Designing Data-Intensive Applications

Martin Kleppmann

November 14, 2023

Contents

1 Storage and Retrieval

1.1 Data Structures That Power Your Database

1.1.1 Hash Indexes

1.1.2 SSTables and LSM-Trees

2 Replication

2.1 Leaders and Followers

2.1.1 Setting Up New Followers

1. Take a consistent snapshot of the leader's database at some point in time
2. Copy the snapshot to the new follower node
3. The follower connects to the leader and requests all the data changes that have happened since the snapshot was taken. This requires that the snapshot is associated with an exact position in the leader's replication log
4. When the follower has processed the backlog of data changes since the snapshot, we say it has **caught up**

2.1.2 Handling Node Outages

1. Follower failure: Catch-up recovery
2. Leader failure: Failover **Failover**: one of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader.

   (a) *Determining that the leader has failed*.

   (b) *Choosing a new leader*.

   (c) *Reconfiguring the system to use the new leader*.

   Failover is fraught with things that can go wrong:

(a) If asynchronous replication is used, the new leader may not have received all the writes from the old leader before it failed. If the former leader rejoins the cluster after a new leader has been chosen, what should happen to those writes? The new leader may have received conflicting writes in the meantime. The most common solution is for the old leader's unreplicated writes to simply be **discarded**, which may violate clients' durability expectations.

(b) Discarding writes is especially dangerous if other storage systems outside of the database need to be coordinated with the database contents. For example, in one incident at GitHub, an out-of-date MySQL follower was promoted to leader. The database used an autoincrementing counter to assign primary keys to new rows, but because the new leader's counter lagged behind the old leader's, it reused some primary keys that were previously assigned by the old leader. These primary keys were also used in a Redis store, so the reuse of primary keys resul- ted in inconsistency between MySQL and Redis, which caused some private data to be disclosed to the wrong users.

(c) In certain fault scenarios, it could happen that two nodes both believe that they are the leader. This situation is called **split brain**.

(d) What is the right timeout before the leader is declared dead? A longer timeout means a longer time to recovery in the case where the leader fails. However, if the timeout is too short, there could be unnecessary failovers.

2.1.3 Implementation of Replication Logs

1. Statement-based replication In the simplest case, the leader logs every write request (*statement*) that it executes and sends that statement log to its followers.

   Problems:

   (a) Any statement that calls a nondeterministic function, such as `NOW()` to get the current date and time or `RAND()` to get a random number, is likely to generate a different value on each replica.

   (b) If statements use an autoincrementing column, or if they depend on the existing data in the database (e.g., `UPDATE ... WHERE <some condition>`), they must be executed in exactly the same order on each replica, or else they may have a differ- ent effect.

This can be limiting when there are multiple concurrently executing transactions.

(c) Statements that have side effects (e.g., triggers, stored procedures, user-defined functions) may result in different side effects occurring on each replica, unless the side effects are absolutely deterministic.

MySQL now switches to row- based replication (discussed shortly) if there is any nondeterminism in a statement.

2. Write-ahead log (WAL) shipping Usually every write is appended to a log:

   - For log-structured storage engine, the log is the main place for storage
   - For B-tree, which overwrites individual disk blocks, every modification is first written to a write-ahead log so that the index can be restored to a consistent state after a crash

   We can use the exact same log to build a replica on another node: besides writing the log to disk, the leader also sends it across the network to its followers.

   Main con: a WAL contains details of which bytes were changed in which disk blocks, which makes replication closely coupled to the storage engine. If the database changes its storage format from one version to another, it is typically not possible to run different versions of the database on the leader an the followers.

   That may seem like a minor implementation detail, but it can have a big operational impact. If the replication protocol allows the follower to use a newer software version than the leader, you can perform a zero-downtime upgrade of the database software by first upgrading the followers and then performing a failover to make one of the upgraded nodes the new leader. If the replication protocol does not allow this version mismatch, as is often the case with WAL shipping, such upgrades require downtime.

3. Logical (row-based) log replication A logical log for a relational database is usually a sequence of records describing writes to database tables at the granularity of a row.

4. Trigger-based replication

2.2 Problems with Prelication Lag

Leader-based replication requires all writes to go through a single node, but read-only queries can go to any replica. For workloads that consist of mostly reads and only a small percentage of writes, this is attractive: create many followers, and distribute the read requests across those followers.

This *read-scaling* architecture only realistically works with asynchronous replication, and follower may have out-dated data. This is *eventual consistency*.

2.2.1 Reading Your Own Writes

In this situation, we need *read-after-write consistency*, also known as *read-your-writes consistency*.

How can we implement read-after-write consistency in a system with leader-based replication? There are various possible techniques. To mention a few:

- When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower.
- If most things in the application are potentially editable by the user, you could track the time of the last update and, for one minute after the last update, make all reads from the leader. You could also monitor the replication lag on followers and prevent queries on any follower that is more than one minute behind the leader.
- The client can remember the timestamp of its most recent write—then the system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp. The timestamp could be a **logical timestamp** or the actual system clock.

Another complication arises when the same user is accessing your service from multiple devices,for example a desktop web browser and a mobile app. In this case you may want to provide cross-device read-after-write consistency: if the user enters some information on one device and then views it on another device, they should see the information they just entered. In this case:

- Approaches that require remembering the timestamp of the user's last update become more difficult. This metadata will need to be centralized.

- If your replicas are distributed across different datacenters, there is no guarantee that connections from different devices will be routed to the same datacenter.

2.2.2 Monotonic Reads

It's possible for a user to see things *moving backward in time*.

This happens if a user makes several reads from different replicas.

Monotonic reads is a guarante that this kind of anomaly does not happen. It's weaker than strong consistency but stronger than eventual consistency.

One way of achieving monotonic reads is to make sure that each user always makes their reads from the same replica.

2.2.3 Consistent Prefix Reads

Consistent prefix reads 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 particular problem in partitioned (sharded) databases.

One solution is to make sure that any writes that are causally related to each other are written to the same partition—but in some applications that cannot be done efficiently.

2.2.4 Solutions for Replication Lag

2.3 Multi-Leader Replication

Leader-based replication has one major downside: there is only one leader, and all writes must go through it.

A natural extension of the leader-based replication model is to allow more than one node to accept writes. Replication still happens in the same way: each node that processes a write must forward that data change to all the other nodes. We call this a multi-leader configuration (also known as master–master or active/active replication). In this setup, each leader simultaneously acts as a follower to the other leaders.

2.3.1 Use Cases for Multi-Leader Replication

1. Multi-datacenter operation In a multi-leader configuration, you can have a leader in *each* datacenter.

Downside: the same data may be concurrently modified in two different datacenters, and those write conflicts must be resolved.

2. Clients with offline operation Another situation in which multi-leader replication is appropriate is if you have an application that needs to continue to work while it is disconnected from the internet.

3. Collaborative editing

2.3.2 Handling Write Conflicts

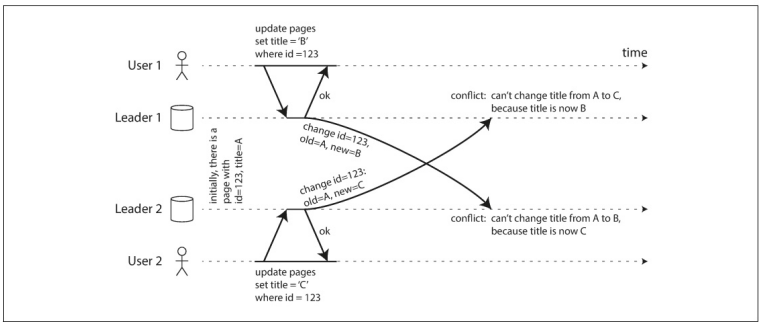

Figure 1: A write conflict caused by two leaders concurrently updating the smae record

1. Synchronous versus asynchronous conflict detection In a multi-leader setup, both writes are successful, and the conflict is only detected asynchronously at some later

   You could make the conflict detetion synchronous - i.e., wait for the write to be replicated to all replicas before telling the user that the write was successfull. However, by doing so, you would lose the main advantage of multi-leader replication: allowing each replica to accept writes independently.

2. Conflict avoidance

3. Converging toward a consistent state

4. Custom conflict resolution logic

2.3.3 Multi-Leader Replication Toplogies

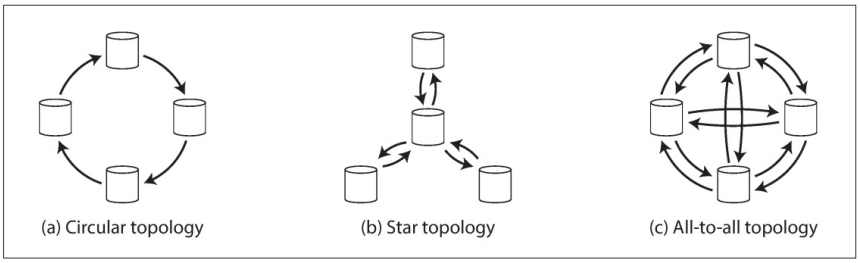

Figure 2: Three example topologies in which multi-leader replication can be set up

A problem with circular and star topologies is that if just one node fails, it can interrupt the flow of replication messages between other nodes, causing them to be unable to communicate until the node is fixed.

All-to-all topoligies can have issues. In particular, some network links may be faster than others.

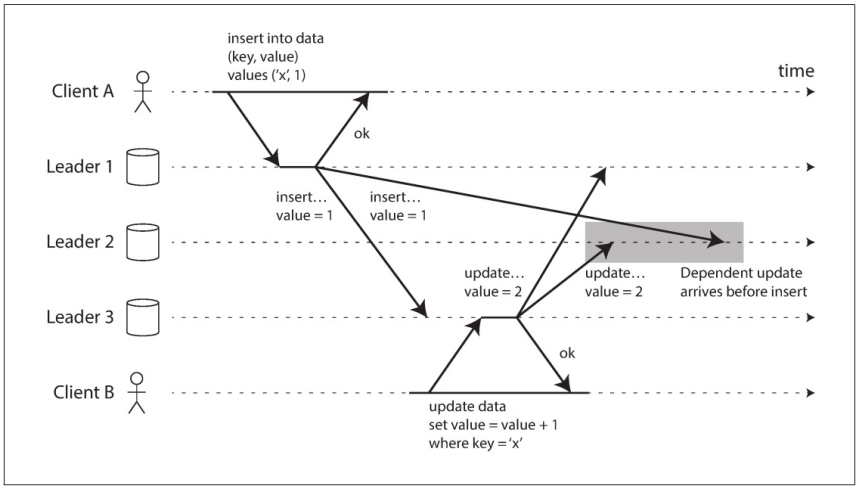

Figure 3: With multi-leader replication, writes may arrive in the wrong order at some replicas

This is a problem of causality. To order these events correctly, a technique called **version vectors** can be used.

2.4 Leaderless Replication

2.4.1 Leaderless Replication