Exercise 4

When should you use multi-leader replication, and why should you use it in these cases? When is leader-based replication better to use

- Multi-leader approach doesn't make sense within a single datacenter, so let us assume we have several datacenters, each with its own leader handling the writes coming into its datacenter. The leaders then communicate over the internet to replicate. This increases performance, as the latency could be significantly reduced to the geographical distance between the client and datacenters. It would also tolerate the failure of an entire datacenter, be it from network problems or power outages.
  Another use case for multi-leader replication is for offline work, and then sync up when back online. Calendar for instance, syncing it up to the computer when the phone has internet again. Here each device acts as its own leader updating itself when online. Google docs, aka. simultaneous collaborative editing, is also possible with multi-leader approach. It allows these applications (users) to share their local replica with other users, and through some conflict handling, allowing them to write in the same document simultaneously.

- Leader-based approach is better when we just have one datacenter. The reason is that we do not have to handle write conflicts between the different leaders. All of the followers will be updated as per usual by the one leader. If a leader fails, it could be a single point of failure however, but normally this is handled through what is called a failover, where a follower is assigned as the new leader.

Why should you use log shipping as a replication means instead of replicating the SQL statements?

- One should use log shipping as a replication means instead of replicating the SQL statements because of all the side effects it could lead to. An example of this is the use of nondeterministic functions like RAND() and NOW(), which most likely would return different results for each replica. Other statements with side effects, for instance user-defined functions, could also result in different side effects if not absolutely deterministic. Autoincrementing columns and statement depending on existing data in

the databases could also be problematic. The need to be executed in exactly the same order on each replica to avoid different values. This is limiting when there are multiple concurrently executing transactions.

One could work around these problems by replacing any nondeterministic function calls with a fixed value in the replication log. Because of all the edge cases, other replication methods are preferred though.

What is the best way of supporting re-partitioning? And why is this the best way, according to Kleppmann?

- The best way of supporting re-partitioning is either to use a fixed, high number of partitions or use dynamic partitioning. The fixed number of partitions works by assigning each node multiple partitions. With the introduction of a new node, it steals some from the other nodes (equally). However, the fixed number of partitions is difficult to get just right. If the partitions are too large, rebalancing and recovery will become expensive. If it is too small, there would be too much overhead.
- Dynamic rebalancing works with a new partition being introduced when a partition exceeds some set size. The data is then split between this new partition and the old one. This way, the number of partitions adapt to the size of the data, which scales well with little overhead. The problem is in the start however, as all partition will be handled by a single node until it splits. *Pre-splitting* prevents this however, where an initial set of partitions is created for an empty database.

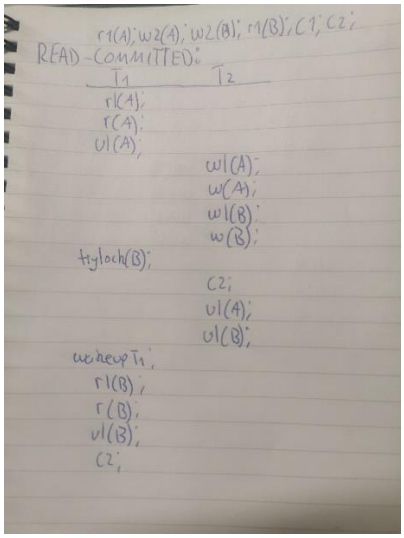
Explain when you should use local indexing and when you should use global indexing.

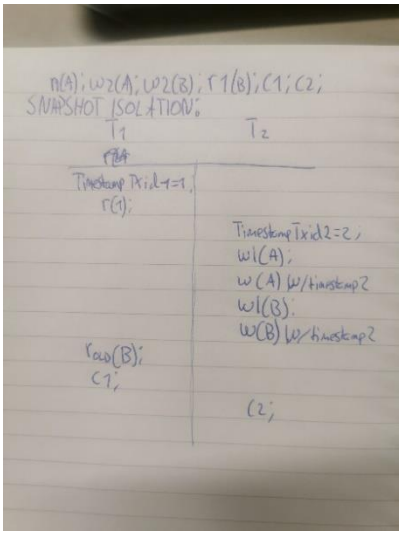
- Local indexing is used when expecting a lot of writes. It writes more efficiently because one could just write to the document using the partition that contains the document. Reads on the other hand comes from querying all partitions and combining the results, which is expensive.

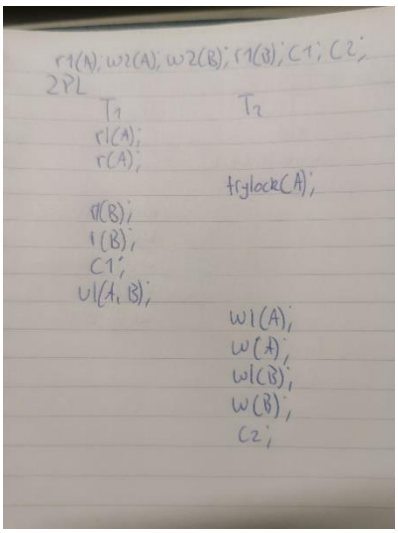
  Global indexing has the opposite effect because the indexes themselves are partitioned. Reads are very efficient as we just query the partition containing the term we want. Writing could affect multiple partitions and would in those cases make it more expensive. It is therefore better to use global indexing when expecting a lot of reads.

Show how the following schedule is executed using *read committed* and *snapshot isolation.*

r1(A); w2(A); w2(B); r1(B);c1;c2







If you send a message in a network and you do not get a reply, what could have happened? List some alternatives

- Different things could have happened. The request got lost, the request is queued and will be processed later, the remote node may have failed, the remote node is temporary down and will respond later, the request is processed but the reply is lost, or the request is processed but the reply is delayed.

Explain why and how using clocks for *last write wins* could be dangerous.

- Using clocks for *last write wins* could be dangerous because physical clocks are rarely completely synchronized between nodes. If a node then makes a write at 20.003, and another one does a write at 20.005 in that nodes time, but its own clock says 20.002, then the second node which actually was last, will lose its write. This leads to the problem that other writes which are the basis for other writes ("happened before"), may have a newer timestamp than the one they rely on.

Explain the connection between ordering, linearizability and consensus

- The goal of linearizability is to make a system appear as if there is only one copy of the data. When two users request some data, they should receive the same values at any point in time. Ordering helps preserve *causality*. This means that we can be certain requests dependent on each other happens in the order we want it to. Concurrent

requests can be ordered in any way we want. A row must for instance be created before it gets deleted. It is a linearizable system, we have a total order of operations. This means that we can pick any two elements and compare them (who came first). Ordering on *causality* will be a partial ordering, as we can compare some elements, but not all. One type of ordering used is *timestamp ordering*, where nodes keep track of requests. Consensus means getting nodes to agree on something. This can be agreeing on who the leader is, what the outcome of a transaction will be, or something else. Consensus involves an asynchronous system of processes, which might be unreliable. Total order broadcast is a protocol for exchanging messages between nodes that must ensure reliability and that the ordering properties are always satisfied. A node cannot change the order of the messages if earlier messages have already been delivered – making total order broadcast stronger than timestamp ordering.

- Ordering means the order of operations, and ordering helps preserve *causality*. This way we know that x happened before y. For instance, a row must have been created before it gets deleted. One type of ordering is *timestamp ordering*. The goal of linearizability is to make it appear as there is only one "copy" of the data. When two users request some data, they should receive the same values. Consensus means getting the different nodes to agree on something, like who is the leader or what the outcome of the transition is. Linearizable systems make an ordering between all operations.

Given events e and f. Logical (Lamport) clock values L leads to L(e) < L(f). Can e have "happened before" f? Why? What happens if one uses vector clocks instead? Explain.

- Even though event two at P2 has a higher Lampart timestamp than event one at P1, we still cannot say that one "happened before" the other, as there are two local events at different processes who doesn't influence each other. The same way, we can't say that L(e) < L(f), because of the same reasoning, they could be concurrent events where f actually happened before e, but it made sense to order them this way to preserve total order. With Vector clocks on the other hand, this is now possible, because we count how many events at other processes that a given event a process can have been influenced by.

RAFT has a concept where the log is replicated to all participants. How does RAFT ensure that the log is equal on all nodes in case of a crash and a new leader?

- The leader replicates the log by sending AppendEntriesRPCs. This also functions as heartbeats. Thus, the leader will send AppendEntriesRPC also in case of no new log. If the leader crashes, a new leader will be eleceted by nodes transomforming from follower to candidate. By randomizing the timeouts, one new leader will win before the others even have started to send out RequestVoteRPCs.The new leader holds the truth and will use its log. Other nodes (followers) may get their log truncated or extended depending on the situation.