Replicatio means keeping a copy of the same data on multipl machines that are connected via a network.
Why replicate?
1. reduce latency: keep data geographically close to users
2. increase availability: to allow the system to keep working even if some parts fail
3. increase read throughput: to increase the number of machines that can serve read requests

For now, we assume that the whole db is small enough to fit on a single machine. Later in chapter 6 we relax this assumption and discuss sharding.

The difficulties with replication come with handling changes to (replicated) data.
Three common strategies for replicating changes between nodes
1. single-leader
2. multi-leader
3. leaderless

Leader-based replication

Each node that stores a copy of the database is called a *replica.*
The question that inevitably arise is: how do we ensure that all the data ends up on all the replicas?

Every write to the db needs to be processed by every replica -- else the replicas will no longer contain the same data.
The most common solution to this is *leader-based replication*
- one replica is designated the leader. All write requests from clients go to the leader
- the leader writes new data to its local storage
- when the leader writes new data to local storage ,it sends the data change to all its followers as part of a *replication log* or *change stream*
- each follower takes the log from the leader and updates its local copy of the db accordingly
- when a client wants to *read* from the db, it can query either hte leader or any follower

This is used in postgreSQL, MySQl, MongoDB, Kafka...

Synchronous vs Asycn Replication
- Does the leader wait until some of /all the followers have replicated a new write before reporting success to the client? -- this is called synchronous replication
-- async is when the leader sends the data change log but doesn't wait for a response from followers

- ADvantage of this is that there is guaranteed to be (at least one) replica that is always up to date with the leader -- so if the leader fails, there is at least one up-to-date copy
- Disadvantage is that if the follower does not respond -- because it has crashed, there is a network fault, or any other reason -- the leader must block writes

In practice, if sync replication is used, typically one replica is sycned -- known sometimes as semi-synchronous
Often, leader based replication is completely async

- this means a write is not guaranteed to be durable even if it has been confirmed to the client
- tradeoff is that leader can continue processing writes even if other followers have fallen behind
- a variant called chain replication used in Azure Storage gives some durability guarantees too

Setting up new followers

- typically done by taking a consistent snapshot of the leader's db at some point in time and copying the snapshot to the new follower node
- then the new follower contacts the leader and requests all the data changes since the snapshot
  - this requires that the snapshot is associated with an exact position in the leader's replication log
  - in postgres this is called the log sequence number -- MySQL calls it the binlog coordinates

The practice steps change per db - this is the general idea

Handling Node outages

- how do we keep the system running even when individual nodes go down?

Follower failure: catch up recovyer

- on local disk, each follower keeps a log of data chagnes it receives from the leader - so it can recover from a failure

Leader failure: failover

- first, need to determine when the leader has failed
  - typically done by heartbeats
- then, need to choose a new leader
  - either done through election process -- which brings up a question of how to build consensus
  - or use a previously appointed controller node
  - the best candidate is the replicat that is most up to date with the old leader, to

minimize data loss

- then, reconfigure the system to use the new leader

Failover is fraught with things that can go wrong

- if async replication is used, doesthe new leader have all the writes that the old leader had?
  - and what if the new leader receives conflicting writes?
  - the easiest answer is probably to discard the old leader's writes that have not been synced, but that may violate some durability guarantees
- sometimes two nodes may both believe that they are the leader -- split brain
  - many systems have a mechanism to shut down a leader if 2 are detected
- when should you declare a leader dead?

There are no right answers to any of these quetstions!

How to implement a replication log?

- Statement based replication
  - in the simplest case, the leader logs every write request that it executes and sends that statement log to its followers
  - for a RDB this means every INSERT, UPDATE and DELETE statement is forwarded to the followers; each follower parses and executes the statement as if it were coming from the client
  - this can break down with
    * non deterministic functions - eg RAND()
    * statements that depend on existing data in the replica, since the replica may not be up-to-date with the leader
- WAL shipping
  - leader appends every write to a write ahead log, and also sends it across the network to its followeres
  - the disadvantage is that this method describes the data at a low level ie at the level of the bytes added -- this ties the replication closely to the storage engine, and if the db changes its storage format, it is not possible to run different versions on the leader and the followers
    * makes a zero downtime update impossible
- Logical (row based replication)
  - to address the issues with WAL shipping, use a different log format for replication and for the storage engine
  - the replication log is typically called a *logical log*
  - see book for more details
- Trigger-based replication
  - lets the developer write custom app code to trigger a replication

Replication Lag

If an application reads from an async follower, it may see outdated information if the follower has fallen behind -- this leads to apparent inconsistencies in the database

- running the same query on the leader and a follower can yield different results
- this is temporary -- eventually, followers will catch up -- *eventual consistency*
    - but this is delibeartely vague and there is no limit to how far behind a follower can fall

Reading your own writes

- situation: client writes to leader -- reads from a async follower that has not been updated -- strange inconsistency
- need a guarantee that a client can always read its own writes -- *read-your-writes consistency*
- how to do this?
    - when reading something a client may have modified, always read from leader
        - but to do this, you need to know whether something may have been modified without actualyl querying it
        - sometimes this is easy enough to tell -- e.g. only I modify my own shopping cart
        - but if the user can edit a lot of stuff, this means you will have to read from the leader a lot, negating hte read scaling advantages from replication
        - can use other heuristics
            - e.g. for one minute after update from a user, only read from leader -- assuming that after one minute all the followers will have picked up the chagnes
    - client remembers timestamp of last write -- then only reads from a replica that reflects updates at least until that timestamp
        - this can get complicated if the same user access the db across multiple devices/clients -- all the clients need to know that the last edit timestamp from the user is
- with global datacentres this becomes a bit more complicated -- need to reroute to datacetnre with the leader

Monotonic reads

- situation: user reads from two different replicas -- one has data that the other doesnt
- one way to get around this is to ensure that a user always reads from the same replica -- eg based on the hash of the user's ID

Consistent prefix reads

- situation: if some partitions are replicated slower than others an observer may see an answer before a question/similarly confusing ordering of causal data
- need a guarantee that if a sequence of writes happens in a certain order, then anyone reading those writes sees them in the same order
- this is only an issue with sharded dbs
- one way around this is make sure all causally related writes are written to the same partition

Multi-Leader replication

big drawback of leader-based replication is that if the leader goes down, no writes can be processed

An extension of the leader-based model is to have multipl leaders that accept writes

- these leaders are simultanerously followers to each other

Multi leader setups are complicated and rarely make sense in the context of a single datacentre -- however it may be useful for:
- a multi-data centre operation
  - imagine a database with replicas in several data centres
    - perhaps to tolerate the loss of a data centre
    - perhaps to be closer to users
  - with a normal leader-based replication setup, the leader has to be in *one* of the data centres, and all writes must go through that datacetnre
  - with a multi leader setup, you can have a leader in each datacentre
  - within each data centre, regular leader-based replication is used
  - between datacentres, each dc's leader replicates its changes to the leaders in other datacentres
  - Advantages
    - perceived perofmrance may be better -- since a user's writes only need to go to local datacentre leader
    - tolerance of datacetnre outages
    - tolerance of network problems
      * inter data centre communication happens over the internet, which is liekly less robust than the intra data centre local network
      * so a temporary network interruption does not prevent writes being processed -→ why?
  - Clients with offline operations
    - if you have an appliation that needs to work while it is disconnected from the

    internet
  - e.g. a calendar app
    * the device will have a local database that acts as a leader in that it accepts write requests
    * and there is an asynchronous replicate process between the replicas of your calendar on all your devices
- Collaborative editing
  - real-time collaborative editing, e.g. google docs
  - this has a lot in common with the offline editing use case
  - when one user edits a document, the changes are instantly applied to their local replica -- the state of the document in their web browser or client applicaiton -- and asynchronously replicated to the server and any other users who are editing the same document
  - to guarantee no edit conflicts, the application can obtain a lock on the document before a user can edit it
  - however, for faster collaboration, you want to make the unit of change very small (i.e. single keystroke) and avoid locking
    * this allows multipl users to edit simultaneously, but brings the challenges of multi-leader replication, requiring conflict resolution
    * *Operational transformation* conflict resolution algorithm for concurrent editing of an ordered list of items

Handling Write Conflicts

The biggest challenge with multi-leader replication is that write conflicts can occur -- which means that conflict resolution is required.
User A overwrites text ABC to XYZ in leader A
User B overwrites text ABC to DEF to leader B
each change is applied successfully to the local leader, but when the changes are asynchronously replicated, a conflict is detected.

- this problem would not occur in a single leader database!
- in a single leader db, the second writer will either block and wait for the first write to complete, or abort the second write transaction, forcing hte user to retry the write
- in a multi leader setup, both writes are succesful, and the conflict is only detected asynchronously at some point in time in the future, after the write is recorded
- Question: can you make the conflict detection synchronous? i.e. wait for the write to be replicated to all replicas before telling the user the write was succesful?
  - yes, but you lose the main advantage of the multi leader setup -- allowing each replica to process writes independently
  - might as well just use single-leader replication then....

Conflict avoidance

- the simplest way to deal with conflicts is to avoid them!
- if the application ensures that all writes *for a particular record* go to the same leader, then conflicts cannot occur -- you are in the single leader regime then for the record!

Converging to a consistent state

- sometimes you may want to change the designated leader for a record
  - perhaps the original data centre has failed
  - perhaps user has moved location
- then conflict avoidance breaks down and you need to deal with concurrent writes on different leaders!

Methods

- Give each *write* a unique ID, pick the write with the highest ID as the winner, and throw away the other writes
  - with timestamps, this is last write wins (LWW) -
  - prone to data loss!
- give each *replica* a unique ID, and let writes that originated at a higher-numbered replica take precedence
  - also prone to data loss
- somehow merge values together?
- record the conflict in an explicit data strucuture and write application code that resolves the conflict
  - custom handlers

What is a conflict?

- not always obvious!

Multi Leader Replication Topologies

- all-to-all: every leader sends its writes to every other leader
  - issues if some network links are faster than others -- can have some leaders fall behind
- star: leader forwards to designated root node, which forwards to all others
- circular
  - with star and circular, need to work around single node failures blocking replication across the system

Leaderless Replication/Dynamo-style

- some systems abandon the idea of a leader based system -- they allow any replica to directly accept writes from clients
- used by Amazon's Dynamo, Voldemort, Cassandra

Write and read requests are both sent in parallel to multiple replicas!

- e.g. a client sends a write update to 3 replicas
- 2 ack the write 1 does not -- and we hold that 2/3 acks is a successful write
- now a client wants to read -- it submits a request to all 3 replicas!
- the client receives different responses from different nodes i.e. the up-to-date value from one and stale values from another
- version numbers are used to determine which value is newer
  - Q: how are monotically increasing version numbers coordinated across the replicas?

The replication system needs to ensure that eventually, all writes are on all replicas.

But how does this happen?

1. Read repair
   - when a client sends read requests in parallel, it receives responses from all replicas
   - the client sees that one (or more) replicas have stale version values and writes the latest value to them
2. Anti-entropy process
   - Some datastores have a bacground process that constantly looks for differences in data between replicas and copuies and missing data from one replica to another
   - unlike the replication log in leader-based replication, this process does not copy writes in any particular order, and there may be significant delay before data is copied

Note that without anti-entropy, values that are rarely read may be missing from some replicas and thus have reduced durability

Quorum for reads and writes

- as long as w + r > n, we are guaranteed that a read request will have at least one up to date response
  - w = number of nodes every write must be confirmed by to be successful
  - r = number of nodes the client must query for a read
  - pigeonhole principle...
- there are edge cases however
  - if a write happens concurrently with a read, the write may be reflected on only some of hte replicas. In this case it is undetermined whether the read returns the old or the new value
  - if a write succeeds on some replicas < w but fails on others, it is not rolled back on the replicas where it succeeded -- so even if a write reports as failed, reads may or may not return the value from that write
  - if a node carrying a new value fails, and it gets backed up from a replica carrying an old value, the number of replicas with the new value may fall below w,

breaking quorum

Databases with appropriately configured quorums can tolerate the failure of individual nodes without the need for failover.
they can also tolerate individual nodes going slow because requests dont have to wait for all n nodes to respond -- they can return when w or r nodes have responded.
These characteristics make leaderless databases appealing for use cases with high availability and low latency, and that can tolerate occasional stale reads

Detecting concurrent writes

Dynamo style databases allow several clients to write concurrently to the same key, which means conflicts will occur when strict quorums are used.
- the problem is that events mary arrive in a different order at different nodes due to variable network delays and partial failures

in order to become eventually consistent, the database needs to converge to one value.
But which one?
- Last Write Wins
- "happens-before"
  - Concurrent operations are operations that don't know about each other
    - assuming two operations A and B. Either
      * A happens before B i.e. B relies on A
      * B happens before A i.e. A relies on B
      * A and B are concurrent
    - if there is an ordering to the operations, then we just need to pick the last operation as being the final outcome
    - but if not i.e. if the operations are concurrent, then we need to resolve a conflict somehow
  - so we need to find some way to determine whether there is indeed an ordering to a set of given operations

Single replica versioning algorithm to handle concurreny writes
- The server maintains a version number for every key, increments the version number every time that key is written, and stores the new version number along with the value written
- when a client reads a key, the server returns all values that have not been over-written, as well as the latest version number. A client must read a key before writing
- when a client writes a key
  - it must include the version number from the previous read
  - and it must merge together alll values that it received in the previous read

    - when the server receives a write with a particular version number, it can overwrite all values with that version number or below (since it knows that they have been merged into the new value) but it must keep all values with a higher version number (because those values are concurrent with the incoming write)

But how do we handle this when there are multiple replicas and no leader?

Version vectors

- need a version number *per replica* and per key!
- each replica implements its own version number when processing a write, and also keeps track of the version numbers it has seen from each of the other replicas
- this information indicates which values to overwrite and which values to keep as siblings