TDT4225
Chapter 5 – Replication

Svein Erik Bratsberg
Department of Computer Science (IDI), NTNU

Distributed data (chap 5-9)

- *Scalability* – partitioning – spread the load across multiple machines
- *Fault tolerance* – replication – takeover when one machine fails
- *Latency* – locality to the application when data is distributed globally
- *Shared nothing* – every computer has all it needs
- *Shared disk* – many computers share disk through network
- *NUMA* – many CPUs, but *non-uniform memory architecture* – each CPU has its «local» memory

Shared-nothing architecture

- Each machine is a node having CPU, memory, disks and network interface
- Coordination between nodes done through network messages
- No special hardware needed
- You need to be aware of the constraints and trade-offs that occur in such a distributed system
- The database cannot magically hide these from you
- Beware: In some cases, a simple single-threaded program can perform significantly better than a cluster with over 100 CPU cores

Replication vs. partitioning

- **Replication**: Keeping a copy of the same data on several different nodes, potentially in different locations.
- Provides redundancy (and fault tolerance)
- **Partitioning**: Splitting a big database into smaller subsets called partitions so that different partitions can be assigned to different nodes (also known as **sharding**). Provides scalability

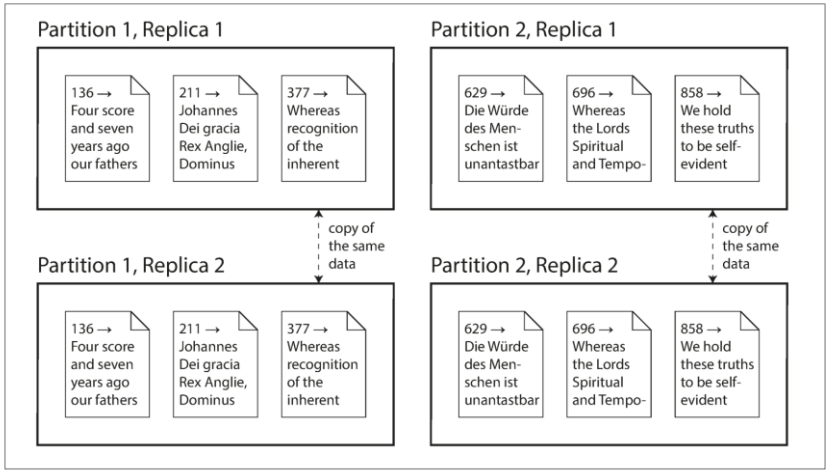

Figure II-1. A database split into two partitions, with two replicas per partition.

Replication

- To keep data geographically close to your users
- Fault tolerance – increased availability
- Scale out for read queries
- If the data that you're replicating does not change over time, then replication is easy
- Replicating changes between nodes:
  - single-leader
  - multi-leader
  - leaderless replication.
- Synchronous vs. asynchronous replication

Leader-based replication

- Also named *active/passive, master–slave, and primary/backup, primary/hot-standby*
- Writes sent to leader which writes the changes to disk
- Followers (read replicas, slaves, secondaries, or hot standbys) are sent the changes through a replication stream / change log
- Must apply all writes in the same order as they were processed on the leader
- Any replica may be read, but only master accepts writes

Leader-based replication (2)

- PostgreSQL, MySQL, Oracle Data Guard, and SQL Server's AlwaysOn Availability Groups, MongoDB, etc

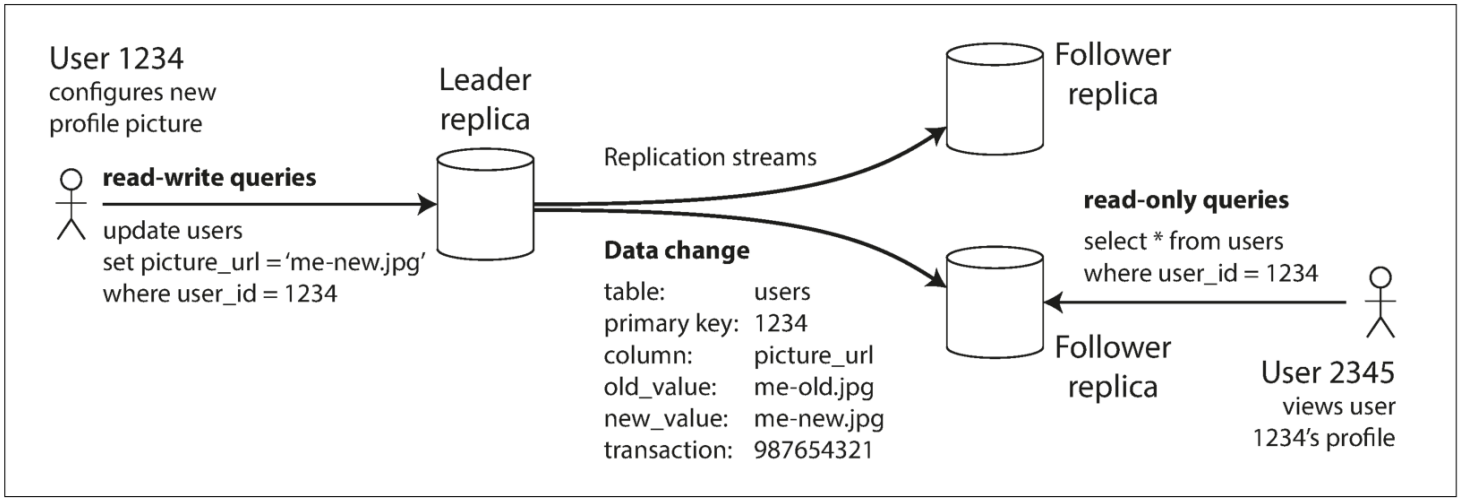

Figure 5-1. Leader-based (master–slave) replication.

Synchronous vs. Asynchronous Replication

- Is the replica update included in the client reponse (synchronous)
- More complicated failure scenarios for asynchronous

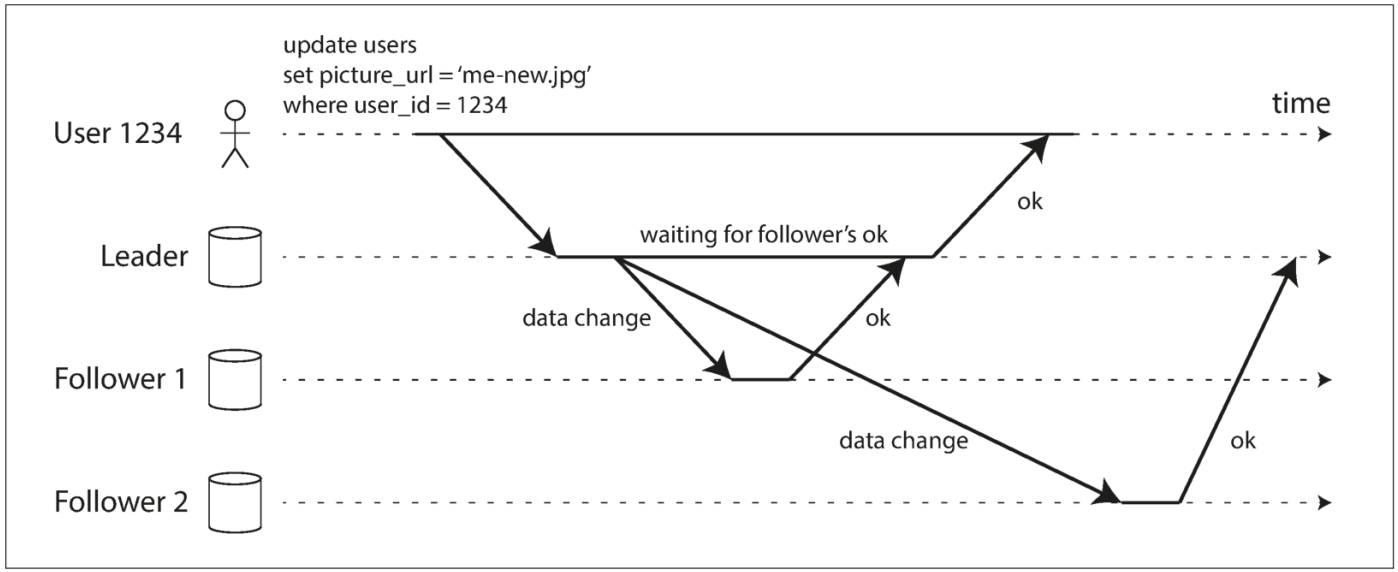

Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.

Asynchronous replication

- Often, leader-based replication is configured to be completely asynchronous
- May cause lost updates in case of crash of leader
- The leader can continue processing writes, even if all of its followers have fallen behind
- Weakened durability. But widely used anyway.
- Chain replication is a method applied by Microsoft (but created at Cornell, Rob van Renesse): Updates at head, reads at tail of replication chain

Adding new replicas (repair?)

- Take a consistent snapshot of the leader's database at some point in time
- Copy the snapshot to the new follower node
- Copy the log: all the data changes that have happened since the snapshot was taken (really: started, all alive transaction log)
- Caught up when all backlog processed
- Fully automated by some systems, but manual adm commands by some others

Takeover / Failure handling

- Failure detection – I-am-alive protocol?
- Election of new leader, or pre-determined
- Takeover – the new one is the leader
- Problems:
  - Some updates weren't received before takeover. The old leader may have «lost updates» when recovering
  - The new leader should «bump-up» anything used for identification (auto-increment keys and log sequence numbers)
  - Split-brain: 2 new nodes both think they are the leader
  - What timeout to use?
    - Too small -> false failures may be detected.
    - Too big -> long time without any leader (unavailability)

Replication logs – Statement based

- For SQL: INSERT, UPDATE, or DELETE statement is forwarded to followers
- Problems
  - nondeterministic function, NOW() or RAND()
  - autoincrementing columns and concurrent updates
  - statements with side effects (triggers etc.)
- Used in MySQL prior to v 5.1, but now row-based replication is mostly used
- VoltDB uses this, but transactions must be deterministic (through a global serial order controller)

Write-Ahead Log-shipping

- Log: append-only sequence of bytes containing all writes (and commits/aborts)
- When the follower processes this log, it builds a copy of the exact same data structures as found on the leader.
- Used in PostgresSQL and Oracle and others
- Low-level details being storage-specific. Copies must be physically equal.
- If the database changes its storage format, typically, it is not possible to run different database versions at different servers
- May be used for online upgrades to database software if the follower is allowed to upgrade its software

Logical (row-based) log replication

- Different formats on log and local storage (e.g. primary keys and not BlockIds in log)
- Inserts, deletes and updates generate different log records containing before and/or after image.
- MySQL’s binlog uses this approach and also ships commit log records
- Leader and follower may have different storage formats and storage engines
- Could provide better fault tolerance

Trigger-based replication

- When flexibility is needed
- Replicate a subset of the data
- Need conflict resolution
- Triggers and stored procedures let «application code» be run when updates appear
- The trigger may log the update into a separate change table, which again may be read by a replication process
- General approach with higher runtime cost that special purpose replication approaches

Problems with replication lag

- Leader-based replication OK for loads having mostly reads and a few writes
- And using asynchronous replication
- You may read out-of-date info from a follower: Eventual consistency
- Ranging from milliseconds out-of-date to minutes at high loads

Reading-your-writes consistency

- Write-then-read-what you've written
- Problems when reading a not-updated replica
- Need *read-after-write consistency* / *read-your-writes consistency*

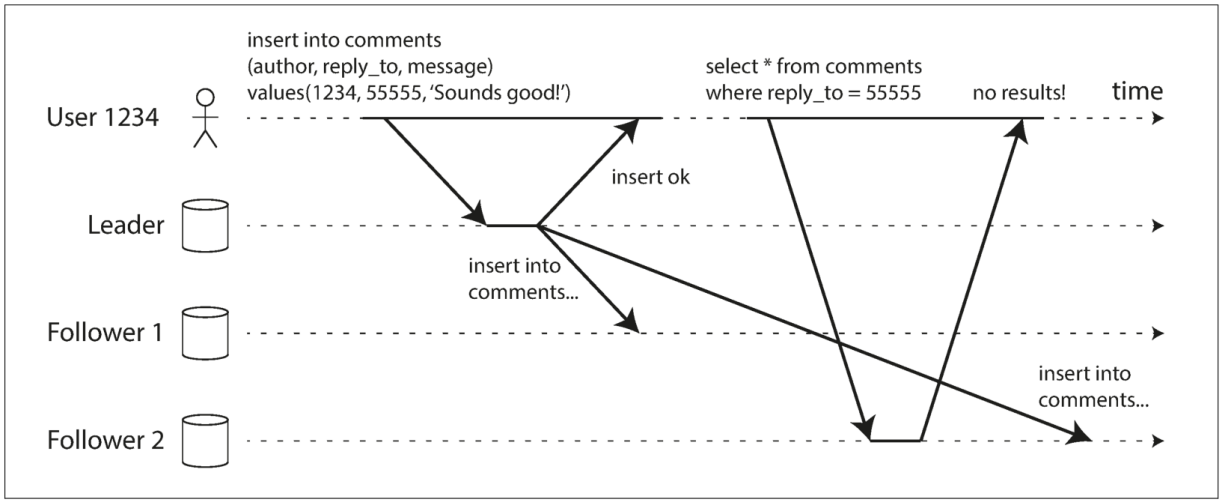

Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

Reading your write consistency (2)

- Write/Read your profile to/from *the leader*, and may read everyone else from a *follower* (social network example)
- When you have to write everywhere, use a one minute wait where you read from the leader in the meantime.
- Using logical timestamps: Register write timestamps and use this when reading replicas
- Multi-datacenters. Route to the leader of the correct datacenter
- Complicated: Routing may change over time. Different devices may connect to different datacenters.

Monotonic Reads

- Monotonic reads is a guarantee that you never read older versions of data
- May be ensured by making reads always to the same replica by hashing on the userId.

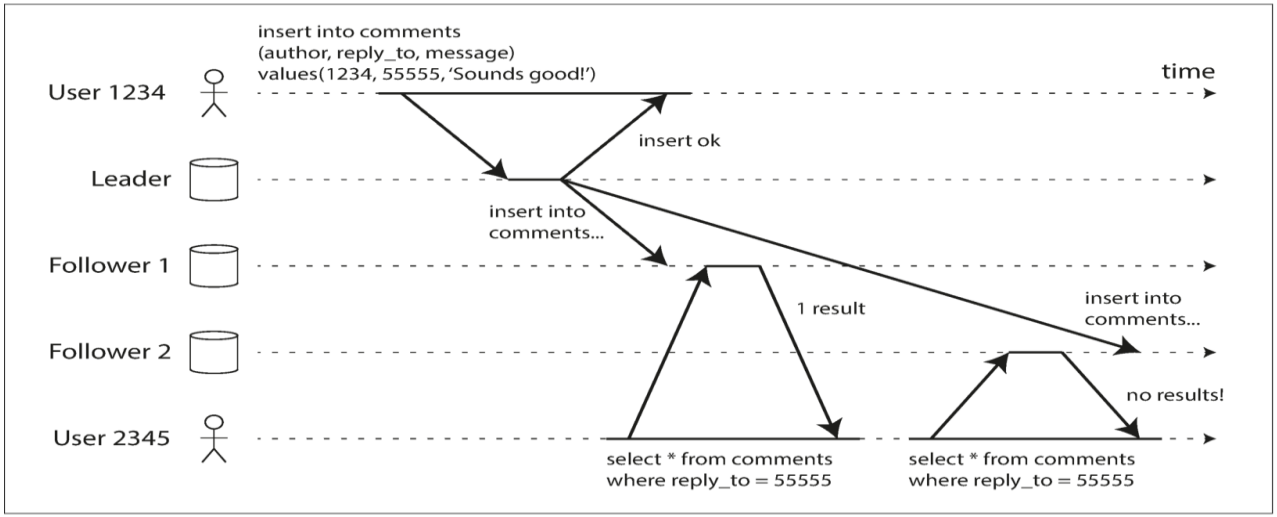

Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.

Consistent Prefix Reads

- This guarantee says that if a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order: Related data go to the same machine.

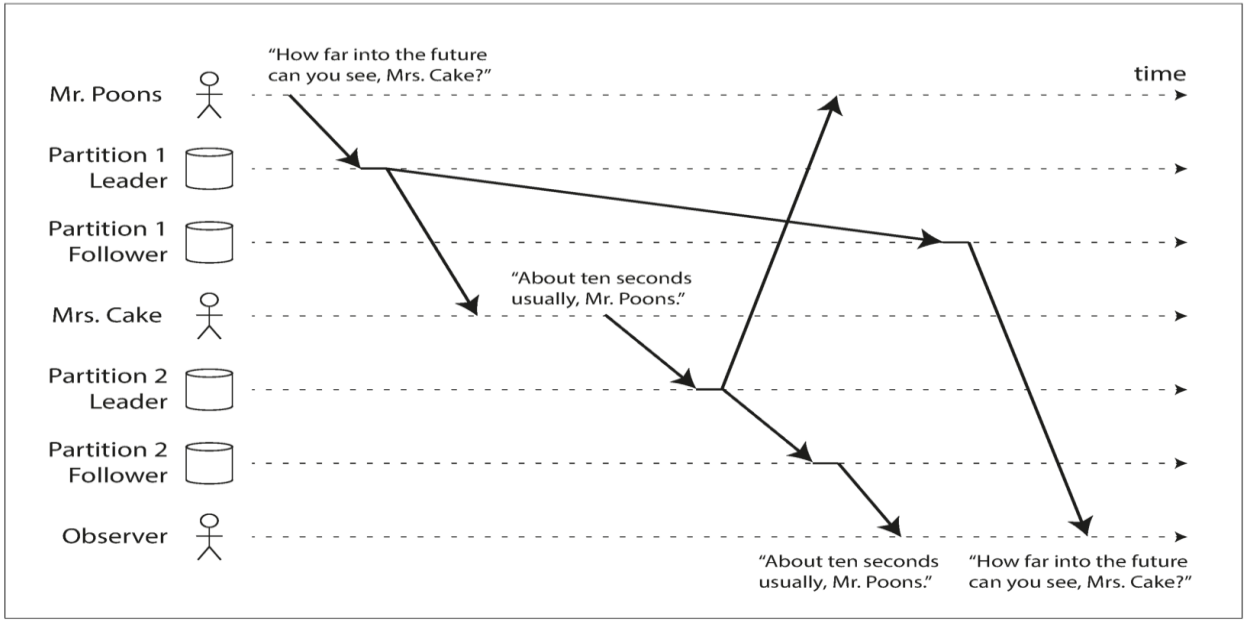

Figure 5-5. If some partitions are replicated slower than others, an observer may see the answer before they see the question.

Solutions for replication lag

- Is replication lag a real problem?
- Use better guarantees: E.g. read-your-writes
- Use transactions
- Some NoSQL databases have skipped transactions: They’re too expensive
- NewSQL databases often support distributed transactions

Multi-Leader Replication

- Allow more than one node to accept writes
- Multi-Leader, master/master or active/active replication
- A leader in each datacenter

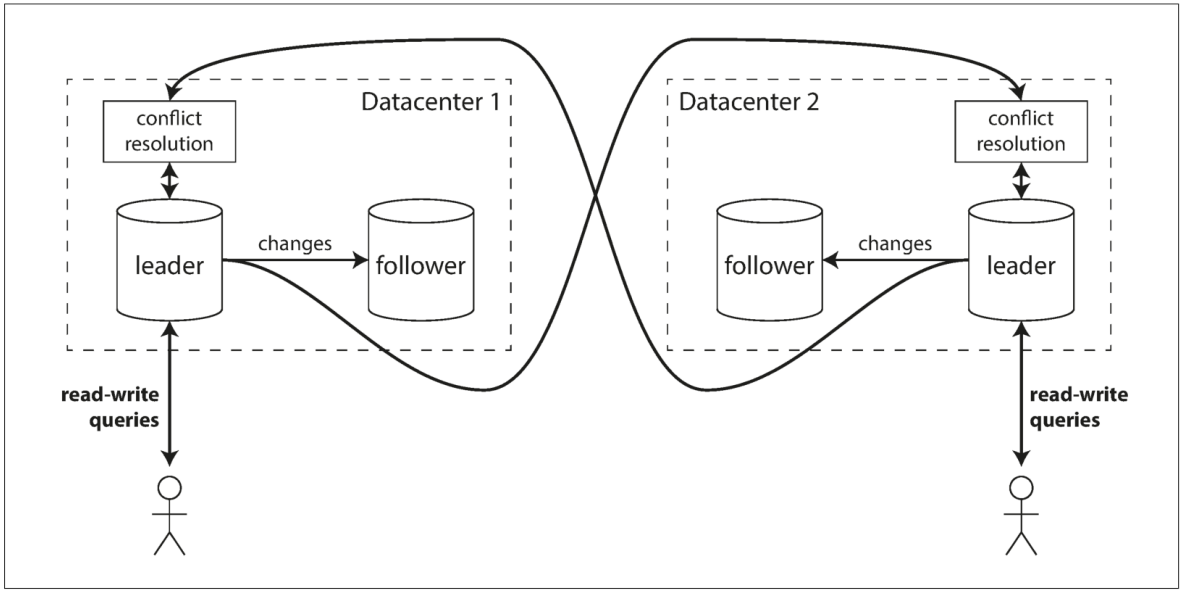

Figure 5-6. Multi-leader replication across multiple datacenters.

Multi-Leader Replication (2)

- Performance (local writes)
- Tolerance of datacenter outages
- Tolerance of network problems
- May cause concurrent update to the same data: Thus, needs conflict resolution
- Disconnected databases, e.g. calendar, have the same problem
- Collaborative editing (Google docs, Office365, etc)

Handling write conflicts

- The biggest problem with multi-leader replication

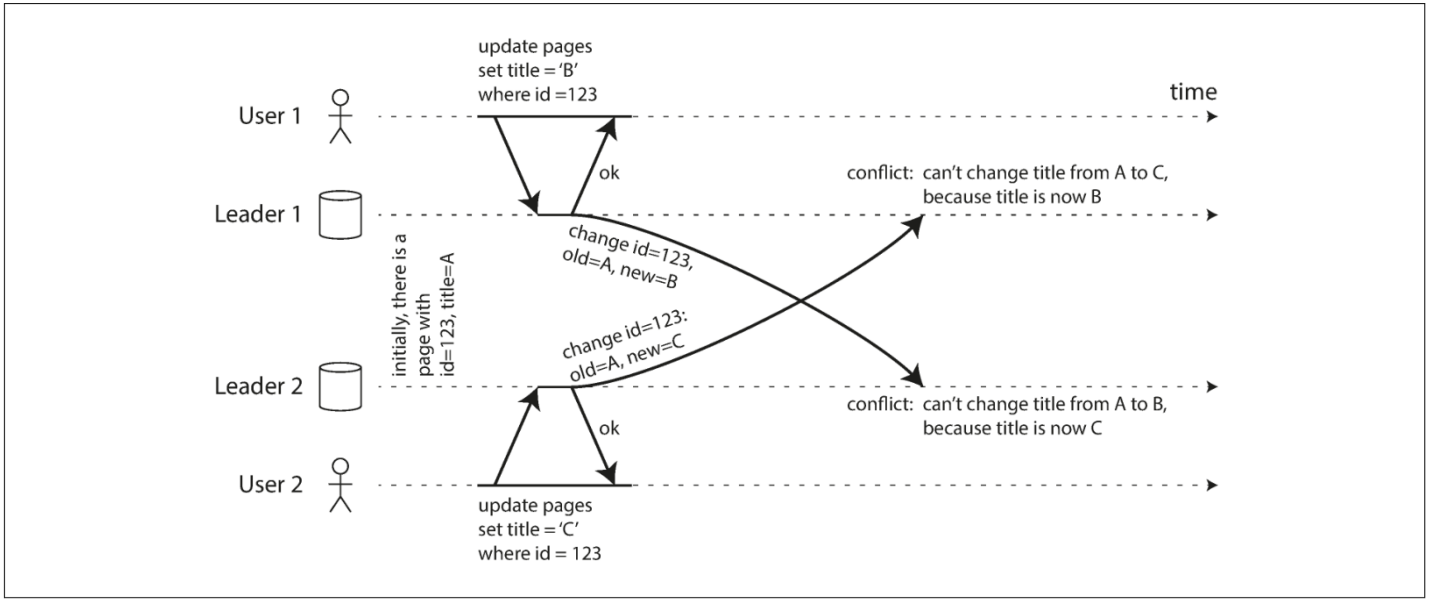

Figure 5-7. A write conflict caused by two leaders concurrently updating the same record.

Handling write conflicts

- Synchronous versus asynchronous conflict detection
- Conflict avoidance: Allow updates only at one site for each data item
- Converging to a consistent state:
- There is no «correct» values when you have write conflicts, however, consistent values are the goal:
  - Each write has a unique ID, e.g., a timestamp, last write wins (lww)
  - Each replica has a unique ID, and higher ID “wins”
  - Merge values and keep all of them (B and C in the example)
  - Apply conflict resolution to them (automatic or manual)

Conflict resolution

- *On write*: As soon as the database system detects a conflict in the log of replicated changes, it calls the conflict handler.
- *On read*: When a conflict is detected, all the conflicting writes are stored. All are read and a flag is set and the user resolves and writes back the correct value (CouchDB)
- *Conflict-free replicated datatypes* (CRDTs): Standard data structures to be used concurrently without locking
- *Mergeable persistent data structures*: Utilizes history and uses 3-way merge
- *Operational transformation*: Concurrent editing of documents (Google docs)

Multi-Leader Replication Topologies

- A replication topology describes the communication paths along which writes are propagated
- Stop propagating a message when it is received at the sender

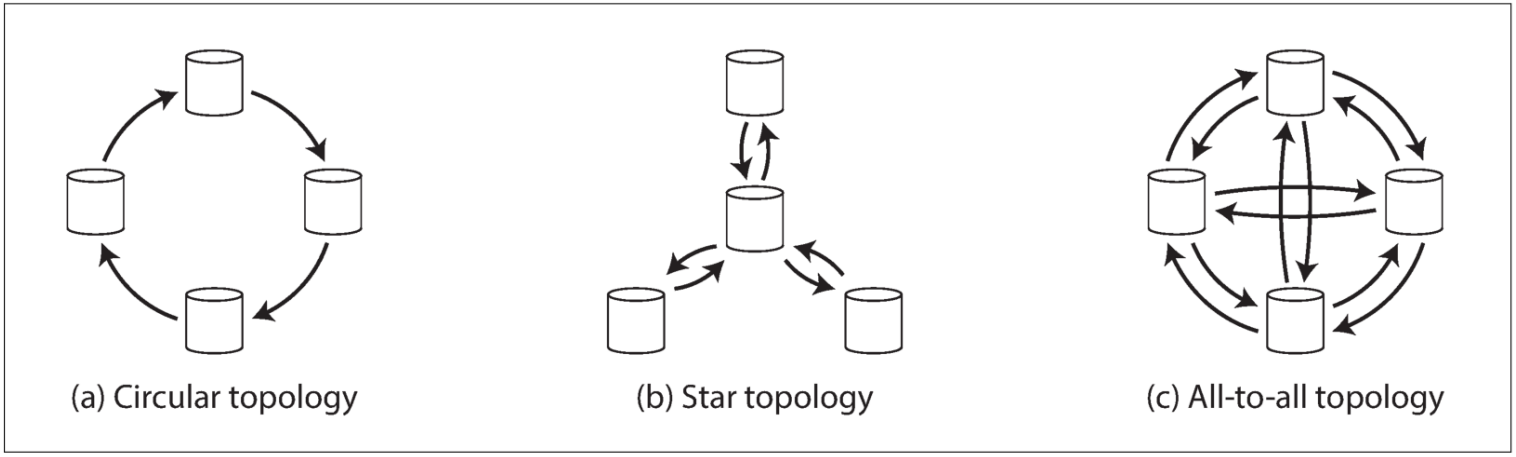

Figure 5-8. Three example topologies in which multi-leader replication can be set up.

Multi-Leader Replication Topologies (2)

- Wrong order of messages received
- May be «solved» by version vectors

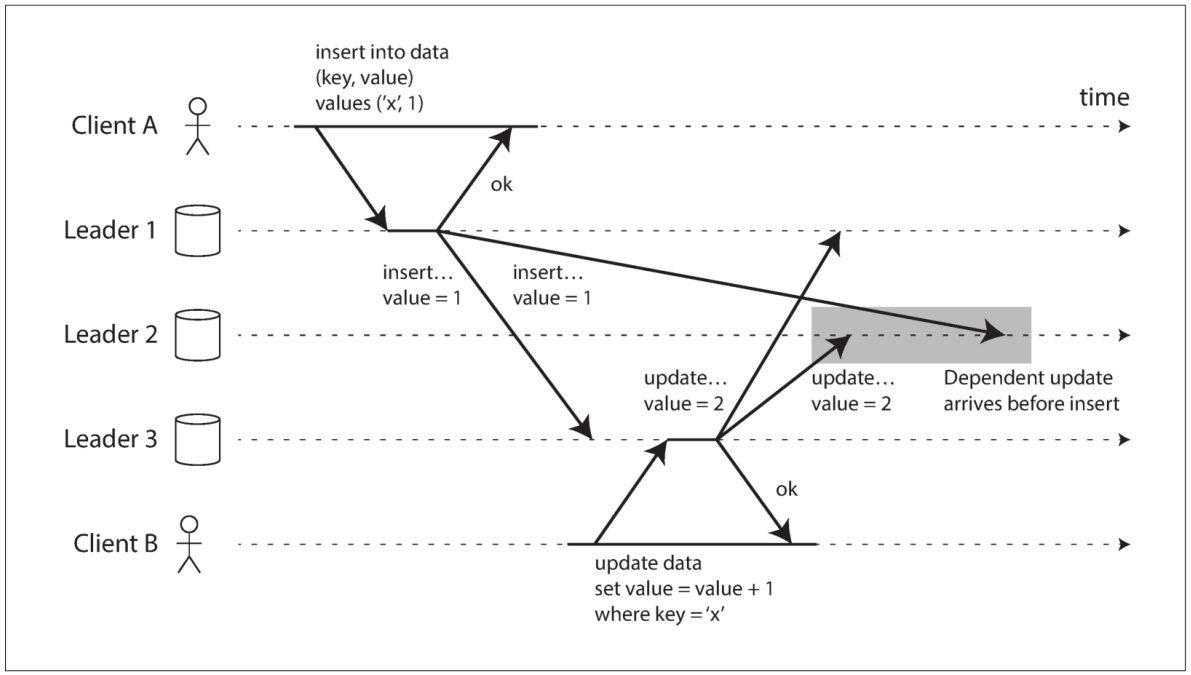

Figure 5-9. With multi-leader replication, writes may arrive in the wrong order at some replicas.

Leaderless replication

- Dynamo-style of replication
- Quorums, read repair and anti-entropy

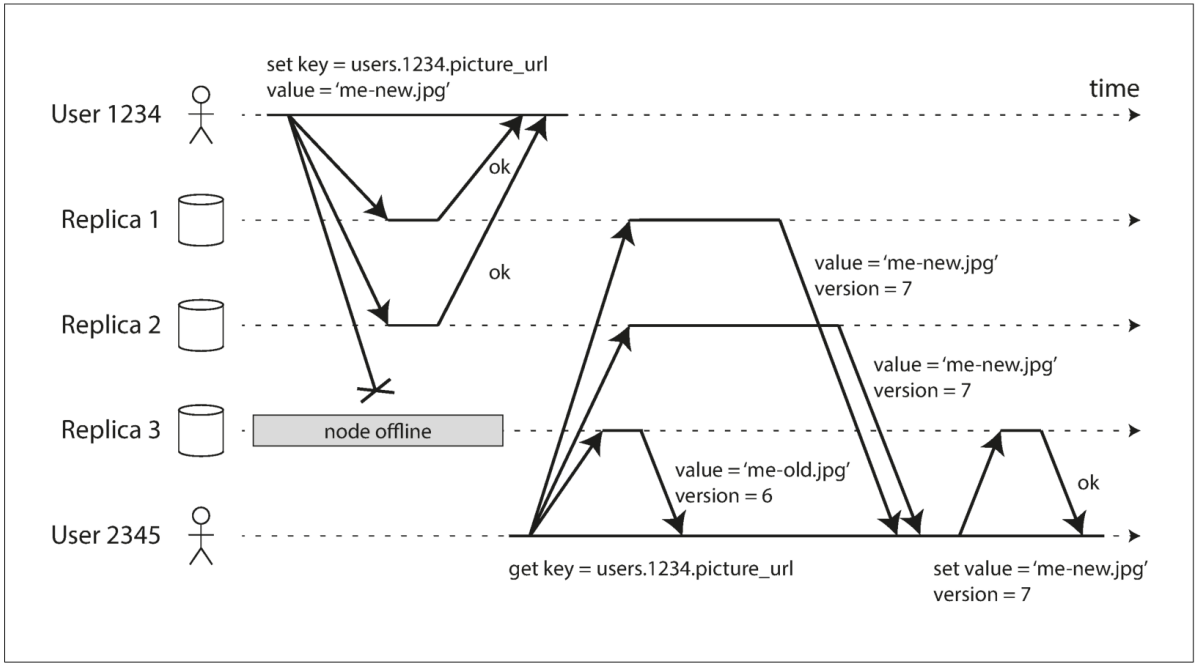

Figure 5-10. A quorum write, quorum read, and read repair after a node outage.

Quorums

- **n** replicas, every write must be confirmed by **w** nodes
- Query at least **r** nodes for each read
- As long as **w + r > n**, we expect to get an up-to-date value
- Dynamo-style databases, the parameters **n**, **w**, and **r** are typically configurable
- w = r = (n + 1) / 2 (rounded up): Common approach
- w = n and r = 1: Fast reads but all nodes are written

Quorums (2)

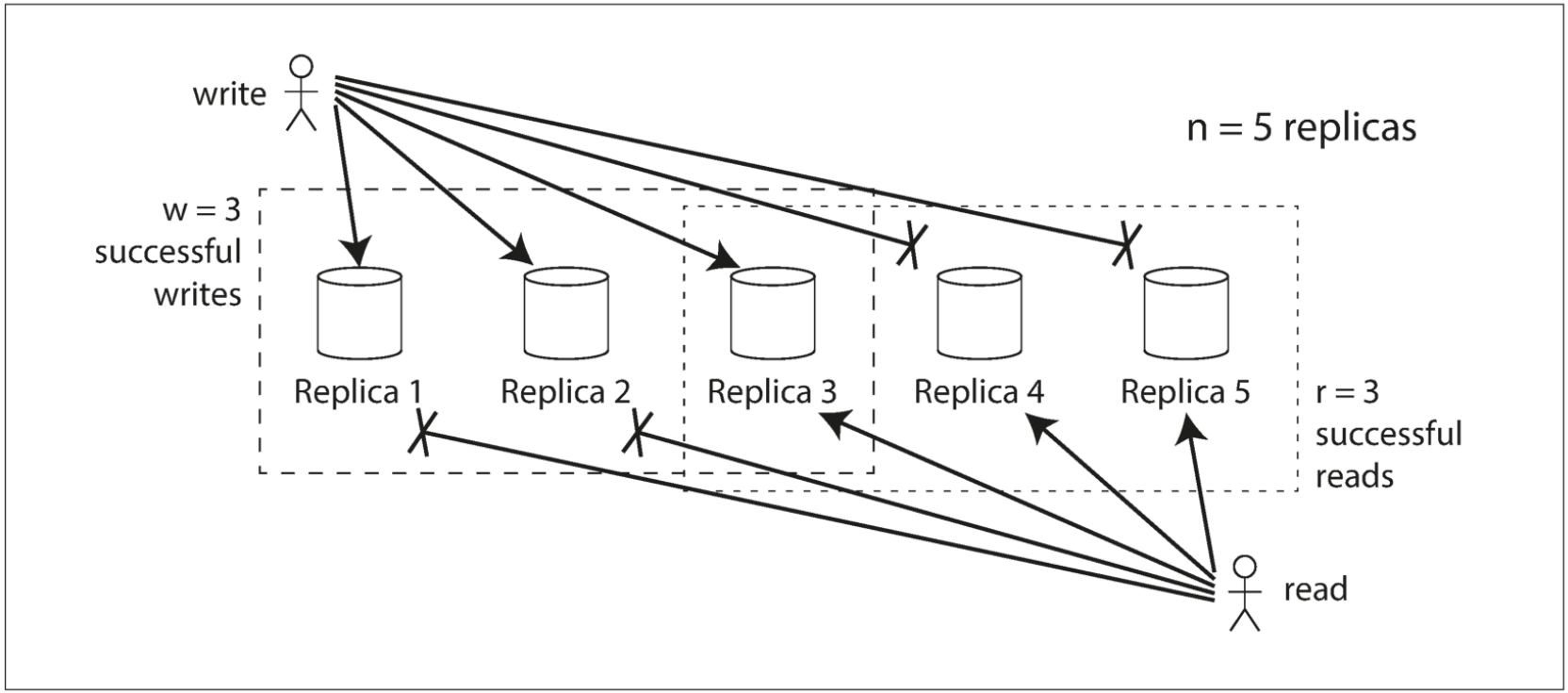

Figure 5-11. If w + r > n, at least one of the r replicas you read from must have seen the most recent successful write.

Quoroms (3)

- W + R < N may also be valid, giving better response times and less overhead, but you may read stale values
- Sloppy quoroms may give that read and write quoroms may be different nodes (no overlap)
- Concurrent writes may have ordering problems, they may be in conflict. Merge conflicts? Last-Write-Wins?
- Concurrent writes and reads may return new or old values
- It should be a pre-defined order of the replicas, primary-backup1-backup2 etc (not leaderless)
- Monitoring staleness? Quantify «eventual» consistency?

Sloppy Quorums and Hinted Handoff

- Dynamo-style systems, with some unreachable nodes
- We should 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.
- 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
- A sloppy quorum is a durability assurance
- Sloppy quoroms are optional in Riak, Voldemort and Cassandra

Multi-datacenter support

- Multi-leader replication is used in multi-datacenters. Why?
- Cassandra and Voldemort: Writes to all replicas, but usually awaits only response from local replicas
- Riak replicates locally, and uses asynch replication for multi-datacenters

Detecting Concurrent Writes

- Problem: Events may arrive in a different order at different nodes, due to variable network delays and partial failures.

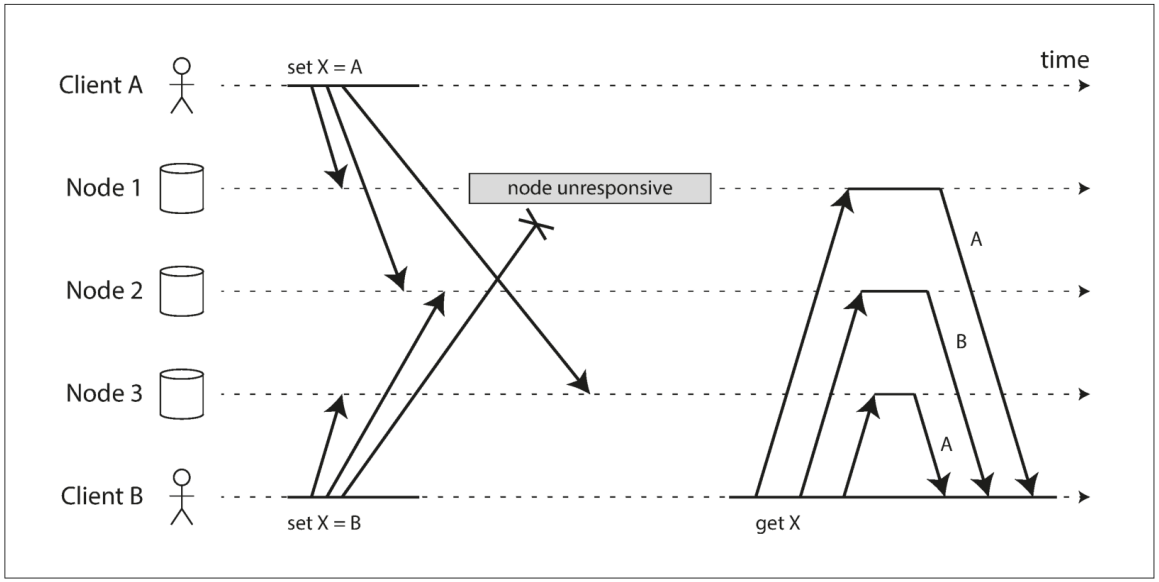

Figure 5-12. Concurrent writes in a Dynamo-style datastore: there is no well-defined ordering.

Solutions to concurrent writes

- Last Write Wins - LWW
- Some way of unambiguously determining which write is more “recent”
- We can attach a timestamp to each write, pick the biggest timestamp as the most “recent,” and discard any writes with an earlier timestamp.

Causality: Happens before and concurrent

- An operation B depends on another operation A, when the user/transaction has read A's value.
- A **happens-before** B, or B is causally dependent on A.
- An operation A happens before another operation B if B knows about A, or depends on A, or builds upon A in some way.
- Two operations are **concurrent** if neither happens before the other.
- Exact time doesn't matter: we simply call two operations concurrent if they are both unaware of each other

Capturing the happens-before relationship

- Single-node case: Two clients adding items to the same shopping cart

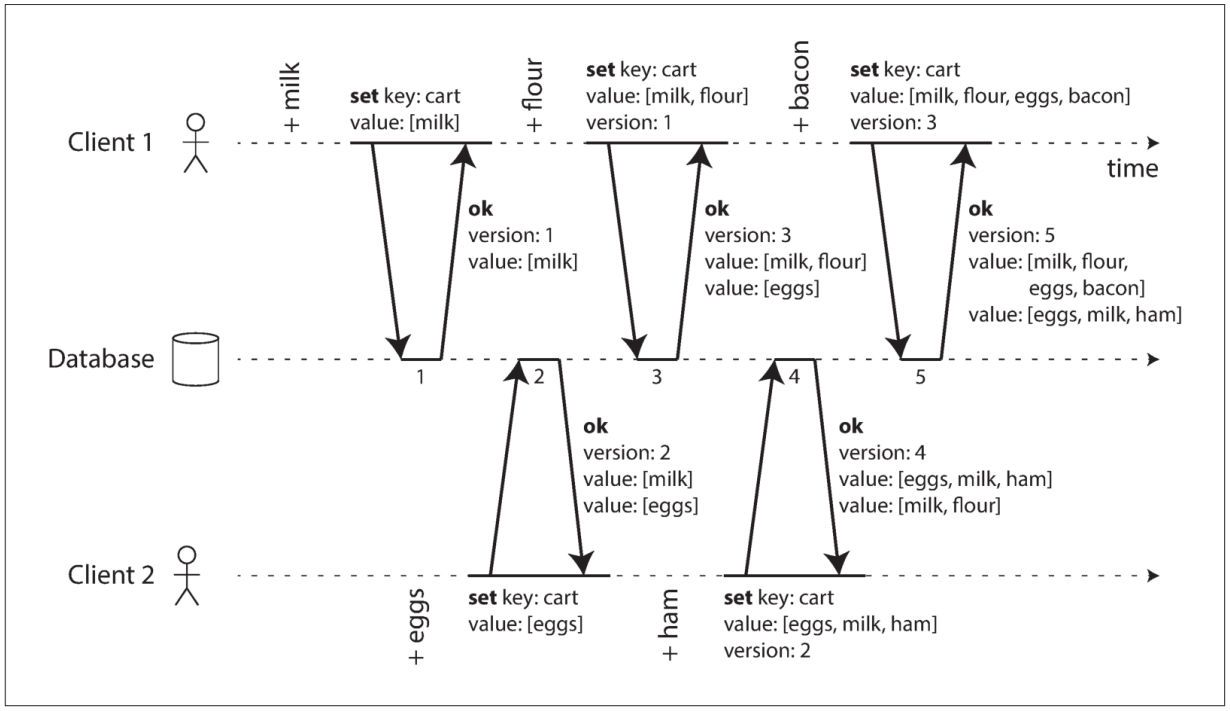

Figure 5-13. Capturing causal dependencies between two clients concurrently editing a shopping cart.

Dataflow in the example

- The database returns multiple values (*siblings*), which the client may merge in.

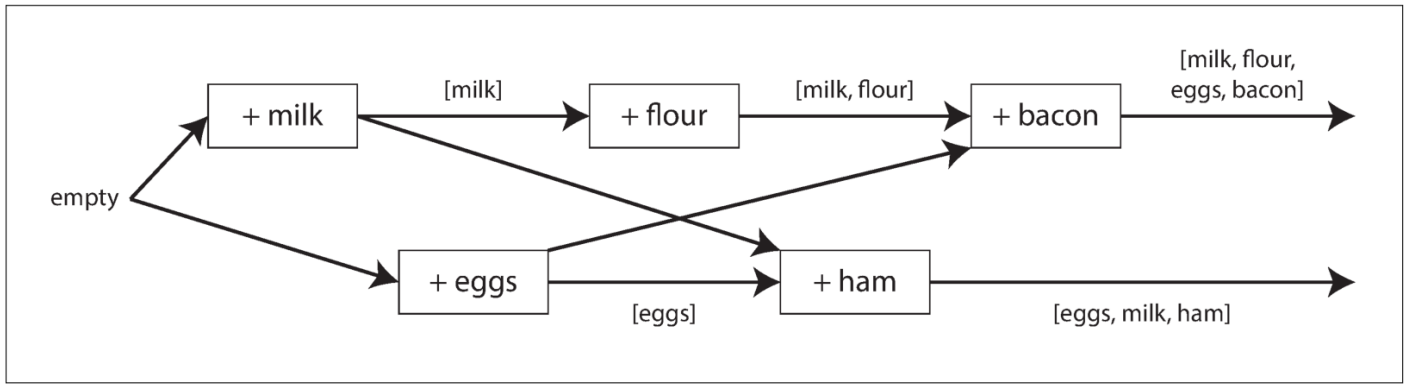

Figure 5-14. Graph of causal dependencies in Figure 5-13.

- Must use *tombstones* for deletes
- *Version vectors (multiple replicas)* and *vector clocks (multiple nodes)*

Algorithm to decide overwrite or «concurrent»

- Server maintains version number for every key, increments version number at writes
- At read, the version number is returned. A client must read before writing.
- When writing, it must include the version number of the read.
- When the server receives a write with a (read) version number, it can overwrite all values with that read version number and lower.
  But it must keep all values with a higher version number (they are «concurrent»).

Summary – Chapter 5

- Why replication?
  - High availability
  - Disconnected operation
  - Latency
  - Scalability
- Types of leadership
  - Single-leader replication
  - Multi-leader replication
  - Leaderless replication
- Types of replication
  - Synchronous
  - Asynchronous