# Distributed Data - Replication

UA.DETI.CBD
José Luis Oliveira / Carlos Costa

# Replication

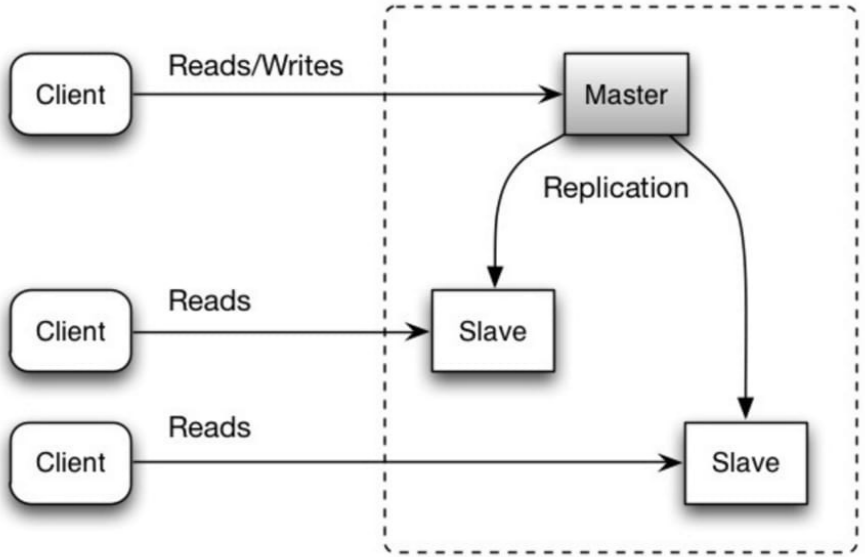
- Definition - keeping a **copy of the same data on multiple machines** that are connected via a network
  - Assumption: the dataset is so small that each machine can hold a copy of the entire dataset
- Why?
  - **Reduce Latency**: keep data geographically close to users
  - **Increase Availability**: allow the system to continue working even if some parts fail
  - **Scalability**: to scale out the number of machines that can serve read queries (and thus increase read throughput)
- Challenge - handling changes in the data
  - easy task if replicating data does not change over time

# Replication algorithms

- Popular algorithms for replicating changes between nodes:
  - **Single-leader**
  - **Multi-leader**
  - **Leaderless**

- Trade-offs to consider with replication
  - synchronous or asynchronous replication
  - how to handle concurrency
  - how to handle failed replicas

# Single-Leader Replication



# Leader and Followers

- **Replica** - each node that stores a copy of the database
  - Every write to the database needs to be processed by every replica
- Leader-based replication
  - one of the replicas is designated the **leader** (also known as master or primary)
  - other replicas are the **followers** (read replicas, slaves, or hot standbys)
- Other designations:
  - active/passive or master-slave replication

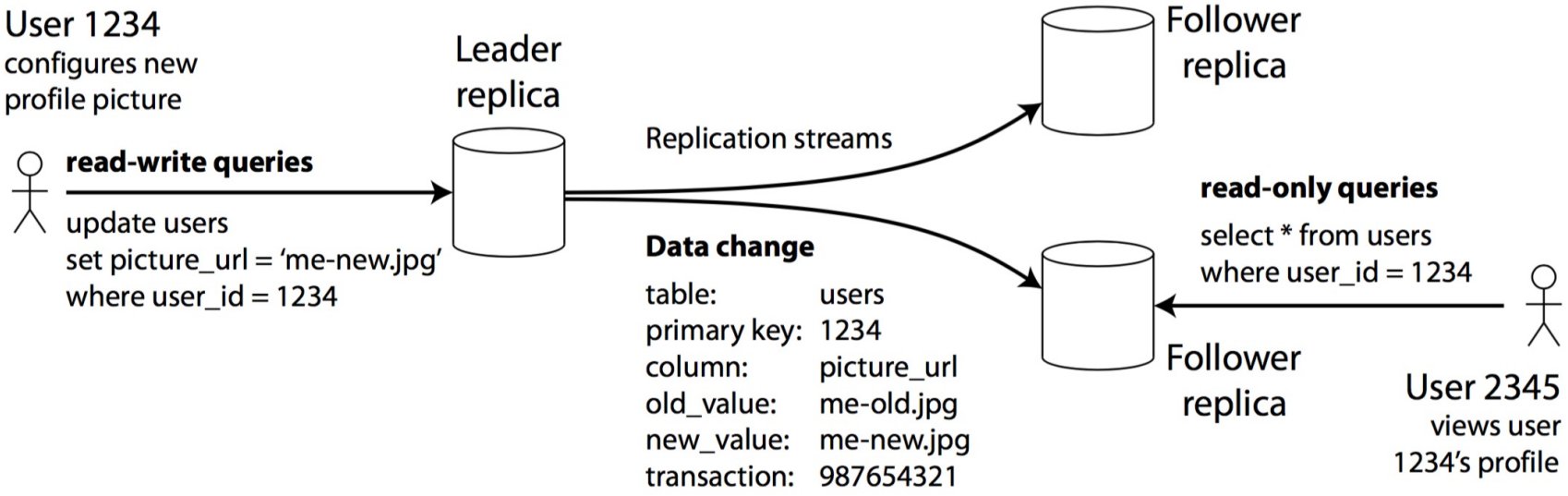
# Leader and Followers

- Single-Leader Replication – **most common solution**

- This mode of replication is a built-in feature of many relational databases
  - e.g. PostgreSQL, MySQL, Oracle Data Guard, and SQL Server, …
- Also used in some non-relational databases
  - e.g. MongoDB, RethinkDB and Espresso

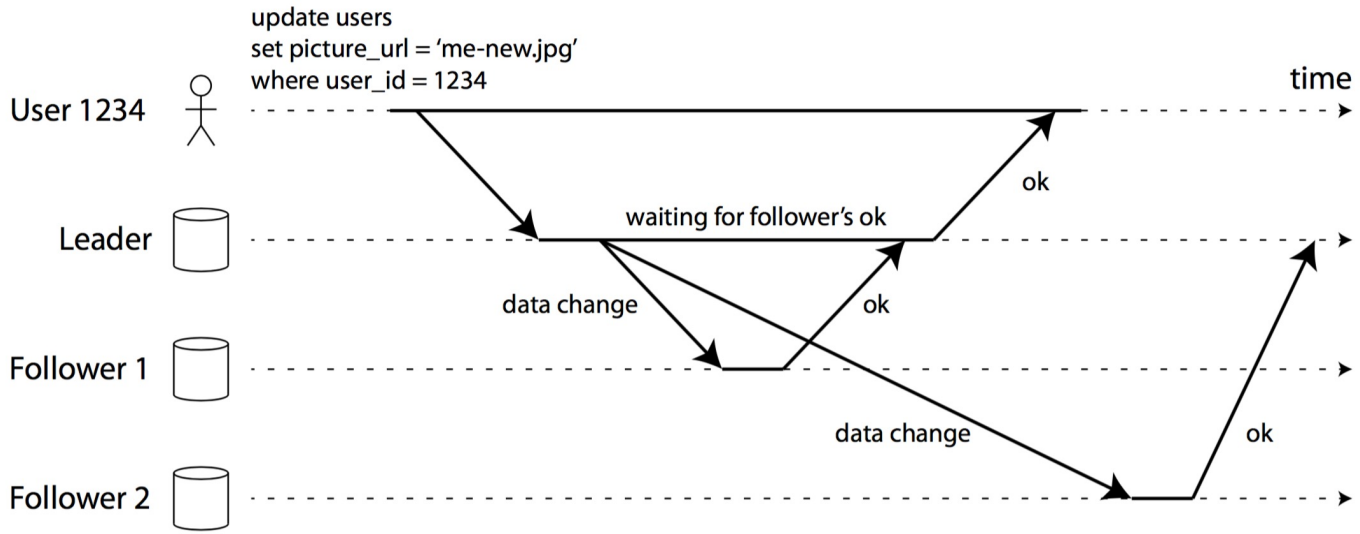
# Leader and Followers – works like

1. **Clients writes** to the database must be send to the leader
2. **Leader** writes the new data to its local storage
3. Leader sends the data change to the followers
   - using replication log or change stream
4. Each **follower** takes the log and updates its local copy
   - all writes are processed in the same order as processed on the leader
5. **Clients reads** from the database can be done by query either the leader or any of the followers



# Synchronous vs. asynchronous (1)

- Follower 1 replication is **synchronous**: the leader waits for follower 1 write confirmation before reporting success to the user, and before making the write visible to other clients
- Follower 2 replication is **asynchronous**: the leader sends the message, but doesn't wait for a response from the follower




- In relational databases, this is often a configurable option; other systems are often hard-coded to be either one or the other

# Synchronous vs. asynchronous (2)

- Replication is usually quite fast
  - most database systems apply changes to followers in less than a second
- But… no guarantee for how long it might take
- There are circumstances when followers might fall behind the leader by several minutes or more, for example:
  - follower is recovering from a **failure**
  - system is operating near maximum **capacity**
  - **network** problems between the nodes

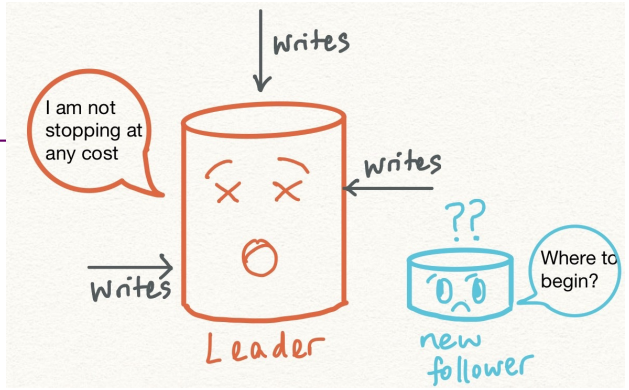
# Synchronous vs. asynchronous (3)

- **Advantage** of synchronous replication - the follower have an up-to-date copy (consistent) of the data
  - if leader fails, the data is available on the follower
- **Disadvantage** - the write cannot be processed if the synchronous follower doesn't respond
  - leader must block all writes until the synchronous replica is available again
- Impractical to have all followers synchronous
  - in practice, if you enable synchronous replication on a database, it usually means that one of the followers is synchronous, and the others are asynchronous

# Synchronous vs. asynchronous (4)

- **Semi-synchronous** configuration
  - if the synchronous follower becomes unavailable or goes slow, one of the asynchronous followers is made synchronous
  - guarantees an up-to-date copy of the data on at least two nodes: the leader and one synchronous follower
- **Fully asynchronous** configuration is often used
  - advantage that the leader can continue processing writes, even if all of its followers have fallen behind
  - however, write is not guaranteed to be durable, even if it has been confirmed to the client
    - if the leader fails and is not recoverable, any writes that have not yet been replicated to followers are lost

# Setting up new followers




- How to setup a new follower?
  - simply copying data files between nodes is typically not sufficient
    - clients are constantly writing to the database
  - locking the database (for writes) goes against high availability
- **Algorithm** (without downtime)
  - take a consistent snapshot of the leader's database at some point in time (most databases have this feature for backups)
  - copy the snapshot to the new follower node
  - connects the follower to the leader
  - Follower requests all data changes since the snapshot was taken

# Handling node outages

- **How do you achieve high availability** with leader-based replication?
- Any node in the system can go down
  - unexpectedly due to a fault
  - planned maintenance
- We need to keep the system running, despite individual failed nodes
  - i.e., minimize the impact of a node outage
- **Two types of fails**:
  - Follower
  - Leader

# Follower failure

- **Catch-up recovery**
  1. followers keep a log of data changes (received from leader)
  2. log used to know the last transaction before the fault occurred
  3. connect to leader and request all data changes that occurred during the time when the follower was disconnected
  4. apply these changes
  5. continue receiving a stream of data changes as before (regular operation state)

# Leader failure

- Handling a failure of the leader is trickier:
  - one of the followers is promoted to **new leader**
  - **clients** need to be **reconfigured** to send **writes** to the new leader
  - other **followers** need to start **consuming data** changes from the new leader
- This process is called **Failover**
- Failover can happen
  - manually
    - an administrator is notified that the leader has failed, and takes the necessary steps to make a new leader
  - automatically

# Automatic failover process

- **Determining that the leader has failed**
  - there is no foolproof way of detecting
  - most systems simply use a timeout
    - a node is assumed to be dead if it doesn't respond for some period of time
- **Choosing a new leader**
  - through an election process, where the leader is chosen by a majority of online replicas, or it can be appointed by a previously-elected controller node
    - the best candidate for leadership is usually the most up-to-date replica
- **Reconfiguring the system** to use the new leader
  - clients now need to send their write requests to the new leader (using a router/dns/service discovery-kind service)
  - problem: if the old leader comes back, it might still believe that it is leader
  - the system needs to ensure that the old leader becomes a follower and recognizes the new leader

# Timeout for declaring a leader dead?

- No easy solution
  - Longer timeout: a longer time to recovery in the case where the leader fails
  - Short timeout: there could be unnecessary failovers
    - for example, a temporary load spike could cause a node's response time to increase above the timeout, or a network glitch could cause delayed packets
    - if the system is already struggling with high load or network problems, an unnecessary failover is likely to make the situation worse, not better

- For this reason, some operations teams prefer to perform failover manually

# Implementation of replication logs

- A replication log includes information about write operations in the database
  - e.g., about Inserted, Deleted and Updated rows

- Four main methods
  1. Statement-based replication
  2. Write-Ahead Log (WAL)
  3. Logical log replication
  4. Trigger-based replication

# 1. Statement-based replication

- Leader logs executed write statement and sends that statement log to the followers
  - Each follower executes the statement in its node
- Problems:
  - calls to non-deterministic function, for example NOW() or RAND(), will generate a different value on each replica.
  - statements using an auto-incrementing column or depend on the existing data in the database
  - statements that have side-effects (e.g. triggers, sp, udf) may result in different results on each replica
  - ensure execution order of statements in every node
- Other replication methods are generally preferred

# 2. Write-ahead log (WAL)

- Whenever a query comes to a system, even before executing that query, it is written in an **append-only log** file also known as Write-ahead log file.
- Besides writing the log to disk, the leader also sends it across the network to its followers
- Disadvantage:
  - log describes the data on a very low level
    - makes the replication closely coupled to the storage engine
    - difficult to run different versions of the database software on the leader and the followers
- This method of replication is used in PostgreSQL and Oracle, among others

# 3. Logical log replication

- An alternative is to use **different log formats** for replication and for the storage engine.
- More easily be kept backwards-compatible
  - leader and follower can run different versions of the database software, or even different storage engines
- A logical log format is also easier for external applications to parse
  - e.g., data warehouse for offline analysis, or for building custom indexes and caches

# 4. Trigger-based replication

- **Replication at application layer** - more flexibility
- Usage examples:
  - replicate a subset of the data
  - replicate from one kind of database to another
- Approaches (at database system layer):
  - by reading the database log
  - use database features (e.g., triggers and stored procedures)
    - trigger can log database writes into a separate table where an external process can read it
    - trigger-based replication typically has greater overheads than other replication methods

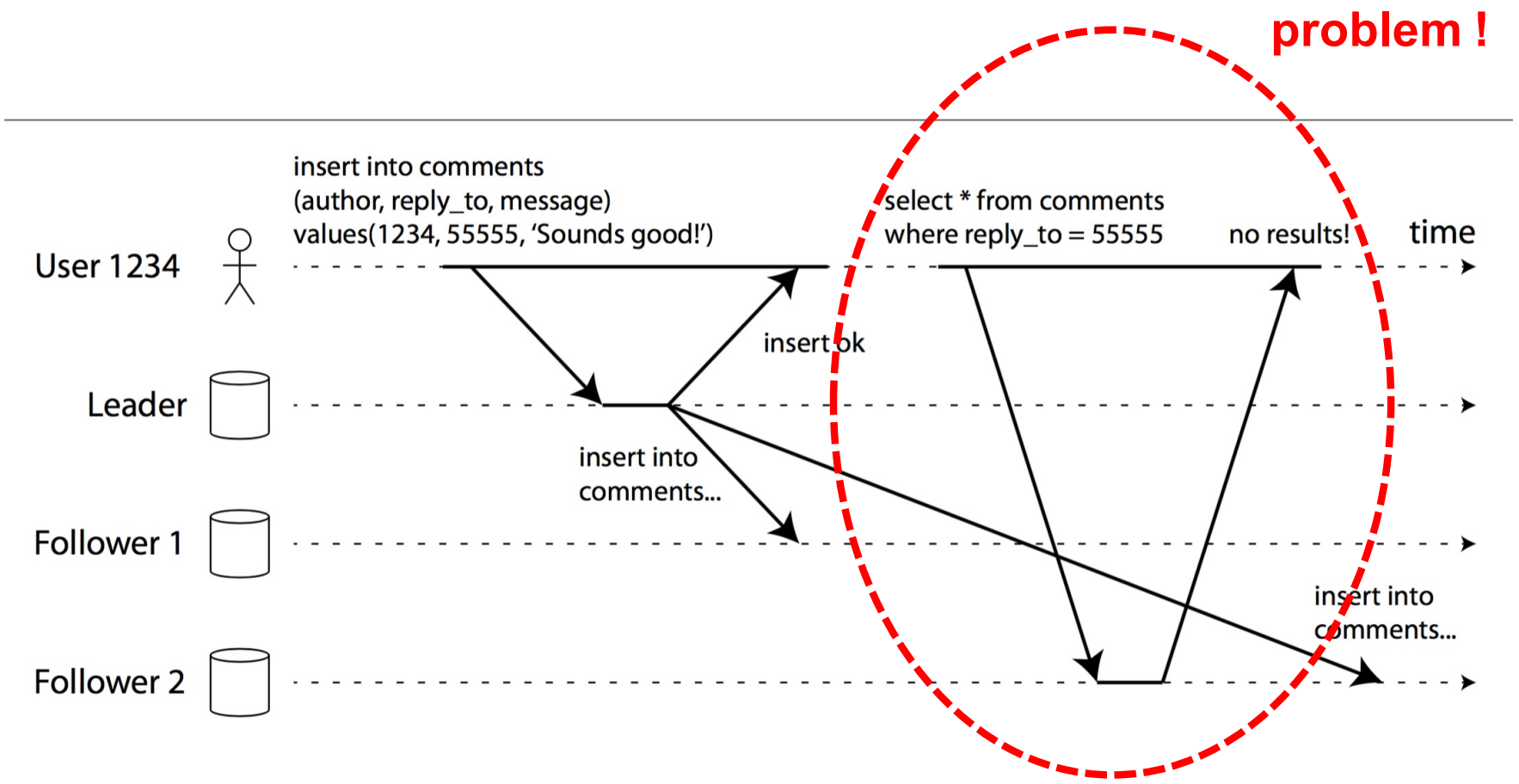
# Replication Lag Problem

- Read-scaling architecture
  - **add followers** to increases the capacity for serving read-only requests
  - removes load from the leader
- But this may cause a **Replication Lag**
  - reads from asynchronous followers may see outdated information
  - e.g., if we run the same query on the leader and a follower at the same time, we may get different results
- **Eventual consistency**
  - This inconsistency is just a temporary state
  - the followers will eventually catch up and become consistent with the leader.

# Replication Lag solutions

- Approaches to **deal** with **Replication Lag** problems:
  - **Read-after-write consistency**: a user should always see data that they submitted themselves
  - **Monotonic reads**: after a user has seen the data at one point in time, they shouldn't later see the data from some earlier point in time
  - **Consistent prefix reads**: users should see the data in a state that makes causal sense, for example seeing a question and its reply in the correct order
    - This is a problem in partitioned (sharded) databases, which we will discuss later

# Replication Lag – example 1



# Read-after-write consistency

- When new data is submitted, it must be sent to the leader, but when the user views the data, it can be read from a follower
  - this is especially appropriate if data is frequently viewed, but only occasionally written

- A problem with asynchronous replication
  - the new data may have not yet reached the replica

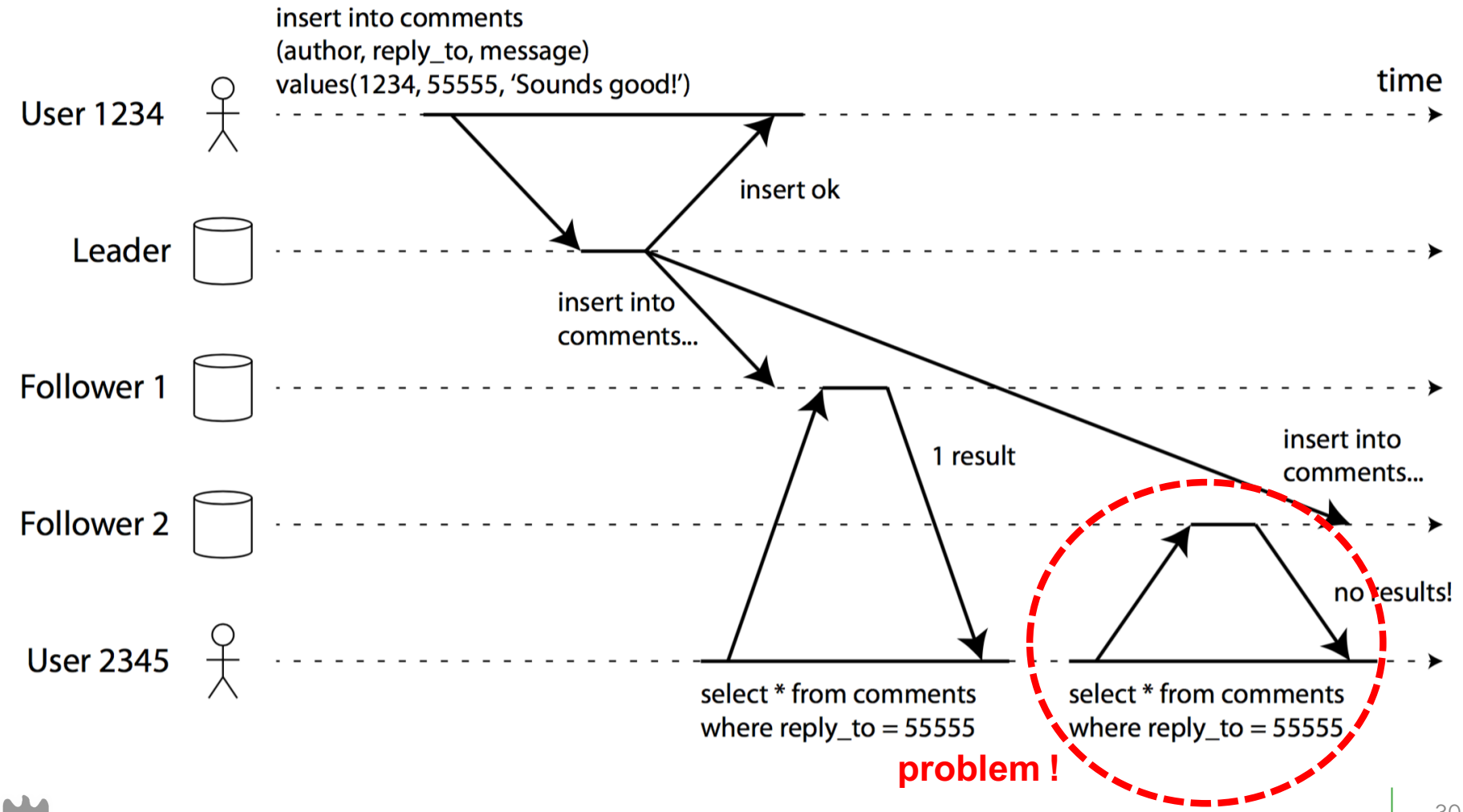
- **Solution**: read-after-write consistency
  - also known as read-your-writes consistency

# Read-after-write consistency

- **Implementation**
  - read from the leader something that the user may have modified, otherwise read it from a follower
  - requires some way of knowing what data have been modified
  - potential problem: if most things are potentially editable by the user, that approach won't be effective (negating the benefit of read scaling)
- **Criteria** to read from the leader
  - tracking the time of the last update - for *X* minute(s) after the last update, all reads are made from the leader
  - client record the timestamp of its most recent writes
    - system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp

# Replication Lag – example 2

- A user is seeing things moving backwards in time



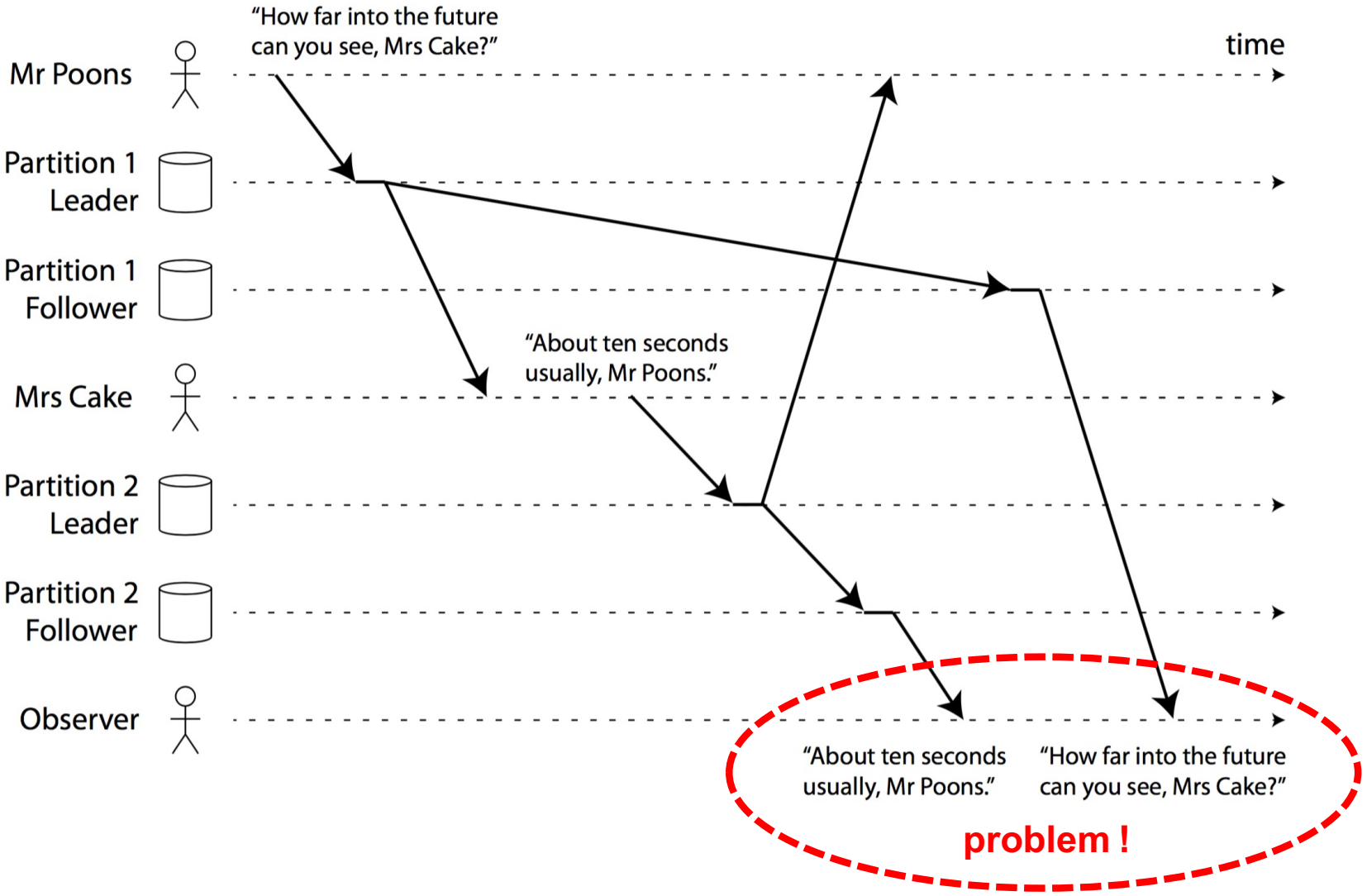
# Monotonic reads

- Problem
  - when you read data, you may see an old value
  - can happen if makes several reads from different replicas

- This scenario is quite likely if the user refreshes a web page, and each request is routed to a random server

- **Solution**: Monotonic reads
  - ensures that this kind of anomaly does not happen by making each user reading always from the same replica
  - different users can read from different replicas

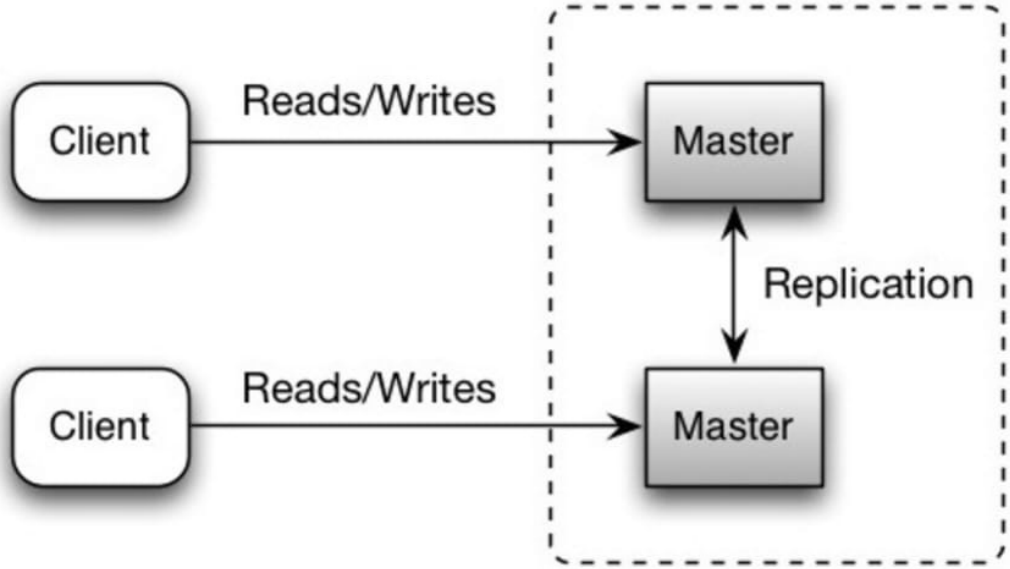
# Replication Lag – example 3



# Consistent prefix reads

- Replication lag anomalies concerns violation of causality
  - if some partitions are replicated slower than others, an observer may see the answer before they see the question
- Consistent prefix reads prevents this kind of anomaly:
  - guarantees that, if a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order
- This is a problem in partitioned (sharded) databases
  - which we will discuss later

# Multi-Leader Replication



# Multi-leader replication

- **Leader-based** replication problems:
  - all writes must go through the single leader
  - inaccessible leader => can't write to the database
- **Multi-leader** configuration
  - **more than one node can accept writes**
  - known as master-master replication or active/active
- An extension of the leader-based replication
  - replication happens in the same way: each node that processes a write must forward that data change to all the other nodes
- Each leader simultaneously acts as a follower to the other leader

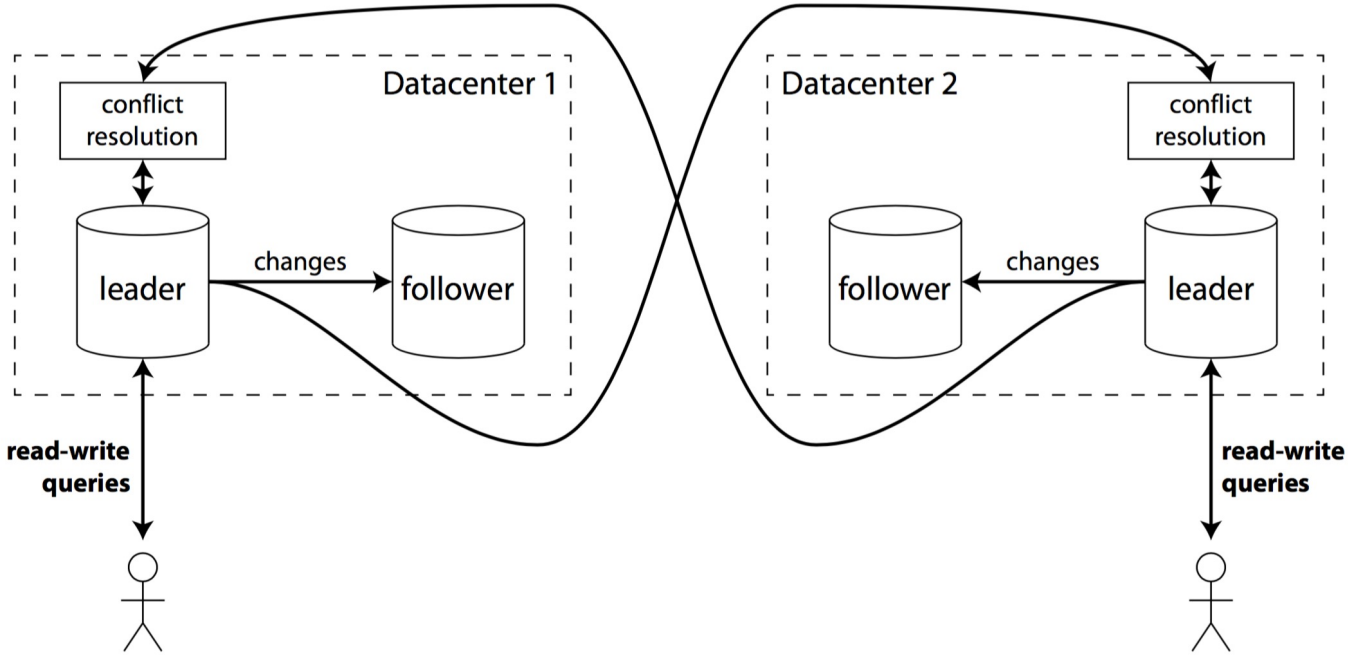
# Multi-leader replication - use cases

- **Recommended** usage:
  - Multi-datacenter operation
  - Clients with offline operation
  - Collaborative editing

- **Not recommended** within a single datacenter
  - the benefits rarely outweigh the added complexity

# 1. Multi-datacenter operation

- A database with replicas in several datacenters
  - To tolerate failure of the datacenter or to be closer to users
- Each datacenter can have a leader
  - leader-based replication is used in each datacenter
  - each leader replicates its changes to other leaders

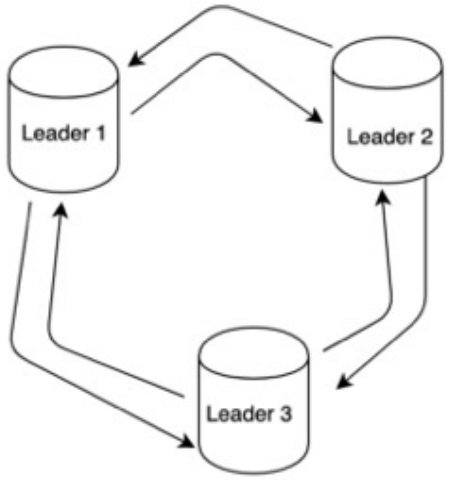


# 1. Multi-datacenter operation

- **Benefits**
  - Performance: local optimization while the inter-datacenter network delay is hidden from users
    - single-leader adds significant latency to writes between datacenters nodes
  - Tolerance to datacenter problems: multi-leader configuration allows each datacenter can continue operating independently of the other(s)
  - Tolerance to network problems: a temporary network interruption does not prevent writes being processed
- **Drawbacks**
  - **Write conflicts**: the same data may be concurrently modified
  - Auto-incrementing keys, triggers and integrity constraints can be problematic
  - Multi-leader replication is often considered much complex, and fail prone, so should be avoided if possible!

# 2. Clients with offline operation

- **Every device with a local database will act as a leader**
- Asynchronous multi-leader replication process
  - changes made in offline need to be synced with a server and other devices when the device comes again online
  - replication lag: hours or even days
- This is like the multi-leader replication between datacenters, taken to the extreme
  - the network connection between them is extremely unreliable
  - each device became a 'datacenter'

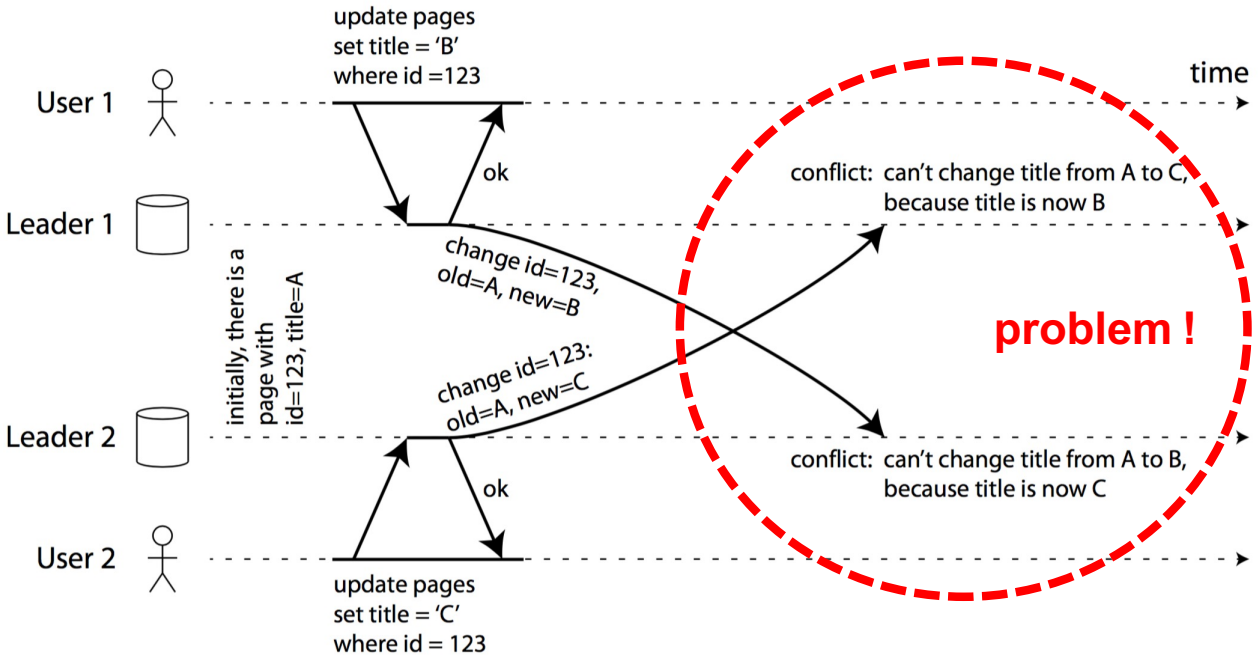


# 3. Collaborative editing

- Real-time collaborative editing applications allow several people to edit a document simultaneously
  - For example, Google Docs
- When one user edits a document
  - the changes are instantly applied to their local replica
  - and asynchronously replicated to the server and any other users who are editing the same document
- To avoid editing conflicts, the application must obtain a lock on the document before a user can edit it
- Faster collaboration: requests a unit of change very small and avoid locking
  - this allows multiple users to edit simultaneously, but it also brings all the challenges of multi-leader replication, including requiring conflict resolution

# Multi-leader Write conflicts

- The biggest problem with multi-leader replication is that write conflicts can occur
  - this problem does not occur in a single-leader
- The conflict is detected when changes are asynchronously replicated

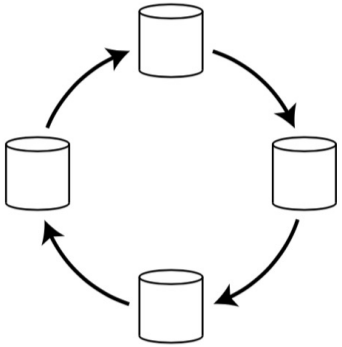


# Conflict resolution

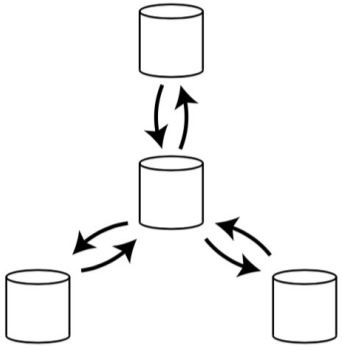
- **Convergent conflict resolution**:
  - give each write a unique ID – pick the highest as winner
    - E.g., timestamp, a technique known as **last write wins (LWW)**
  - give each data replica a unique ID - writes from the higher-numbered replica will be the winner
  - merge the values together (e.g., data concatenation)
  - record the conflict in an explicit data structure with all information, and write application code which resolves the conflict at some later time (perhaps by prompting the user)
- **Custom conflict resolution logic**
  - It use application code to resolve conflicts on read and on write operations

# Multi-leader replication topologies

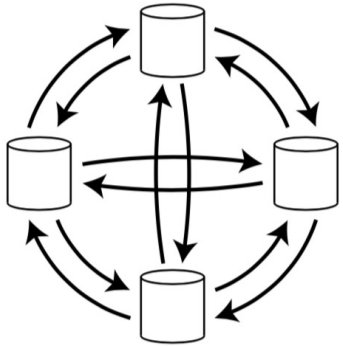
- Replication topology describes the communication paths that propagates writes from node to node
- Different topologies are possible:
  - all-to-all is the most general topology
  - circular topology is a more restricted topology (for example, MySQL support this by default)
  - star is another popular topology that can be generalized to a tree



(a) Circular topology



(b) Star topology



(c) All-to-all topology

# Leaderless Replication

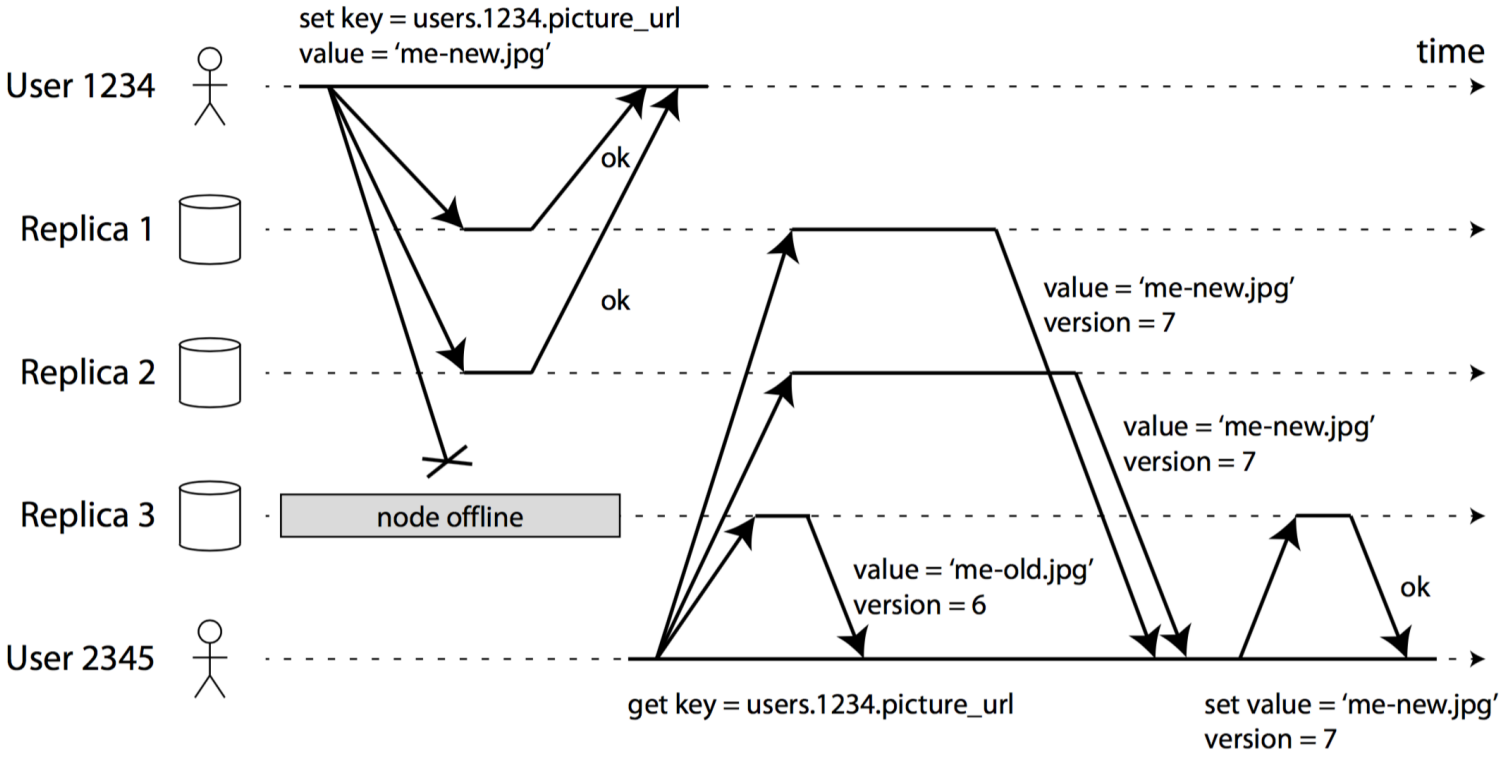
# Leaderless replication

- **No leader**
- Any replica accepts writes from clients
  - in some implementations, the client directly sends its writes to several replicas
  - in others, a coordinator node does this on behalf of the client
    - unlike a leader database, that coordinator does not enforce a particular ordering of writes.

- Used in some of the **earliest replicated data systems**
  - fashionable architecture for databases after Amazon used it for their in-house Dynamo system
    - Riak, Cassandra and Voldemort are datastores inspired by Dynamo

# Writing when a node is down

- In a leaderless configuration, there in **no failover**!
- Example: database with three replicas, and one of the replicas is currently unavailable
  - A **quorum write**, **quorum read**, and **read repair**



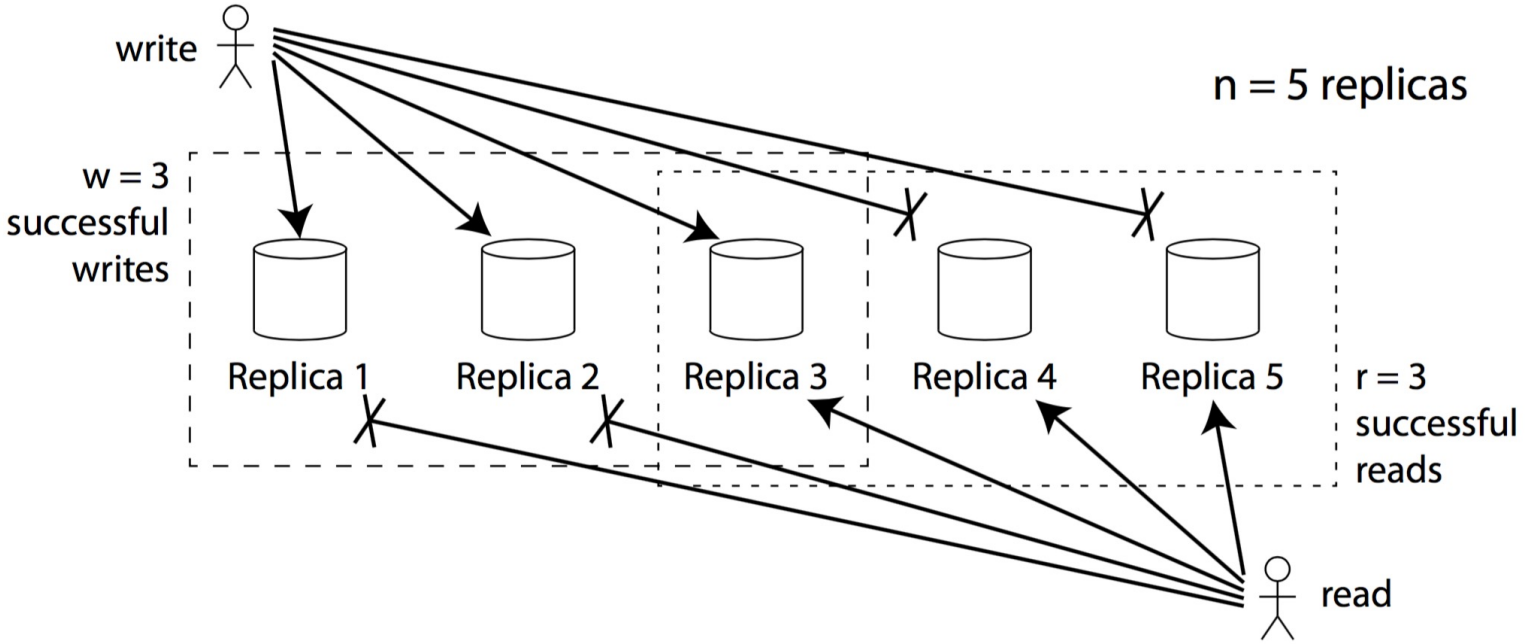
# Recovering missing writes

- A node comes back online, how does it get the missing writes?
- Two mechanisms are often used:
  - **Read repair**: client reads from several nodes in parallel. If detecting a stale responses, it writes the newer value back to that replica. Works well for values that are frequently read
  - **Anti-entropy process**: in addition, some datastores have a background process that constantly looks for differences in the data between replicas, and solves the problems
- Not all systems implement both mechanisms
  - without an anti-entropy process, values that are rarely read may be missing from some replicas and thus have reduced durability

# Quorum for reading and writing

- **Consistency Quorum**
  - architecture with **n replicas**
  - every **write must be confirmed by w nodes** to be considered successful
  - the **client must query at least r nodes** for each read
- No quorum?
  - writes or reads return an error
- **Consistency Quorum condition**: w + r > n
- Normally, reads and writes are always sent to all n replicas in parallel
- Databases with appropriate quorum can tolerate the failure of individual nodes without need for failover

# Quorums example




- A common choice is to make n an odd number (typically 3 or 5), and to set *w = r = (n + 1) / 2* (rounded up)
- However, we can vary the numbers as you see fit
  - for example, a workload with few writes and many reads may benefit from setting w = n and r = 1
  - this makes reads faster, but has the disadvantage that just one failed node causes all database writes to fail

# Without quorum consistency

- Case: $w + r \leq n$
- Reads and writes will still be sent to n nodes, but a smaller number of successful responses is required for the operation to succeed
- We are more likely to read stale values
- This configuration allows **lower latency** and **higher availability**
  - if there is a network interruption and many replicas become unreachable, there's a higher chance that you can continue processing reads and writes

# Sloppy quorum

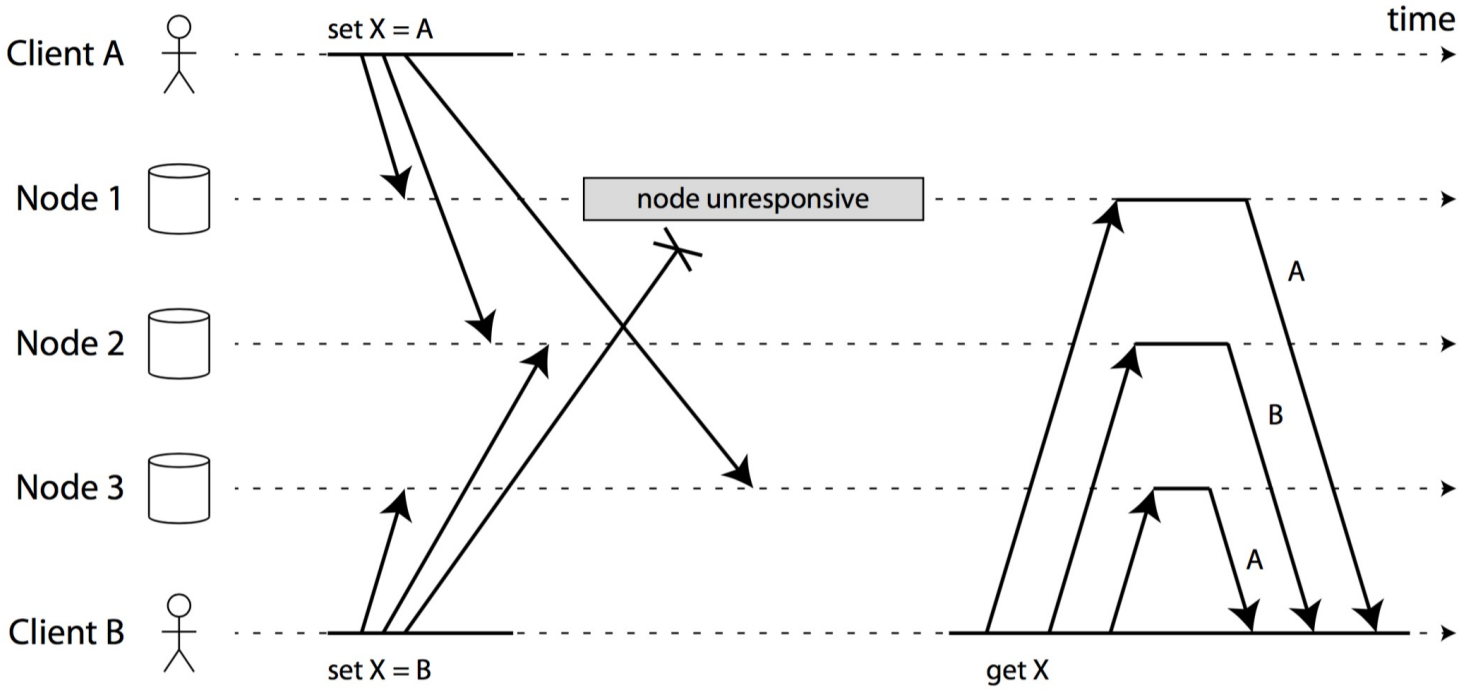
- When it is **not possible to assemble a quorum**?
  - Large clusters, network interruption, etc., a client may not be able to connect all the nodes
- In that case, database designers face a trade-off:
  - is it better to **return errors** to all requests for which we cannot reach a quorum of w or r nodes?
  - or **should we accept writes** anyway, and write them to some nodes that are reachable but aren't among the n nodes on which the value usually lives?
- The latter is known as a **sloppy quorum**
  - writes and reads still require w and r successful responses,
    - but those may include nodes that are not among the designated n "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 **hinted handoff**

# Leaderless in multi-datacenter operation

- **Leaderless replication** is also well suited for multi-datacenter
  - tolerates conflicting concurrent writes, network interruptions and latency spikes
    - E.g., Cassandra implement multi-datacenter support within the normal leaderless model – allow specifying how many replicas per datacenter
- **Each client write is sent to all replicas**, regardless of datacenter
  - the client usually only waits for acknowledgement from a quorum of nodes within its local datacenter
    - avoiding delays and interruptions on the cross-datacenter link
- The higher-latency **writes to other datacenters** are often configured to happen **asynchronously**

# Detecting concurrent writes

- Several clients can concurrently write to the same key
  - Conflicts will occur even if strict quorums are used
- Problem: events may arrive in a different order at different nodes (network delays and partial failures)



# Handling concurrent writes

- Last write wins (LWW)
- Keys with version number
- Merging concurrently written values
- Version vectors

# Handling concurrent writes

## 1. Last write wins (LWW)

- Alike LWW in multi-leader replication
- Each replica only stores the most 'recent' value
  - and allows 'older' values to be overwritten and discarded
- Example:
  - attach a timestamp to each write and pick the biggest timestamp as the most 'recent'
- There are some situations, such as caching, in which lost writes may be acceptable.
  - If losing data is not acceptable, LWW is a poor choice for conflict resolution.
- LWW is the supported conflict resolution method in Cassandra, and an optional feature in Riak

# Handling concurrent writes

## 2. Keys with version number

- The server maintains a **version number for every key**
  - incremented every time it is written
  - When a client reads a key, the server returns the value and its version number
- Clients must **read a key before writing** it
  - Clients can update an item, but only if the version number on the server side has not changed
  - If there is a version mismatch, it means that someone else (concurrent process) has modified the item before
  - the update attempt fails, because you have a stale version of the item
  - If this happens, you simply try again by retrieving the item and then attempting to update it

# Handling concurrent writes

## 3. Merging concurrently written values

- Ensures that **no data is silently dropped**
  - But it, implies that clients have to merge the concurrently written values
    - Riak calls these concurrent values **siblings**
  - Merging sibling values is essentially the same problem as conflict resolution in multi- leader replication
- Possible approaches:
  - use a simple approach like LWW or version number
  - do something more intelligent like a union to merge values
  - no delete data, leaving a marker with an appropriate version number to indicate that the item has been removed when merging siblings.
    - Such a deletion marker is known as a tombstone.

# Handling concurrent writes

## 4. Version vectors

- Algorithm for multiple replicas, but no leader
- Use a **version number per replica as well as per key**
  - This collection of version numbers is called a version vector
- Each replica increments its own version number when processing a write
  - also keeps track of the version numbers it has seen from all of the other replicas
  - it can then use that information to figure out which values to overwrite and which values to keep as siblings
  - The version vector structure ensures that it is safe to read from one replica and subsequently write back to another replica

# Summary

- **Replication**
  - a copy of the same data on several machines
- Replication can serve several purposes
  - High availability
  - Disconnected operation
  - Latency
  - Scalability
- Replication is a tricky problem
  - requires carefully thinking about concurrency and about all the things that can go wrong
  - dealing with the consequences of those faults
- Three main approaches to replication:
  - Single-leader replication
  - Multi-leader replication
  - Leaderless replication

# Summary (cont)

- Each approaches has advantages and disadvantages
  - **Single-leader replication** is popular because it is easy to understand and there is no conflict resolution to worry about
  - **Multi-leader** and **leaderless replication** can be more robust in the presence of faulty nodes, network interruptions and latency spikes — at the cost of being harder to reason about, and providing only very weak consistency guarantees
- Replication can be **synchronous** or **asynchronous**
  - option with profound effect on the system behavior when fails
  - asynchronous replication can be fast when the system is running smoothly
  - If a leader fails, recently committed data may be lost in asynchronously follower update

# Summary (cont)

- Discussed a few consistency models which are helpful for deciding how an application should behave under **replication lag**:
  - **Read-after-write consistency**: a user should always see data that they submitted themselves
  - **Monotonic reads**: after a user has seen the data at one point in time, they shouldn't later see the data from some earlier point in time
  - **Consistent prefix reads**: users should see the data in a state that makes causal sense, for example seeing a question and its reply in the correct order
- Discussed the **concurrency** issues that are inherent in **multi-leader** and **leaderless** replication approaches