5. Replication

- Intro
  - Scaling to Higher Load
    - Shared-Memory Architecture
      - vertical scaling (or scaling up)
      - Problem
        - Twice more powerful machine costs more than twice as much
        - Machine twice the size cannot necessarily handle twice the load
        - Fault tolerance is limited
          - Single geo location
    - Shared-Disk Architecture
      - Used for some data warehousing workloads
      - Problem
        - contention and the overhead of locking
          - limit the scalability
    - Shared-Nothing Architectures
      - horizontal scaling (or scaling out)
  - Replication vs Partitioning
- Why it is good
  - Keep data geo close to users
    - Reduce latency
  - System can work when some parts failed
    - Increase availability
  - Scale number machines that can serve requests
    - Increase read throughput
- If data doesn't change
  - Then very easy
    - Just copy and you are done
- All difficulties lies in handling changes
- Three algos
  1. single-leader
  2. multi-leader
  3. leaderless

1. Single-Leader Replication

- Leaders and Followers
  - replica
    - Each node that stores a copy of the database
  - leader-based replication
    - aka active/passive or master–slave replication
  - leader
    - aka master or primary
  - followers
    - aka read replicas, slaves, secondaries, or hot standbys
  - Leader saves data locally and sends it to Followers
    - replication log or change stream
  - Client
    - Can read from any replica
    - Can write only to Leader
  - This approach is used by many databases, message brokers and even file systems
- Synchronous vs Asynchronous Replication
  - synchronous
    - Leader waits until Follower(s) confirm they received the message
  - asynchronous
    - Leader sends the message but doesn't wait responses from the Follower(s)
  - It is impractical to have many sync Followers
    - semi-synchronous configuration
      - Make just 1 sync Follower
      - And other Followers are async
      - If sync Follower becomes slow then one of async Followers is made sync
  - Often completely async
    - ⚠ i.e. writes are not guaranteed to be durable even if confirmed to the client!
      - Sounds like bad trade-off but it is widely used, especially for geo distributed replicas
    - If leader failed and not recovered then some writes may be lost
    - Leader can continue working even if all it's Followers have fallen behind
  - Research on Replication
    - chain replication
      - Sync replicas in Azure
    - consensus
      - Several nodes agree on value
- Setting Up New Followers
  - Take snapshot from Leader
    - W/o locks
  - Copy snapshot to new Follower and load it
  - Follower connects to Leader and asks for data changes since snapshot
    - Position in Leader's log is named "log sequence number" or "binlog coordinates"
  - When Follower "caught up" it starts processing regular changes from Leader
- Handling Node Outages
  - Follower failure: Catch-up recovery
    - Knows from local log last transaction
    - Request from leader all data changes
    - When catch up continue receiving stream of data changes
  - Leader failure: Failover
    - 👍 Steps
      - 1. Determining that the leader has failed
        - There is no foolproof way to do it
        - Usually timeout is used
      - 2. Choosing a new leader
        - Can be done through election process
        - Or can be appointed by a previously elected controller node
        - Usually best candidate is the one that has the most recent data
      - 3. Reconfiguring the system to use the new leader
        - Clients need to send write requests to new Leader
        - Old Leader should understand that he is not a Leader anymore
    - ⚠ What can go wrong
      - Old leader after repair may have some unsynced writes in case of async replicas. What to do with them?
        - Usually they are discharged
      - Discarging writes can cause problems – if outside system relies on it
        - E.g. auto incrementing counter used as PK
      - In some scenarios two replicas may thing they are Leader
        - "Split brain" situation
        - As a safety catch there can be mechanism to shut down one leader
          - But risk that both Leaders will be shot down
      - What is right timeout before Leader is declared dead?
        - When high load it may reach timeout and things will become even worse
- Implementation of Replication Logs
  - Statement-based replication
    - Each statement is resent to followers
    - But
      - Non-deterministic functions should work differently (now(), rand())
      - Autoinc can work differently
      - Side-effects of triggers / procedures
  - Write-ahead log (WAL) shipping
    - Followers reads log from LST-Trees or WAL used for B-Trees
    - But
      - Need the same format of data for storage - i.e. the same software
  - Logical (row-based) log replication
    - Can have different versions of software
    - Easier to read such log but other apps
  - Trigger-based replication
    - Is more flexible – manually created or with some settings
    - But can have bugs
- Problems with Replication Lag
  - read-scaling architecture
    - Is common for web – lots of reads, few writes
  - Eventual consistency
    - Has replication lag
  - 3 replication lag anomalies
    - Reading Your Own Writes
      - "read-after-write consistency" aka "read-your-writes consistency"
        - Few techniques
          - When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower.
          - for one minute after the last update, make all reads from the leader
            - +monitor the replication lag on followers and prevent queries from slow followers
          - 👍 Writer remembers last timestamp and ensures that replica as at least that timestamp
            - logical timestamp
            - Or actual system clock
              - But them problem with clock sync
          - If multiple datacenter then extra problem – routing requests to the datacenter with leader
      - cross-device read-after-write consistency
        - Store user's last update timestamp across all devices is not easy.
        - May need to route requests for the user from all devices to the same datacenter to read the leader
    - Monotonic Reads
      - Use can see moving backward in time
        - e.g. new comment can appear and then disappear again
      - Monotonic reads
        - Is stronger guarantee than eventual consistency
        - Lesser guaranty than full consistency
      - Techniques
        - Make sure that each user reads data from the same replica
        - ❓ What are other technique?
          - what about also using timestamps from reads?
            - But it can lead to read from leader only?
    - Consistent Prefix Reads
      - If some partitions are replicated slower than others, an observer may see the answer before they see the question.
      - If the database always applies writes in the same order, reads always see a consistent prefix, so this anomaly cannot happen
      - Techniques
        - any writes that are causally related to each other are written to the same partition
  - Solutions for Replication Lag
    - 👍 Transactions are needed to let the DB provides stronger guarantees to the apps can be simpler
      - But they are too expensive for partitioned and distributed databases

2. Multi-Leader Replication

- also known as master–master or active/active replication
- 👍 Use Cases for Multi-Leader Replication
  - Multi-datacenter operation
    - One leader per datacenter
    - It is outdated approach because of problems
      - Autoincrement
      - Triggers
      - Integrity constraints
  - Clients with offline operation
    - Local database on local device
    - CouchDB
  - Collaborative editing
    - Lock on write. Lock small keystrokes
- Handling Write Conflicts
  - Synchronous versus asynchronous conflict detection
    - Can ask user to wait until sync is done but this will lose benefits of multi-leaders
  - Conflict avoidance
    - All writes for particular records should be done via the same leader
    - But when one datacenter fails and you need to reroute you still should solve conflicts
  - Converging toward a consistent state
    - Last write wins (LWW)
      - Can cause data loss!
    - Higher numbered replica wins
      - Can cause data loss!
    - Merge values somehow
    - Record all conflicts
      - Maybe resolve them later by user
  - Custom conflict resolution logic
    - On write
      - Show user conflict on write and ask to solve it
    - On read
      - Show user conflict on read and ask to solve it
  - Automatic Conflict Resolution
    - 👍 Conflict-free replicated datatypes (CRDTs)
      - Should read about it!
    - Mergeable persistent data structures
      - Like GIT
    - 👍 Operational transformation
      - Should read about it!
  - What is a conflict?
- Multi-Leader Replication Topologies
  - 👍 3 approaches
    - Circular
    - Star
    - All-to-all
  - ⚠ Problems
    - Cirtular and Star
      - Need to prevent infinite looping of the messages
      - If node fails it can block replication
    - All-to-all
      - Some replication directions can be faster than others
      - (a): Insert. (b): select + update. (c): Can receive update then insert
      - Solutions
        - version vectors

3. Leaderless Replication

- Aka "Dynamo-style"
  - Client or coordinator sends writes to multiple nodes
- Writing to the Database When a Node is Down
  - Read repair and anti-entropy
    - Read repair
      - Client reads and repairs data
    - Anti-entropy process
      - Process that reads and repairs
      - Important to ensure durability of rarely reading data
  - Quorums for reading and writing
    - w + r > n
- Limitations of Quorum Consistency
  - There are cases when in w+r>n you can see stale records - when nodes w & r has small overlapping
  - ⚠ 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.
  - ⚠ Monitoring staleness
    - It is important metric for such type of databases!
- Sloppy Quorums and Hinted Handoff
  - Multi-datacenter operation
  - ❓ Q: what is non-sloppy quorum? How we read from where we write?
- Detecting Concurrent Writes
  - Last write wins (discarding concurrent writes)
    - If overwriting writes is acceptable - e.g. caching
  - The "happens-before" relationship and concurrency
    - A before B
    - B before A
    - A and B are concurrent
      - Doesn't mean that in the exact same time!
  - Capturing the happens-before relationship
  - Merging concurrently written values
    - Developer should merge siblings (concurrent values)
      - The same as conflict resolution
      - Not just remove removed item but mark it with tombstone
  - Version vectors
    - version number per replica as well as per key
  - Version vectors and vector clocks
- ❓ Q: Is leaderless slower? When leaderless is good?