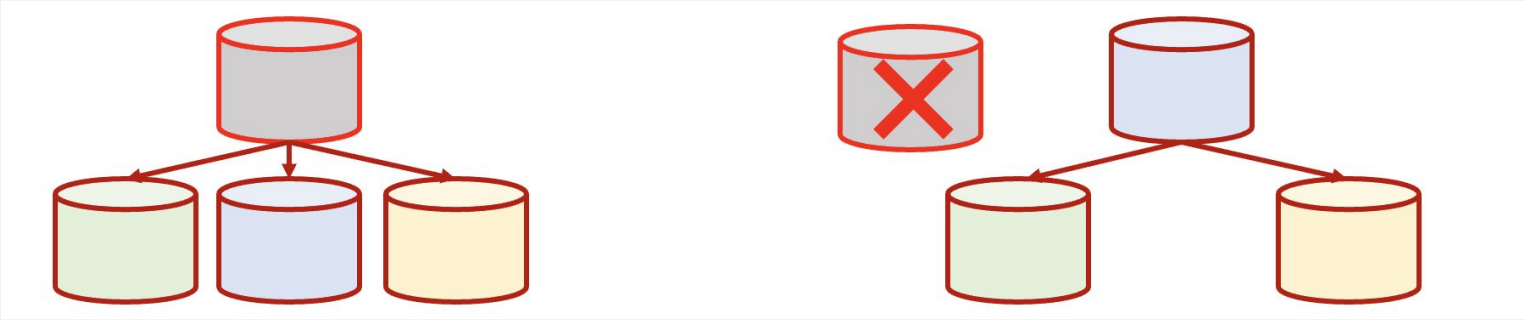


Challenges: How do we pick a new Leader Node?

- Consensus strategy – perhaps based on who has the most updates?
- Use a controller node to appoint new leader?

AND… *how do we configure clients to start writing to the new leader*?

What Happens When the Leader Fails?



More Challenges:

- If asynchronous replication is used, new leader may not have all the writes How do we recover the lost writes? Or do we simply discard?
- After (if?) the old leader recovers, how do we avoid having multiple leaders receiving conflicting data? (Split brain: no way to resolve conflicting requests.
- Leader failure detection. Optimal timeout is tricky.

Replication Lag

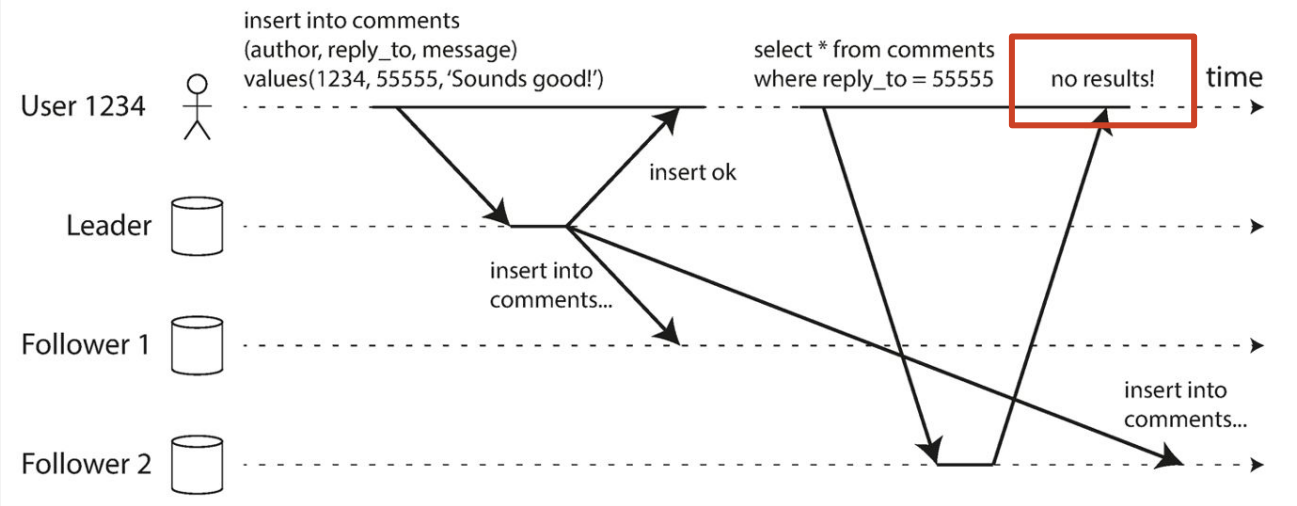
Replication Lag refers to the time it takes for writes on the leader to be reflected on all of the followers.

- **Synchronous replication**: Replication lag causes writes to be slower and the system to be more brittle as num followers increases.
- **Asynchronous replication**: We maintain availability *but at the cost of delayed or eventual consistency*. This delay is called the *inconsistency window*.

Read-after-Write Consistency

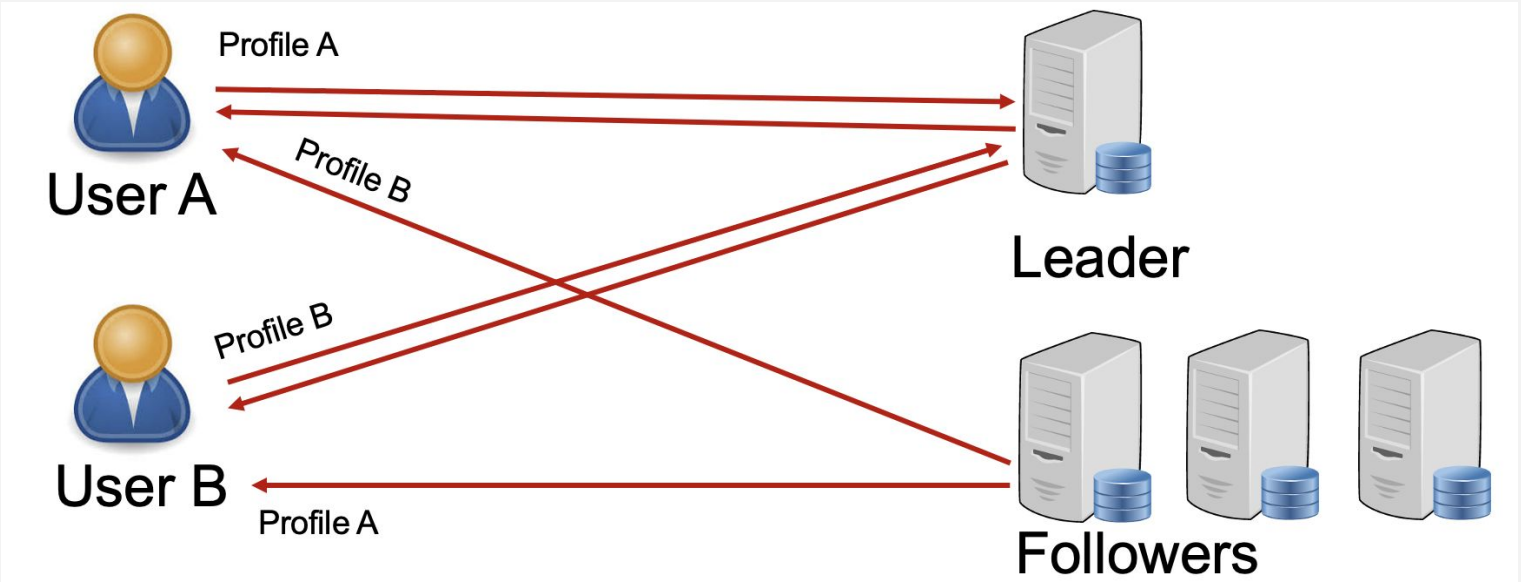
Scenario - you're adding a comment to a Reddit post… after you click **Submit** and are back at the main post, your comment should show up for you.

- Less important for other users to see your comment as immediately.



Implementing Read-After-Write Consistency

Method 1: Modifiable data (from the client's perspective) is always read from the leader.

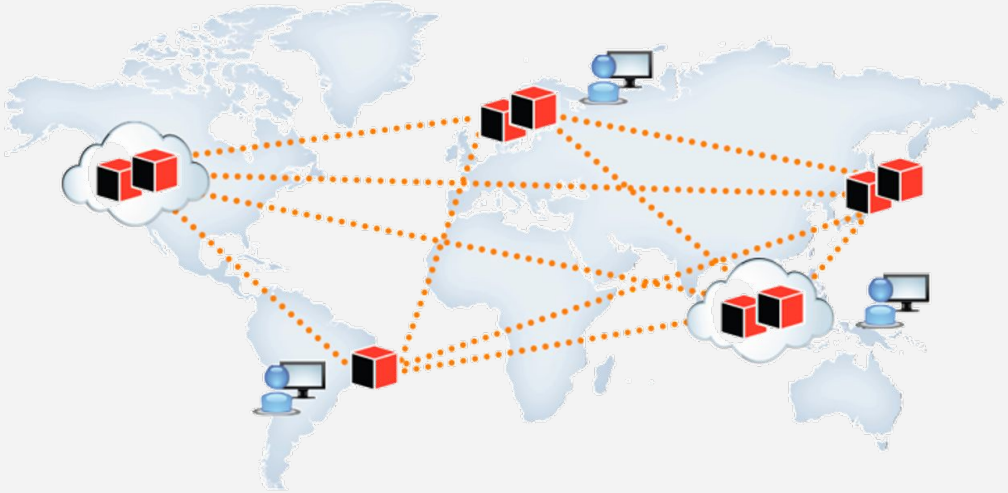


Implementing Read-After-Write Consistency

Method 2: Dynamically switch to reading from leader for "recently updated" data.

- For example, have a policy that all requests within one minute of last update come from leader.

But... This Can Create Its Own Challenges

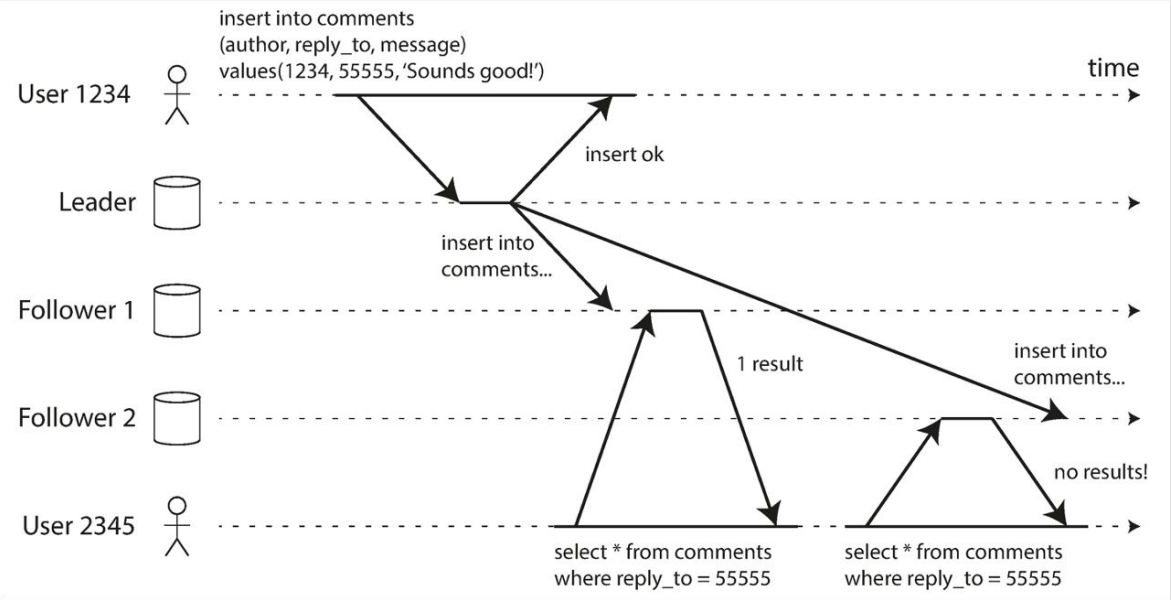


We created followers so they would be proximal to users. BUT… now we have to route requests to distant leaders when reading modifiable data?? :(

Monotonic Read Consistency

Monotonic read anomalies: occur when a user reads values out of order from multiple followers.

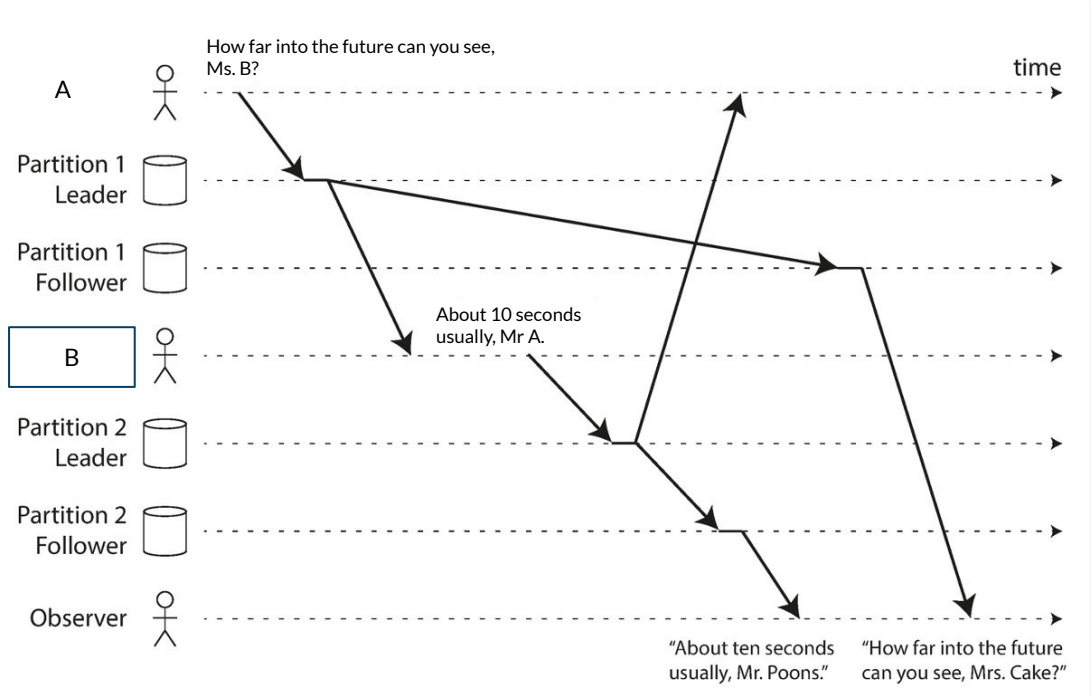
Monotonic read consistency: ensures that when a user makes multiple reads, they will not read older data after previously reading newer data.



Consistent Prefix Reads

Reading data out of order can occur if different partitions replicate data at different rates. There is *no global write consistency*.

Consistent Prefix Read Guarantee - ensures that if a sequence of writes happens in a certain order, anyone reading those writes will see them appear in the same order.



??