# Cloud-Based Data Processing

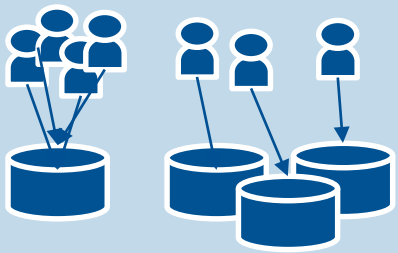
## Distributed Data: Replication

Jana Giceva

# Distributed Data

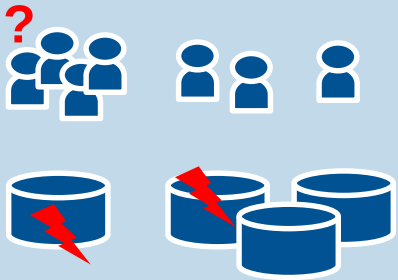
- **Why distribute data across multiple machines?**

## Scalability



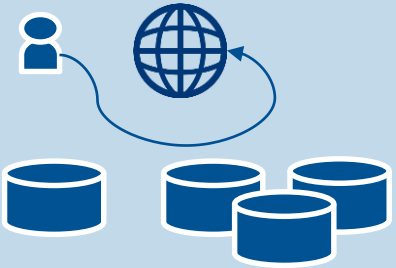
If data volume, or the read and write load grows bigger than what a single node can sustain, spread it across different machines

## Fault Tolerance




If a machine (or DC / network) fails, you can use redundancy. When one fails, another one can take over.

## Tail Latency



If you have users across the world, you want data to be in a DC that is geographically close to the users → reduce the response time.

# Replication vs. Partitioning

- There are two common ways data is distributed across multiple nodes.

- **Replication**
  - Keeps a copy of the same data on different nodes (potentially different locations).
  - Provides redundancy – If some nodes are unavailable, others can continue serving requests.
  - Reduces latency especially for high load or wide distribution of users across the globe.

- **Partitioning**
  - Split the big dataset into smaller subsets called *partitions.*
  - Each partition placed on a separate node.
  - Reduces latency for analytical jobs
  - Can improve availability

- One can combine both replication and partitioning!

# Replication

# Replication

- **Replication** – keeping a **copy** of the same data **on multiple machines** that are connected via network.

- **Benefits** of replication:
  - Keep data geographically close to users – **reduce the access latency**
  - Ensure the system continues working even in case of failures – **increase availability**
  - Scale out the number of machines used that can serve read queries – **increase read throughput**

- **For read-only workload, data replication is easy and always beneficial.**

- Replication is an old topic, that was extensively studied in the 1970s, but has been popularized recently.

# Challenges of Replication

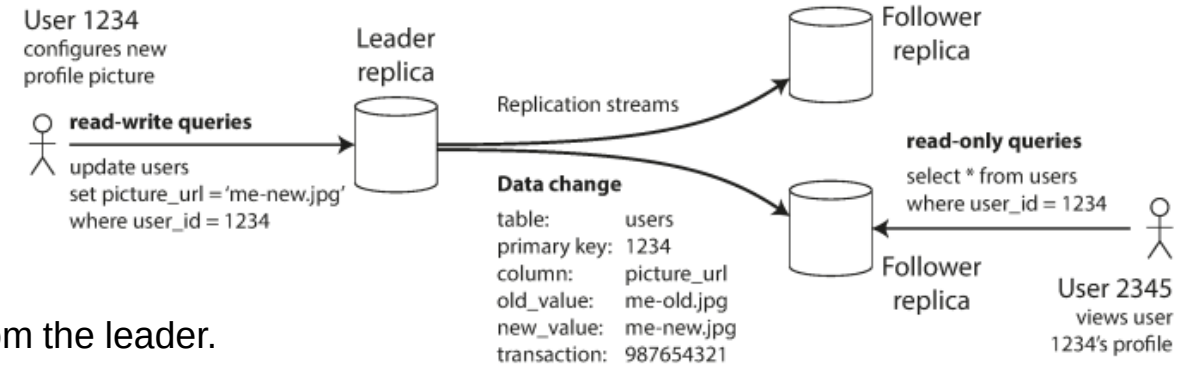
- **How to handle data that changes in a replicated system**?
  - Should there be a **leader replica** and if yes, how many?
  - Should one use a **synchronous** or **asynchronous propagation** of the **updates** among the replicas?
  - How to **handle a failed replica** if it is the follower?
  - What if the leader failed?
  - How does a resurrected replica catch up with the leader?

# Leaders and Followers

- Each node that stores a copy of the dataset is called a **replica.**

- Every write needs to be processed by every replica; otherwise, the nodes will not hold the same data.
- The most common solution is **leader-based replication.**

- **Leader** – when clients write to the database, they must send their request to the leader.



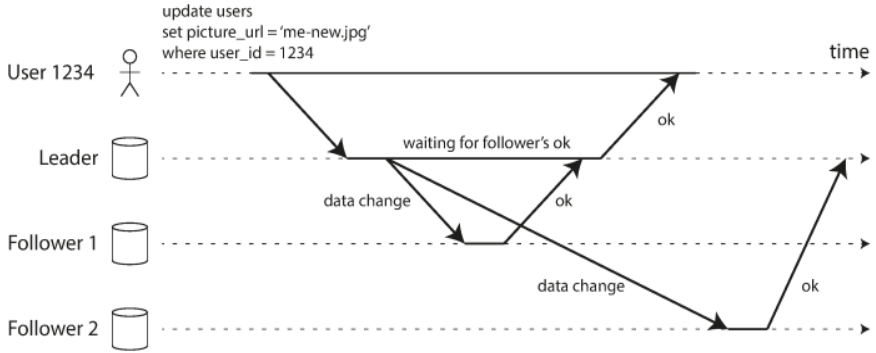

- Other replicas are known as **followers,** which are updated by applying the **replication log** from the leader.

- A client can **read from anywhere** (the leader or any follower).

PostgreSQL (since v9.0), MySQL, Oracle Data Guard, SQL Server's AlwaysOn Availability Groups, MongoDB, RethinkDB, Espresso, Kafka, RabbitMQ, some networked FS and replicated block devices.

# Synchronous vs. asynchronous Replication

- **Synchronous** if the leader waits until the follower(s) have confirmed that they applied the write before reporting success to the user.
  - e.g., the replication to follower 1 is synchronous.




- **Asynchronous** if the leader sends the update message to its follower(s), but does not wait for a response before answering success to the user.
  - e.g., the replication to follower 2 is asynchronous.

- What are the advantages and disadvantages of each?
  - Advantages of synchronous is that the follower is guaranteed to have an up-to-date version of the data.
  - But, if a synchronous follower does not respond – the system will not be able to support writes.

  - Fully-asynchronous replication trades availability at the cost of weakened durability

# Setting up new Followers

- How to ensure that the **new follower** has an **accurate copy** of the leader's data **without downtime**?
  - Simply copying data files from one node to another is not sufficient as clients are constantly writing.

- The process needs a few steps:
  1. **Take** a **consistent snapshot** of the leader's database (the same one used for back-up).
  2. **Copy** the **snapshot to** the **follower** node.
  3. The follower **gets** the **leader's replication log** since the snapshot.
  4. Once the follower has processed the backlog, we say it has **caught up.**

# Handling node outages/failures

- Any node in the system can go down.

- Goal is **high availability with leader-based replication**
  - i.e., how to reboot individual nodes without downtime.

- **Follower** failure
  - → **catch-up recovery**
  - Keep a log of the data changes already applied on a local disk
  - After a reboot, apply the outstanding changes before re-connecting to the leader

- **Leader** failure
  - → **failover**
  - Detect that the leader has failed.
  - Promote one of the followers as a new leader.
  - Reconfigure the system to use the new leader

# Implementation of Replication Logs

- **Statement based replication**
  - The leader logs every write request (statement) that it executes
- **Write-ahead log (WAL) based replication – physical log**
  - Use WAL to build and maintain the followers.
  - But, the log is very low level and replication is coupled to the storage engine
- **Change data capture (CDC) based replication – logical log**
  - Sequence of records that describe the change to database tables at the granularity of rows.
  - Replicas can run on different versions or storage engines.
  - Easier to parse for external applications.
- **Trigger-based replication** (done in application layer)

# Problems with Replication Lag

- In **Leader-based replication** all writes go to the leader, but read-only queries can go to any replica.
- This makes it **attractive** also **for scalability** and **latency**, in addition to fault-tolerance.

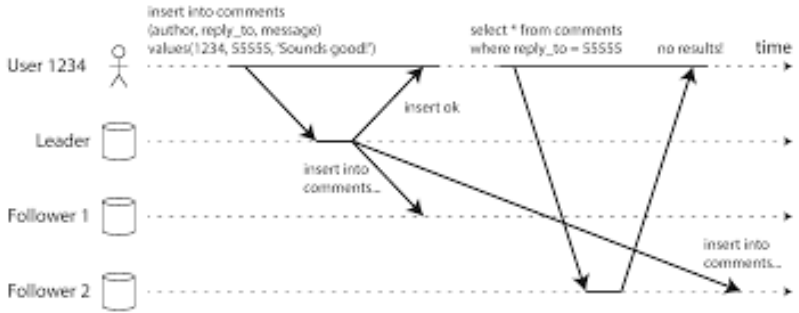
- **For read-mainly workloads: have many followers and distribute the reads across those followers.**
  - Removes load from the leader, allows read requests to be served by nearby replicas.
  - But, **only realistic for asynchronous replication** otherwise the system will not be available.

- If an application reads from an asynchronous follower, it may see outdated information.
  - Running the same query on the leader and a follower at the same time may get inconsistent results.
  - The effect is known as **eventual consistency.**

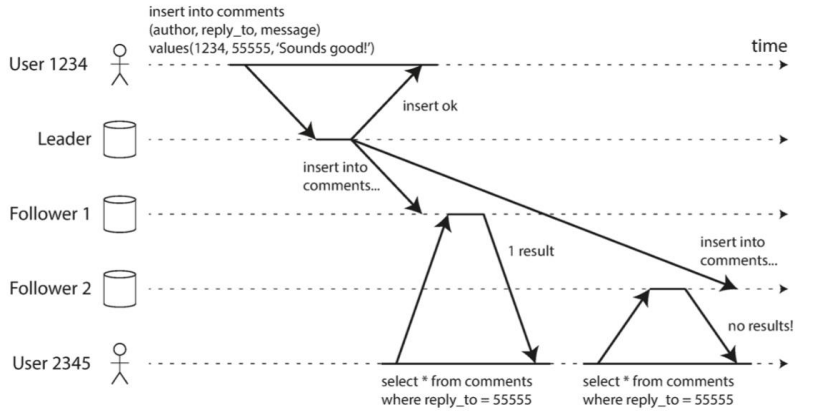
- The term **eventually** is deliberately vague – there is no limit how far a replica can fall behind.

# Example problems with eventual consistency

- **Reading you own writes**
  - Requires read-after-write consistency
  - Makes no promises to other users
  - e.g., (always or for some time after a write) read from the leader, read based on timestamp
  - Cross-device read-after-write consistency



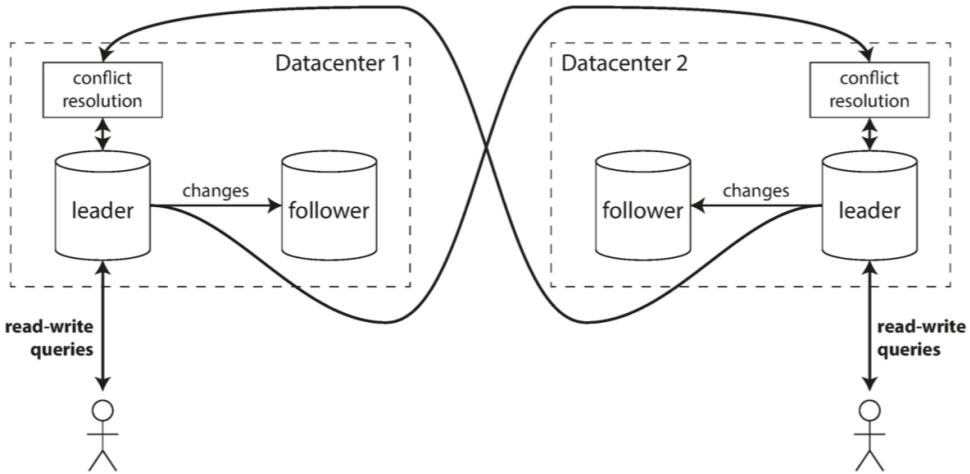

- **Monotonic reads**
  - Avoid a user to see things as moving back in time.
  - If one user makes several reads in sequence, they will not read older data after previously reading newer data.
  - e.g., by enforcing that each user always makes their reads from the same replica.



# Multi-leader Replication

- **Leader-based replication has a single bottleneck – the leader.**
  - All writes must go through it. If there is a network interruption between the user and the leader, then no writes are allowed.

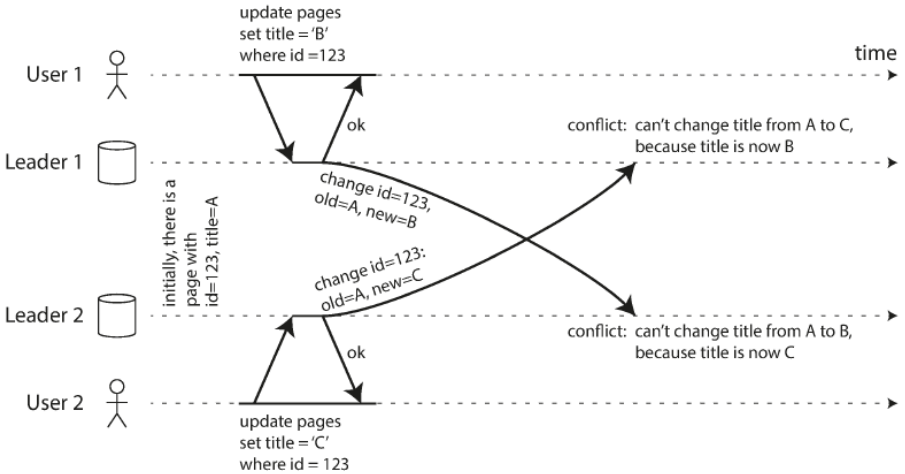
- **Alternative** approach **is multi-leader based replication.**
  - Multi-datacenter operation
    - advantages to single-leader replication for performance, tolerance to DC and network outages.
  - Clients with offline operations
    - Every device has a local database that acts as a leader.
  - Collaborative editing



# What problems do you anticipate here?

# Conflict resolution

- A problem with multi-leader replication is that **write conflicts** can **occur.**




- **Handling write conflicts:**
  - **Synchronous** vs **asynchronous** make the conflict detection synchronous.
  - **Conflict avoidance:** make all writes for a particular record go through the same leader
  - **Converging to a consistent state:**
    - last writer wins (LWW): each write has a unique ID, pick the write with the highest ID as the winner and throw away all other writes → prone to data loss.
    - let the application decide on the custom conflict resolution logic (on read or write).

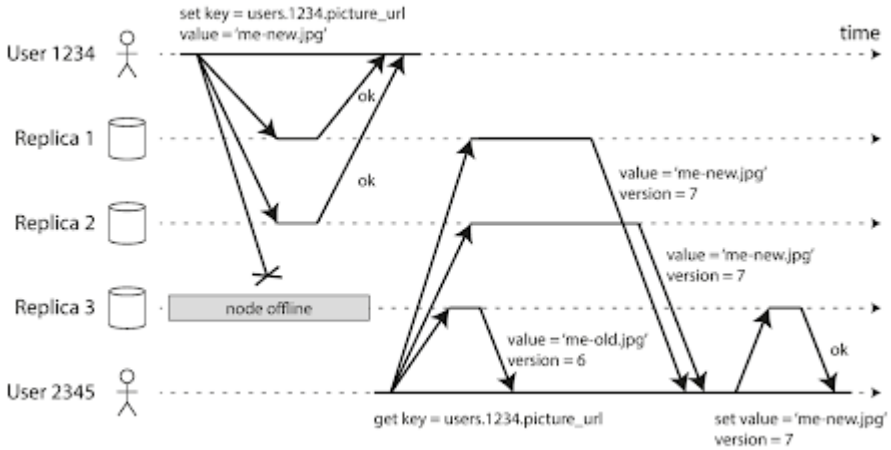
# Leaderless Replication

- Abandon the concept of a leader, and **allow any replica to directly accept writes from clients**.

- Some of the earliest replicated data systems were leaderless (from the 1970s), but the idea was **resurrected by Amazon's Dynamo**.
  - Riak, Cassandra, and Voldemort are open source datastores with leaderless replication models, inspired by Dynamo. Also known as Dynamo-style replication.

- In a leaderless replication, there is no failover when a node fails.

- **A client both writes to and reads from multiple nodes in the system**.
  - **Version numbers** are used to determine which value is newer in case of different read values.



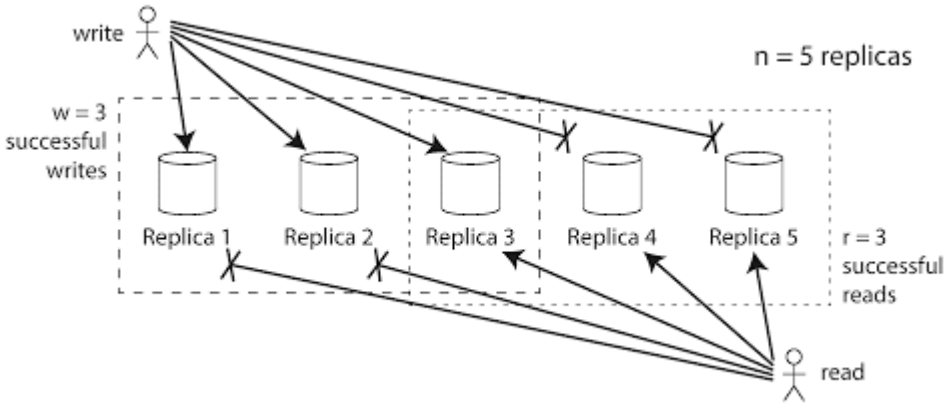
# How does a node catch up the writes it missed

- **The replication system should ensure that eventually all the data is copied to every replica.**
- After an unavailable node comes back online, **how does it catch up on the writes it missed?**
- Two mechanisms are often used:
  - **Read repair**: the client can detect a stale response, and can write a newer value back to the replica.
    - Works well for values that are frequently read.
  - **Anti-entropy process**: a background process that checks for inconsistencies and fixes them.
    - Unlike the replication log in the leader-based replication mechanisms, here the order is not preserved.

# So, what is the truth in a leaderless system?

# Quorums for reading and writing

- **If there are $n$ replicas, every write must be confirmed by $w$ nodes to be considered successful, and we must query at least $r$ nodes for each read.**




- As long as $w + r > n$, we expect to get an up-to-date value when reading.
- Reads and writes that obey these $r$ and $w$ values are called **quorum** reads and writes.
- The quorum conditions, allows the system to tolerate unavailable nodes as follows:
  - If w<n, we can still process writes if a node is unavailable.
  - If r<n, we can still process reads, if a node is unavailable.
  - With (3,2,2) we can tolerate 1 node failure, with (5,3,3) we can tolerate 2 nodes.
  - Normally, reads and writes are sent to all n replicas in parallel; w and r determine how many nodes we wait for before we consider the read or write to be successful.

# Limitations of Quorum Consistency

- **Even with w+r > n, there are likely to be edge-cases where stale values are returned.**
  - E.g., with sloppy quorum;
  - if two writes occur simultaneously;
  - if a write happens at the same time as a read;
  - if a write succeeded on some replicas but failed on others and overall succeeded on less than w nodes;
  - if a node carrying a new value fails, and its data is restored from a replica carrying an old value.

- **Monitoring staleness and quantifying “eventual”.**
  - There is no fixed order in which writes are applied – making monitoring of data staleness difficult.
  - It would be good to include staleness measurement in the standard set of metrics to quantify “eventual”.

# Sloppy quorum and hinted handoff

- Quorums are not as fault-tolerant as they could be.
  - A network interruption can cut off a client from a large number of database nodes.

- A **sloppy quorum**: used in case of network partitioning. The writes and reads still require w and r successful responses, but those may be by nodes that are not the designated “home” nodes for a value.
  - Once the network interruption is fixed, any writes that one node temporarily accepted on behalf of another node are sent to the appropriate “home” nodes. This is called the **hinted handoff**.

- Particularly good to increase write availability

- Note that it is not a quorum in the traditional sense, it is only an assurance of durability.

# Summary

- **Replication is used for high availability, disconnected operation, latency and scalability.**

- **Three main approaches to replication**:
  - Single-leader replication
  - Multi-leader replication
  - Leaderless replication

- Replication can be **synchronous** or **asynchronous**. Follower replicas apply the **replication log.**

- Different ways to keep replicas in sync, or recover when a replica fails, etc.

- Replication **lag** can **lead** to **eventual consistency**. Some other consistency models that may be helpful:
  - Read-after-write consistency
  - Monotonic reads
  - Consistent prefix reads

# References

The material covered in this class is mainly based on:

- The book *“Designing Data-Intensive Applications – The Big Ideas Behind Reliable, Scalable, and Maintainable Systems”* by Martin Kleppmann (Chapters 5 and 6) (link)