TDT4225 Assignment 4

Simen Refsland

November 1, 2023

Task 1 Kleppmann Chap 5

a)

Multi-leader replication may be useful when you have replicas in different datacenters. This is because for a single-leader architecture, writes must go over the internet to the datacenter with the leader, which can increase latency. For multi-leader replication, writes can happen in the local datacenter and be replicated asynchronously to the other ones. This architecture is also resilient towards failures of whole datacenters, as other datacenters can continue operating independently of the failed ones.

Another use-case is if there are clients who need to be able to make writes while not connected to the Internet. In this case, a local database on the client's device needs to act as a leader, and synchronize with the server when the client is again connected to the Internet. Collaborative editing is also quite similar in this regard, as the local changes you write in the document must be synchronized and reconciled with the other people's contributions.

The biggest downside to multi-leader replication is the conflict that arises when the same data is concurrently modified, leading to the question how to reconcile the differences. Problems such as integrity constraints and auto-incrementing keys may lead to undesired and dangerous behavior, which makes single-leader architectures desirable if there aren't any compelling reasons not to use them, such as the ones listed above.

b)

One problem with SQL statements are that they may be non-deterministic, such as NOW() or RAND(), which may not correctly reproduce the original data. Another problem is with autoincremented columns, or conditional updates, where they must be executed in the exact same order on the replica as the original to ensure that they are the same. Likewise, statements with side-effects may also be non-deterministic. It is possible to deal with this by for example returning the exact value for non-deterministic functions, but due to the number of edge cases, it is less painful to use log shipping.

Task 2 Kleppmann Chap 6

a)

Kleppmann doesn't (as far as I can see) say what strategy for rebalancing is best, but using hash mod N is not encouraged because it moves a lot of data around uncessarily when the number of nodes are increased. Besides this, two main strategies are outlined: fixed number of partitions, where the number of partitions are much greater than the number of nodes and usually not changed after setting up the database, and dynamic partitioning, where the number of partitions grows with the size of the database, splitting partitions when they grow over a certain size.

Using a fixed number of partitions would not be ideal if key range partitioning is used, because the data could potentially be all in one partition if the boundaries are not correctly set. The main caveat for fixed number of partitions is that the size of partitions need to be "just right" to perform well. Too large, and rebalancing and recovery becomes expensive, too small leads to bigger overhead. Dynamic partitioning may lead to poor utilization of other nodes if the dataset is small, and it may be beneficial to do preemptive pre-splitting prior to creation.

b)

Local indexing involves scattering/gathering for reads, meaning several partitions are involved, but only requires updating a single partition on write. Global indexing has the opposite properties, a read can be served from a single partition, but writing requires several updates to different partitions. I assume that ratio of reads/writes are the most important factor to consider when deciding which to use.

Task 3 Kleppmann Chap 7

a)

Read-committed: transaction 1 acquires read lock of object A, reads the value of A (old value), releases it, transaction 2 acquires write lock for object A for rest of transaction, does the change, transaction 2 acquires the lock of object B for rest of transaction, does the change, transaction 1 can't read the value of B yet, can't therefore commit the transaction, transaction 2 commits, transaction 1 acquires read lock of B, reads value of B (new value), releases lock, and commits. Conclusion: transaction 1 reads an inconsistent state of the db. It should be noted that most implementations do not require read locks, but remembers the old and new value so that no dirty reads are performed.

Snapshot isolation: transaction 1 reads the value of object A without acquiring locks, transaction 2 acquires write lock and creates a new version of A, transaction 2 acquires write lock of B and creates a new version of B, transaction 1 reads the version committed that existed before transaction 1 started, transaction 1 commits, transaction 2 commits. Here transaction 1 reads a consistent version of the database, where both the value from A and B are the old ones.

b)

Transaction 1 first obtains a lock in shared mode when reading object A, transaction 2 tries to write to object A, can't because an exclusive lock can't be acquired before transaction 1 commits, transaction 2 acquires an exclusive lock on object B and writes to it, transaction 1 cannot read object B before transaction 2 is committed. This is a deadlock, since transaction 1 is waiting for the exclusive lock on object B to be released, and transaction 2 waits for the shared lock on object A to be released. This should be detected by the system, and aborts one of the transactions. If transaction 1 is aborted, the shared lock on object A is released, and transaction 2 writes to A. If transaction 2 is aborted, transaction 1 acquires a shared lock on object B and reads the previous value. I would assume transaction 1 is the reasonable one to abort in this case.

Task 4 Kleppmann Chap 8

a)

The request may have been lost, the request may be queued and not sent yet, the remote node may be down, the request may have been received but the response was lost etc.

b)

Client B's write (on the same data) may be causally later than client A's, but the timestamp may say that client B was earlier, which would mean that client A's write would win in a last write wins (LWW)

approach. The example in the book shows that for a multi-leader replication setup, client A sets x=1, which client B increments, so that x=2, but client B has an earlier timestamp than client A, even though client B modifies the value set by A. This leads to the write x=1 winning, and client B's operation being lost.

Task 5 Kleppmann Chap 9

a)

Linearizability is a consistency model that imposes a special ordering constraint (total order) which makes the system behave as if there was only a single copy of the data and every operation is atomic, making it possible to distinguish which operation happened before the other. So while linearizability provides this specific ordering, consensus is what makes it possible to agree on this order.

b)

Many distributed data systems are not only usable, but also desirable to be not linearizable. Distributed data systems being linearizable comes with increased latency and reduced availability. There exists other consistency models, such as eventual consistency, which states that after some time without new writes, the replicas converge to the same value, or casual consistency. The different domains will have different requirements in regard to these guarantees, for example financial systems versus social media sites.

Task 6 Coulouris Chap 14

a)

We can not deduce that $e \rightarrow f$, because Lamport clocks only estimate *potential* causality. In the example in the book, e has a lower count than b, even though they are happening concurrently. However, if we use vector clocks, then we can deduce whether $e \rightarrow f$ or not because by keeping track of the knowledge that one process has of other processes, concurrent events will not have counts that are strictly less or greater than the other. If now $V(e)_i \leq V(f)_i \wedge V(e)_i \neq V(f)_i \quad \forall i$, then we can conclude event e happened before f.

b)

P1:
b: (4, 0, 0)
n: (5, 4, 0)
P2
k: (4, 2, 0)
m: (4, 3, 0)
u: (4, 4, 0)
P3
c: (4, 3, 2)

c)

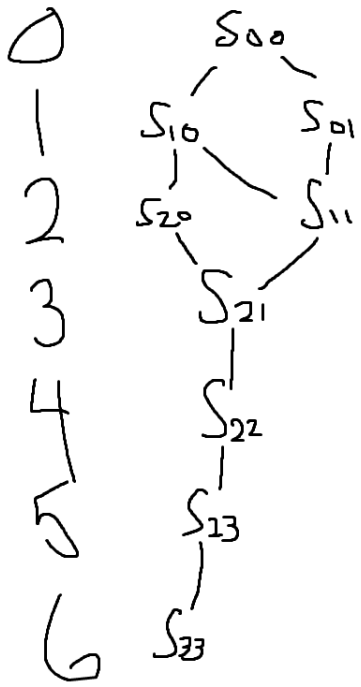

Figure 1: Alternative consistent runs for the events

Task 7 RAFT

When a new leader is elected, its log entries are considered to be the truth. The election of a leader ensures that this leader has the up-to-date committed log. Log entries are only committed if the majority of followers acknowledge that this log entry is received and at least one of the followers entries are from the leaders term (when doing LogReplication RPCs). This ensures that any new leader will have these committed entries as well (as the followers will deem this server to be the most complete). The leader deals with inconsistencies (comparing term numbers for the given index) among followers by deleting extraneous entries and inserting missing entries by backtracking to earlier log entries, where any subsequent entries will be removed, thus ensuring that the logs will be equal to the leaders. If any old leader reconnects, the RPCs from this server will be rejected, as the term number for this old leader is stale.

Task 8 MySQL RAFT

The main takeaways from introducing RAFT into MySQL was higher reliability, greatly reduced failover time (2 seconds vs 30-40 seconds) and operational simplicity with comparable write performance to semisync replication protocol.