Chapter 5 Replication

Reasons to replicate data

1. Performance / Reduce latency - data geographically close to your users
2. Increase availability - any failures
3. Increase read throughput - scalability

All of the difficulties in replication lies in **handling changes to replicated data.**

Leaders and Followers

Each node stores a copy of the database is called a replica.

- Leader-based replication
- active/passive replication
- Master-slave replication

How it works

1. One of the replicas is designated as a leader.
    a. Clients write data to the leader
    b. Sends replication log or change stream to followers
2. Other replicas are followers(read replica, slaves, secondaries, or hot standbys)
    a. Take logs from leader and update local data copy
3. When read, clients can query leader or any followers

This is a build-in feature in many:

- Relational databases - MySQL
- Non-relational databases - MongoDB
- Distributed message brokers - Kafka, RabbitMQ highly available queues
- Network filesystems
- Replicated block devices - DRBD

Synchronous VS Asynchronous Replication

Replication happens synchronously or asynchronously is a configurable option.
For example, some systems enable synchronous replication by having one synchronous follower and others as asynchronous followers. At least 2 nodes have up-to-date copies of data.

Synchronous replication
Advantage: Guaranteed data consistency with leader; No data discrepancy when leader failover.
Disadvantage: Cannot process write if follower crashes, network fault or other reasons.

If the synchronous follower is unavailable or slow, one of the asynchronous followers is made synchronous. This configuration is called semi-synchronous.

Completely asynchronous
Advantage: Leader can process write even if all followers fallen behind,
Disadvantage: A write is not guaranteed to be durable.

Setting Up New Followers

Steps:
1. Take a consistent snapshot of the leader’s database at some point of time.
2. Copy the snapshot to the new follower node
3. The follower requests all data changes from the leader from snapshot taken time and current time. The snapshot is associated with an exact position in the leader's replication log.
4. Now the follower caught up

Handling Node Outages

Follower failure: Catch-up recovery

On its local disk, each follower keeps a log of the data changes it received from the leader.
If it crashes and then recovers, it requests data from the leader, from the last transaction before crash and current time.

Leader failure: Failover

Steps:
1. Determine that the leader failed. Timeout / No response for some period of time then it is assumed to be dead.
2. Choose a new leader. Election process is that the leader is chosen by a majority of the remaining replicas. The best candidate is the replica with the most up-to-date data changes from the old leader. Getting all the nodes to agree on a new leader is a consensus problem.
3. Reconfigure the system to use the new leader. Clients send write requests to the new leader; Make the old leader come back as a follower.

Things may go wrong:
- For asynchronous replication. When the old leader comes back, it may receive conflicting writes, the solution is for the old leader’s unreplicated writes to be discarded, which may violate durability expectations.
- Discarding writes is dangerous if other storage systems get involved, this leads to inconsistency.
- Split brain. If two nodes both believe they are the leader and accept writes. No process for resolving conflicts. If so, shut down one node if two leaders are detected.
- Timeout for declaring a dead leader is tricky. If too long, longer time for the system to recover. If too short, unnecessary failover, the system would have problems with high load or network issues.

Implementation of Replication Log

Statement-based replication

Leader logs every write request it executes, and sends statement logs to followers. Followers parse and execute those SQL statements from the log.
This may break down because:

- Statements which call a nondeterministic function(NOW(), RAND(), etc) generate different values on each replica.
- For UPDATE execution, which depends on existing data, must be executed in exactly the same order on each replica. Can be limiting when multiple concurrently executing transactions.

Write-ahead log(WAL) shipping

For a Log-structured storage engine, or B-tree. Both of them are append-only sequences of bytes. Besides writing the log to disk, the leader also sends write logs to followers. When the follower processes this log, it builds a copy of the exact same data structures as found on the leader.
Disadvantage: the log describes the data on a very low level, WAL contains details of which bytes change in which disk blocks. Replication closely coupled to storage engine. If we change storage format, we cannot run different versions of database software on the leader and followers.
Zero-downtime upgrade of the database software by first upgrading the followers and then performing a failover and getting a new leader. Otherwise if the replication protocol doesn't allow version mismatch it would require downtime.

Logical(row-based) log replication

- Use logical log as replication log.
- Decoupled from storage engine internals.
- Easy to be backward compatible.
- Allow leader and follower to run different versions of database software or different storage engines.
- Easier for external applications to parse.

Trigger-based replication

- Replication by application code.
- Advantage is more flexibility(subset of data, or from one DB to another, or need conflict resolution logic)
- Greater overheads, more prone to bugs and limitations.

Problems with Replication Lag

Fully synchronous configuration is unreliable.

Read from asynchronous replicas have outdated information.
Types of guarantees: strong consistency, eventual consistency, and the following 3 guarantees.

Reading Your Own Writes

If a user writes to the leader and reads from a follower, it may see stale data.

- Read-after-write consistency
- Read-your-writes consistency
- Guarantee the user read up-to-date data submitted by themselves.
- No promises for other uses

Techniques to implement read-after-write consistency in leader-based replication system:

- When reading something maybe modified by the user, read from the leader.
  Need to know in advance whether something might have been modified. E.g. user profile information of a social network, read user's own profile on leader and other users' info from followers.
- If most things are editable by the user.
  Track the time of the user's last update, for any time after the last updates, make reads from the leader.
  Monitor the replication lag on followers, prevent queries on any follower that is more than any time behind the leader.
- Clients remember the most recent write timestamp. Can be a logical timestamp or actual system clock. Wait until the follower caught up, or read from other replicas.
- There is additional complexity if replicas are distributed across multiple datacenters, since any request involving the leader must be routed to the datacenter who has the leader.
- For the same user is accessing from multiple devices, cross-device read-after-write consistency can be more tricky.
  It's difficult to remember the user's last update timestamp, so this metadata needs to be centralized.
  No guarantee that connections from different devices will be routed to the same datacenter.

Monotonic Reads

A user first reads from a fresh replica, then from a stale replica. Monotonic reads guarantee this anomaly does not happen.
If a user makes several reads in sequence, time cannot go backward.
Lesser guarantee than strong consistency, stronger guarantee than eventual consistency.
How to achieve: Same user can make reads from the same replica unless the replica fails.

Consistent Prefix Reads

Prevent from violation of causality. This type of guarantee is consistent prefix reads.

If a sequence of writes happen in a certain order, then anyone reading those writes should see them in the same order.
For example, Two people having a conversation and one observer. There are two partitions, each partition has one leader and one asynchronous follower. If some partitions are replicated slower than others, the observer may see the dialog in a weird order.
Solutions: 1. Make those writes that are causally related to each other are written to the same partition. 2. There are algorithms to keep track of causal dependencies.

Solutions for Replication Lag

Redesign the system to provide a stronger guarantee, such as read-after-write.
That's why transactions exist. They are a way for databases to provide stronger guarantees so that application can be simpler.

Multi-Leader Replication

More than one node accepts writes. Each node processes a write must forward that data change to all other nodes.

- Multi-leader configuration
- Master-master replication
- active/active replication

Use Cases for Multi-Leader Replication

Multi-datacenter operation

- Each datacenter has a leader.
- Within each datacenter, regular leader-follower replication.
- Between datacenters, each leader replicates its changes to the leaders in other datacenters.
- **Better Performance** – Every write can be processed in a local datacenter and asynchronously replicated to other data centers.
- **Better tolerance of datacenter outages** – When there is a failover of one leader, each datacenter can operate independently, replication catches up when the failed datacenter comes back online.
- **Better tolerance of network problems** – Traffic between datacenters usually goes over public internet, less reliable than the local network within a datacenter. A temporary network interruption does not prevent writes being processed.
- **Big downside** – Write conflicts/Concurrency issues

Clients with offline operation

- Applications need to continue to work while it is disconnected from the internet. E.g. calendar app, notes app etc.

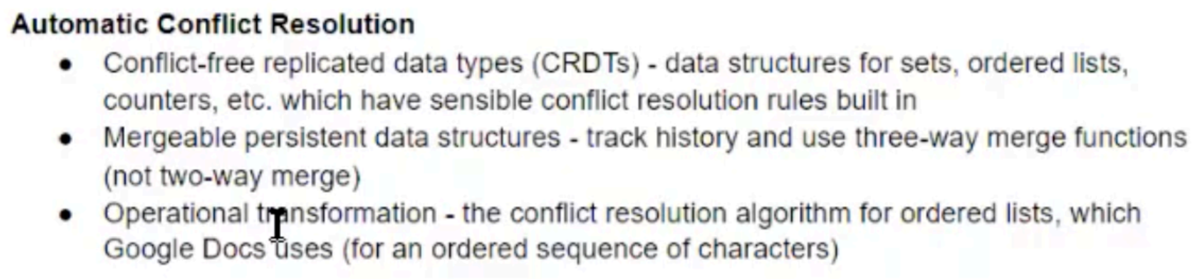
- Every device has a local database that acts as a leader.
- Asynchronous multi-leader replication process(sync) between all devices.
- Replication lag may be hours or days.

Collaborative editing

- Real-time collaborative editing applications. Google doc, Wikipedia, etc.
- Many users edit the same document. The changes are instantly applied to their local replica and asynchronously replicated to the server and other users.
- In order to guarantee no editing conflicts, lock the doc for the first user to commit changes and release the lock, other users have to wait.
- However for faster collaboration, make the unit of change very small and avoid locking, thus this requires multi-leader replication with conflict resolution.

Handling Write Conflicts

1. Synchronous conflict detection make no sense to use multi-leader replication
2. Asynchronous conflict detection makes it too late to ask users to resolve the conflict because they all made successful writes.
3. Conflict avoidance is the simplest strategy.
4. Converging towards a consistent state. Ways to achieve convergent conflict resolution:
    a. Give each write a unique ID(timestamp, UUID, hash of key and value, etc), pick the highest ID as the winner and throw away the others. If you use a timestamp, it's known as last write wins(LWW). This approach implies data loss.
    b. Give each replica a unique ID, let writes from higher-numbered replica wins. This approach implies data loss.
    c. Merge the values together. Order them and then concatenate them.
    d. Record the conflicts in some data structure, and write application code resolves the conflict later.
5. Custom conflict resolution logic - Most appropriate way to resolve a conflict may depend on application:
    a. On write - when detected a conflict, conflict handler run in a background process immediately
    b. On read - when data is read, application may ask user or automatically resolve the conflict and write the result back to the database



Automatic Conflict Resolution

- Conflict-free replicated data types (CRDTs) - data structures for sets, ordered lists, counters, etc. which have sensible conflict resolution rules built in
- Mergeable persistent data structures - track history and use three-way merge functions (not two-way merge)
- Operational transformation - the conflict resolution algorithm for ordered lists, which Google Docs uses (for an ordered sequence of characters)

Multi-Leader Replication Topologies

A replication topology describes the communication paths along which writes are propagated from one node to another. Example topologies:

1. Circular topology
2. Star topology
3. All-to-all topology - most general topology, every leader sends its writes to every other leader.

In 1 and 2, if one node fails, it can interrupt the flow of replication messages between other nodes. The topology could be reconfigured to work around.
In 3, due to network congestion, some network links may be faster than others, therefore some replication messages overtake others. Writes may arrive in the wrong order. Same causality problem as Consistent Prefix Reads. Version vectors can be used.

Leaderless Replication

- Allow any replica to accept writes from clients.
- Leaderless replication models are called Dynamo-style.
- In some implementations, the client directly sends its writes to several replicas, while in others, a coordinator node does it. The coordinator does not enforce write ordering.

Writing to the Database When a Node Is Down

If one of the replicas goes offline and comes up online afterwards, read may be stale data.
Solution: Send read requests to several nodes in parallel, get different response, Version numbers are used to determine which value is newer.

Dynamo-style datastore often use two mechanisms:

Read repair
Client makes read from several nodes and detects stale responses from some replicas, then it writes the newer value back to those replicas.
This approach works well for frequently used values.
Anti-entropy
A background process constantly looks for differences between replicas and copies missing data from one to another.
Unlike replication log in leader-based replication, it does not copy writes in any particular order, and there may be a significant delay.

Quorums for reading and writing

- n: total number of replicas

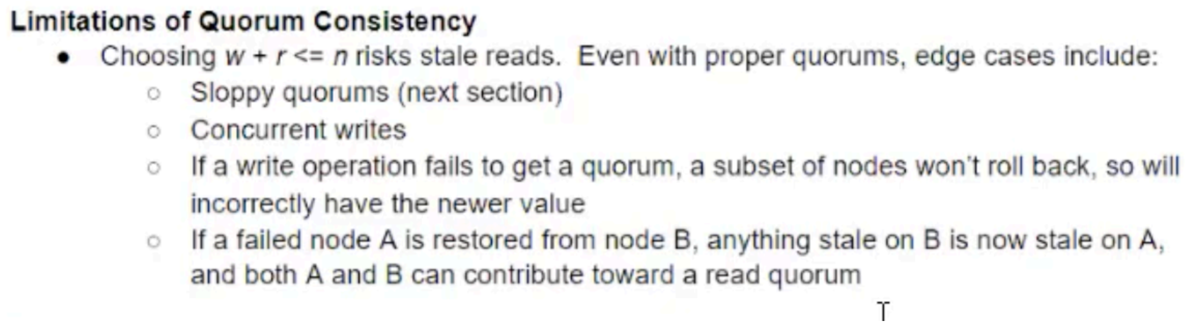
- w: each write must be confirmed by w nodes
- r: each read must query at least r nodes

- As long as w + r > n, we expect to get an up-to-date value when reading, because at least one of the r nodes has up to date data.
- Reads and writes that obey these r and w values are called quorum reads and writes.
- r and w as the minimum number of votes required for the read or write to be valid.
- In Dynamo-style databases, n, w and r are configurable as parameters. A common choice is to make n an odd number, and to set w = r = (n + 1) / 2 and round up.
- If w = n and r = 1, read fast, but one failed node causes writes to fail.

The quorum condition w + r > n, allows the system to tolerate unavailable nodes:
- If w < n, the system tolerates n - w nodes to fail to process writes.
- If r < n, the system tolerates n - r nodes to fail to process reads.

Limitations of Quorum Consistency

Even with w + r > n, there are likely to be edge cases where stale values are returned. E.g., sloppy quorums, concurrency issues, read/write failures.



Limitations of Quorum Consistency

- Choosing *w* + *r* <= *n* risks stale reads. Even with proper quorums, edge cases include:
    - Sloppy quorums (next section)
    - Concurrent writes
    - If a write operation fails to get a quorum, a subset of nodes won't roll back, so will incorrectly have the newer value
    - If a failed node A is restored from node B, anything stale on B is now stale on A, and both A and B can contribute toward a read quorum

Dynamo-style databases are generally optimized for use cases that can tolerate eventual consistency. The parameters w and r allow you to adjust the probability of stale values being read, but it's wise to not take them as absolute guarantees.

From an operational perspective, it's important to monitor staleness. In leader-based replication it's easy to expose metrics for replication lag and feed into the monitoring system. But in systems with leaderless replication, there is no fixed order for writes. Or systems only use read repair and no anti-entropy. There is no limit to how old a value might be if it is infrequently read.

Sloppy Quorums and Hinted Handoff

- Quorums are not as fault-tolerant as they could be.
- In order to continue to accept reads and writes during network interruption in a large cluster (with significantly more than n nodes), read from or write to some available nodes but aren't among the original designated n nodes. This is sloppy quorum.
- In this case, writes and reads still require w and r successful responses, but those may include nodes that are not among the original designated n nodes.
- Hinted handoff: 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.
- Sloppy quorums increase write availability.
- Even when w + r > n, not sure to read the latest value for a key.
- A sloppy quorum is just an assurance of durability, not a quorum in a traditional sense. Sloppy quorums are optional in common Dynamo implementations.
- Leaderless replication is also suitable for multi-datacenter operation, since it is designed to tolerate conflicting concurrent writes, network interruptions, and latency spikes.
    - Cassandra implements multi-datacenter in leaderless model: the number of replicas n includes nodes in all datacenters.
    - Riak implementation is that n describes the number of replicas within one datacenter.

Detecting Concurrent Writes

Concurrent writes in a Dynamo-style datastore have no well-defined ordering.
Conflict resolution

Last write wins

- One approach to achieve eventual convergence is to declare that each replica only stores the most "recent" value and allow "older" values to be overwritten and discarded.
- LWW achieves the goal of eventual convergence, but at the cost of durability. It may even drop non-concurrent writes because of timestamp ordering issues(Chapter 8).
- If losing data is not acceptable, LLW is not a choice for resolving conflicts.
- The only safe way to use a database with LWW is to ensure a key is only written once and then become immutable, to avoid concurrent updates to the same key. For example, in Cassandra, use a UUID as the key, this gives each write operation a unique key.

The "happens-before" relationship and concurrency

Two writes are not concurrent, but one operation is causally dependent on another.
Possible relationships between two operations A and B:

- A and B are independent
- A happened before B - causally dependent
- B happened before A - causally dependent
- A and B are concurrent

Capturing the happens-before relationship

E.g. Two clients add items to the same shopping cart.
The server can determine whether two operations are concurrent by looking at the version numbers - it does not need to interpret the value itself, so the value could be any data structure.
The algorithm works as follows:
The server maintains a version number for each key, increments it everytime the key is written, stores the new version number with the new value.
When client reads a key,

Merging concurrently written values

Version vectors

- Single replica uses a single version number to capture dependencies between operations. When multiple replicas accept writes concurrently, use a version number per replica + per key.
- Each replica increments its own version number when processing a write, and also keeps track of the version numbers from all other replicas.
- This information indicates which values to overwrite and which values to keep as siblings.
- The collection of version numbers from all the replicas is called a version vector.