1. (a) One should use multi-leader replication in cases where
   i. we have multiple datacenters
   ii. we have an application that needs to continue working even in cases where its connection to the internet has dropped.
   iii. we have a system in which multiple users need to be able to collaborate and edit in real time.

   This type of replication allows for better performance, since writes can be processed locally, and then replicated asynchronously to other datacenters. Additionally, this makes for a higher tolerance of outages since each datacenter can continue operating normally even when one datacenter has failed. A replication like this will also make it so that the system tolerates network problems better, since a temporary network outage would not prevent writes from being processed.
   One could use leader-based replication in single datacenters or in general in cases where for instance it is not a problem for the system that all writes must go through this one leader, and that the system won't often experience network interruption [Kle17, pp. 168-170].

   (b) One should use log shipping as a replication means instead of just replicating the SQL statements because sending SQL statements can be unpredictable in terms of non-deterministic functions, where functions calling for a time now or a randomly generated number will be different on the different replicas. This method can also cause unpredictable scenarios in terms of auto-incrementing IDs, in addition to side effects possibly having different results on different replicas [Kle17, p. 159].

2. (a) The best way of supporting re-partitioning is by using a hash function to decide the partition for a given key. This way, skewed data is uniformly distributed [Kle17, pp. 201-205].

   (b) Global indexes make reads more efficient than when using a local index, so if the system is read-heavy, global indexes would be better, rather than if the system is write-heavy, in which case local indexes would be better [Kle17, p. 208].

3. (a) The given scheduel will be executed the following way using read commmitted:
   i. r1(A): Transaction 1 reads A. No lock conflict.
   ii. w2(A): Transaction 2 wants to write to A, but must wait because A was read by Transaction 1 and the write operation cannot proceed until after Transaction 1 commits.

iii. w2(B): This operation must also wait because Transaction 2 is waiting to write to A.
iv. r1(B): Transaction 1 reads B. No lock conflict.
v. c1: Transaction 1 commits and the lock on A is released.
vi. w2(A), w2(B): Now that Transaction 1 has committed, Transaction 2 can write to A and B.
vii. c2: Transaction 2 commits.

In snapshot isolation, the given schedule will be executed the following way:

i. r1(A): Transaction 1 reads A and gets a snapshot of A at the time the transaction began.
ii. w2(A): Transaction 2 writes to A, which does not affect Transaction 1's snapshot.
iii. w2(B): Transaction 2 writes to B, which also does not affect Transaction 1's snapshot.
iv. r1(B): Transaction 1 reads B and sees the snapshot from when the transaction began, not the updated value from Transaction 2.
v. c1: Transaction 1 commits.
vi. c2: Transaction 2 commits.

(b) Using serializable with 2PL, the given shceule will be executed the following way:

i. r1(A): Transaction 1 reads and gets a lock on A.
ii. w2(A): Transaction 2 wants to write to A, but must wait because A is locked by Transaction 1.
iii. w2(B): Transaction 2 wants to write to B, and since there is no lock on B, this operation can proceed. Transaction 2 now holds a lock on B.
iv. r1(B): Transaction 1 wants to read B, but must wait because B is locked by Transaction 2.
v. c1: Transaction 1 commits. The locks on A and B are released.
vi. w2(A): Now that Transaction 1 has committed, Transaction 2 can write to A.
vii. c2: Transaction 2 commits.

4. (a) There can be numerous reasons as to why a message in a network won't get a reply once its been sent. Some of them are:

i. The packet may be lost.
ii. It may be delayed due to it waiting in queue.

iii. The remote node may have failed, either from crashing or potentially being powered down. It may also have stopped working altogether.

iv. The response to the message could have been lost or delayed.

[Kle17, p. 278]

(b) Using clocks for last write wins could be dangerous due to the fact that database writes could disappear due to nodes with lagging clocks being unable to overwrite any value already written by a node with a faster clock, until the skew has been resolved. Last write wins can also not differentiate between writes that happened sequentially in quick succession. It is also possible for two nodes to generate writes with the exact same timestamp, which needs another tiebraker value to solve the conflict, though this can lead to violations of causality [Kle17, pp. 292-293].

5. (a) The connection between ordering, linearizability, and consensus is that they all deal with the issue of maintaining consistency in distributed systems. Linearizability is a strong form of consistency that makes it easier to understand and predict how concurrent operations will behave. Consensus algorithms can be used to decide on a global order of operations, which is useful for implementing linearizable systems [Kle17, pp. 373-375].

(b) There are numerous distributed systems that are usable even when they are not linearizable, usually due to the fact that they provide other forms of consistency guarantees. Examples of these are leaderless and multi-leader replication systems [Kle17, p. 375].

6. (a) Using logical clocks, one cannot necessarily deduce that e "happened before" f. This is because logical clocks only provides a partial ordering of events: e and f could still be concurrent events even though L(e) ¡ L(f) [Cou+11, p. 607].
If one uses vector clocks instead, one can deduce that e "happened before" f when V(e) ¡ V(f), since an array of logical clocks are maintaned and updated based on specific rules for each process in the system [Cou+11, p. 609].

(b) i. b: (4, 0, 0)

ii. c: (0, 3, 2)

iii. k: (4, 2, 0)

iv. m: (0, 3, 0)

v. n: (5, 4, 0)

vi. u: (0, 4, 0)

[Cou+11, p. 610]

(c) A series of possible consistent states the system can have had would be:

i. S00: Initial state where both P1 and P2 are at (0,0).
ii. S10: P1 has processed its first event, moving to (1,0).
iii. S11: P2 receives the message from P1 and moves to (1,1).
iv. S20: P1 processes another local event, moving to (2,1).
v. S21: P2 processes a local event, moving to (2,2).
vi. S31: P1 sends a message to P2, moving to (3,2).
vii. S32: P2 receives the message from P1 and moves to (3,3).

[Cou+11, p. 622]

7. RAFT ensures that the log is equal on all nodes in case of a crash and a new leader by electing a new leader when an existing one fails, and accepting and replicating logs across the cluster by enforcing other logs to agree with the leader's log [OO14, p. 307].

8. The main advantages from introducing RAFT into MySQL was making it into a truly distributed system by moving the source of truth of membership and leadership inside the server, since operations like promotions of new leaders and membership changes were the cause of most of the issues they were facing. This enabled provable correctness of promotions of new leaders and any membership changes within the server [Rah+23].

References

[Cou+11] George Coulouris et al. *Distributed Systems: Concepts and Design*. 2011.

[OO14] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of USENIX ATC'14* (2014).

[Kle17] Martin Kleppmann. *Designing Data-Intensive Applications*. 2017.

[Rah+23] Anirban Rahut et al. "Building and deploying MySQL Raft at Meta". In: *Engineering at Meta* (2023). DOI: https://engineering.fb.com/2023/05/16/data-infrastructure/mysql-raft-meta/.