Assignment 4

Task 1 - Kleppmann Chap 5

a) 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 replication is where clients send each write to one of several leader nodes, any of which can accept writes. The leaders send streams of data change events to each other and to any follower nodes. This is preferable in some use-cases, such as multi-datacenter operation, offline operations and collaborative editing. When having multiple datacenters, it is better for the performance when every write can go to a leader within the same datacenter instead of over the internet to another. In addition, multi-leader replication in multi-datacenter operations means better tolerance of datacenter outages and network problems as each datacenter can operate independently and catch up later. This is also preferable with offline operation, where each device has its own leader, which catches up with the rest of leaders when connected to the internet again, and collaborative editing (similar to offline editing) where each local device has its own replica (leader) and asynchronously replicated to the server and any other users in the same document.

The disadvantage of multi-leader replication is the complexity, which is the advantage of single leader replication. It is easier to know the sequence with one leader, there is a natural ordering as it can apply a sequence number per transaction. This makes it much easier to know what happened first if a conflict/problem occurs. It is easy to understand, and there is no conflict resolution to worry about.

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

Log shipping, both WAL and logical, have the advantage of being the recipe of the leaders current state. One may think initially that the same is for statement-based replication, but there is some instances that might cause the followers to have some slight differences in their versions in comparison to the leader. For example, any calls to a nondeterministic function, such as rand() or now() is likely to generate different values. It is possible to work around those issues by for example replacing any nondeterministic

function calls with a fixed return value when logged so that the followers all get the same value. Other factors such as the use of *auto increment* and *side effects* may give you differences if not run in the exact same order as the leader, yet the order is possible to insure if ran sequential. However, because of the amount of edge cases, it is generally preferred to use other replication methods.

c) How could a database management system support *read your writes consistency* when there are multiple replicas present of data?

The book mentions a few of the various techniques to implementing read-after-write consistency. Firstly, you have the way of always reading something that may have been modified, i.e. your own social media profile, from the leader, else one of the followers. This will work as you dont't have the option of editing all other social media profiles, and they can be read from a follower. If most things in the application are edible, this won't be effective, as you have to read from the leader most of the time (negating the benefit of read scaling). In this case, other criteria could be used to decide whether to read from the leader, e.g. track time and after a certain amount of time after update always read from the leader, else from a follower. The use of timestamps is also possible, then the system can verify that the follower it reads from have updates until at least that timestamp, or use a logical timestamp such as the log sequence number.

Task 2 - Kleppmann Chap 6

a) Why should we support sharding / partitioning?

Sharding/partitioning has multiple possible advantages; such as improved scalability, performance, security and availability, in addition to operational flexibility and matching of data store to pattern of use. If you divide data across multiple partitions, each hosted on separate servers, you can scale out the system almost indefinitely. Data access on each partition take place over a smaller volume of data, and if done correctly, the system will be more efficient. Separation of sensitive and nonsensitive data into different partitions and apply different security controls brings improved security. Lastly, partitioning allows each partition to be deployed on a different type of data store, based on cost and the built-in features that data store offers. The reasons are many!

b) What is the best way of supporting re-partitioning? And why is this the best way? (According to Kleppmann).

Kleppmann describes a fairly simple solution: create many more partitions than there are nodes, and assign several partitions to each node. For example, a database running on a cluster of 10 nodes may be split into 1000 partitions, assigning 100 partitions to each node. Now, if a node is added to a cluster, the new node can steal a few partitions from every existing node until partitions are fairly distributed once again. If a node is removed from the cluster, the same happens in reverse. This method is called fixed number of partitions.

c) Explain when you should use local indexing, and when you should use global indexing?

For answering this question, we have to take a look at what advantage we get with each of them. The advantage of a global (term-partitioned) index over a document-partitioned index is that it can make reads more efficient: rather than doing scatter/gather over all partitions, a client only needs to make a request to the partition containing the term that it wants. However, the downside of a global index is that writes are slower and more complicated, because a write to a single document may now affect multiple partitions of the index (every term in the document might be on a different partition, on a different node). This means that an application where reading speeds are more important, and writing to the database rarely happens, global indexing is preferable. If on the other hand writing happens more often than reading, one may prefer local indexing.

Task 3 - Kleppmann Chap 7

a) Read committed vs snapshot isolation. We want to compare *read committed* with *snapshot isolation*. We assume the traditional way of implementing read committed, where write locks are held to the end of the transaction, while read locks are set and released when doing the read itself. Show how the following schedule is executed using these two approaches:

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

trans 1 read A — trans 2 write A — trans 2 write B — trans 1 read B — trans 1 commit — trans 2 commit

All use exclusive write locks, which are released when transaction ends.

Read committed:

1. Transaction 1: Acquire lock on A
2. Transaction 1: Read A
3. Transaction 1: Unlock A
4. Transaction 2: Acquire lock on A
5. Transaction 2: Write A
6. Transaction 2: Acquire lock on B
7. Transaction 2: Write B
8. Transaction 2: Commit transaction and unlock A and B
9. Transaction 1: Acquire lock on B
10. Transaction 1: Read B
11. Transaction 1: Commit and unlock B

This method guarantees for no dirty reads and no dirty writes, but transactions have to wait for each other to finish. Dirty read is that a process can read an uncommitted value from a resource, and dirty write is that a process can overwrite an uncommitted value.

Snapshot isolation

1. Transaction 1: Acquire shared lock on A
2. Transaction 1: Read A
3. Transaction 2: Acquire write lock on A
4. Transaction 2: Write A
5. Transaction 2: Acquire write lock on B
6. Transaction 2: Write B

7. Transaction 2: Commit transaction and unlock A and B
8. Transaction 1: Acquire shared lock on B
9. Transaction 1: Read B
10. Transaction 1: Commit transaction and unlock A and B

With snapshot isolation, we still guarantee for no dirty write/read, but we allow transactions to read and write on the same resources. How we prevent dirty writes/reads is that the database has a snapshot, you may think about it as a image of a state of the database, where when you are trying to read an uncommitted value, you read a copy of the old committed value which is now soon to be changed if committed by the the process changing it. I am not sure how it prevents dirty write though, I think I have to read up on this chapter again for gaining a better understanding. All these answers I got may have holes too, but I try to understand how the different methods uses locks and if they prevent other processes to read/write or not.

b) Also show how this is executed using serializable with 2PL (two-phase locking).

2PL:

1. Transaction 1: Acquire lock on A
2. Transaction 1: Read A
3. Transaction 1: Commit and unlock A
4. Transaction 2: Acquire lock on A
5. Transaction 2: Write A
6. Transaction 2: Acquire lock on B
7. Transaction 2: Write B
8. Transaction 2: Commit transaction and unlock A and B
9. Transaction 1: Acquire lock on A
10. Transaction 1: Read A
11. Transaction 1: Commit and unlock A

In this method, a process have to commit before another process can acquire the resource.

c) Explain by an example write skew, and show how SSI (serializable snapshot isolation) may solve this problem.

Kleppmann described a good example on this in his book: Imagine that Alice and Bob are the two on-call doctors for a particular shift. Both are feeling unwell, so they both decide to request leave. Unfortunately, they happen to click the button to go off call at approximately the same time. In the application the scenario looks like this: In each transaction, your application first checks that two or more doctors are currently on call; if yes, it assumes it's safe for one doctor to go off call. Since the database is using snapshot isolation, both checks return 2, so both transactions proceed to the next stage. Alice updates her own record to take herself off call, and Bob updates his own record likewise. Both transactions commit, and now no doctor is on call. Your requirement of having at least one doctor on call has been violated; we have a write skew.

This may be solved by implementing serializable snapshot isolation (SSI). For providing serializable isolation, the database have to detect situations in which a transaction may have acted on an outdated premise and abort the transaction in that case. There are two cases which it has to detect: reads of a stale MVCC object version (uncommitted write occurred before the read) and writes that affect prior reads (the write occurs after the read). By detecting these and handling the premises correctly, we get rid of write skew and fix the problem described above. No lack of doctors on shift :-)

Task 4 - Kleppmann Chap 8

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

When you send a packet over the network, it may be lost or arbitrarily delayed. Likewise, the reply may be lost or delayed, so if you don't get a reply, you have no idea whether the message got through or not.

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

A node's clock may be significantly out of sync with other nodes (despite the best of efforts to set up NTP), it may suddenly jump forward or back in time. As Kleppmann said himself in the book, relying on the node's clock is dangerous because you most likely don’t have a good measure of your clock’s error interval. During a conflict in a system which takes into use *last write wins,* the system may end up picking the value from a node which is very out of sync in favor of the better/more correct value.

c) Given the example from the text book on “process pauses”, what is the problem with this solution to obtaining lock leases?

```
while (true) {
  request = getIncomingRequest();

  //Ensure that the lease allways has at least 10 seconds remaining
  if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
    lease = lease.renew();
  }

  if (lease.isValid()) {
    process(request);
  }
}
```

First of all, as stated in the book, it is relying on synchronized clocks. This means that the expiry time on the lock lease is set by a different machine, and it’s being compared to the local system clock. The code depends on the clocks not being out of sync, which is a naive assumption. Further on, even if we change the protocol to only use the local monotonic clock, there is another problem here. The code assumes that very little time passes between the point that it checks the time `System.currentTimeMillis()` and the time when the request is processed `process(request)`. Normally this code runs very quickly, so the buffer of 10 seconds (10000ms) is more than enough to ensure that the lease doesn’t expire in the middle of processing a request. However, what if there is an unexpected pause in the execution of the program? For example, imagine the thread stops for 20 seconds around the line `lease.isValid()` *b*efore finally continuing. In that case, it’s likely that the lease will have expired by the time the request is processed, and another node has already taken over as leader. But, there is nothing to tell this thread that it was paused for so long, so this code won’t notice that the lease has expired until

the next iteration of the loop—by which time it may have already done something unsafe by processing the request.

Is it crazy to assume that a thread might be paused for so long? Unfortunately not. And that is some of the reasons why this solution to obtaining lock leases is problematic.

Task 5 - Kleppmann Chap 9

a) Explain the connection between ordering, linearizability and consensus.

Linearizability is a popular consistency model: its goal is to make replicated data appear as though there were only a single copy, and to make all operations act on it atomically. Although linearizability is appealing because it is easy to understand—it makes a database behave like a variable in a single-threaded program; it has the downside of being slow, especially in environments with large network delays. Then you also have causality, which imposes an ordering on events in a system (what happened before what, based on cause and effect). Unlike linearizability, which puts all operations in a single, totally ordered timeline, causality provides us with a weaker consistency model: some things can be concurrent, so the version history is like a timeline with branching and merging. Further on in the book, Kleppmann talks about how achieving consensus means deciding something in such a way that all nodes agree on what was decided, and such that the decision is irrevocable. All these three is desireable to have in a system, and appear together when talking about how we make a system seem like one even though it is partitioned and replicated on multiple nodes in a cluster. We want to have one system to talk to, with the right order of events as it happens, with all the nodes agreeing on this.

b) Are there any distributed data systems which are usable even if they are not linearizable? Explain your answer.

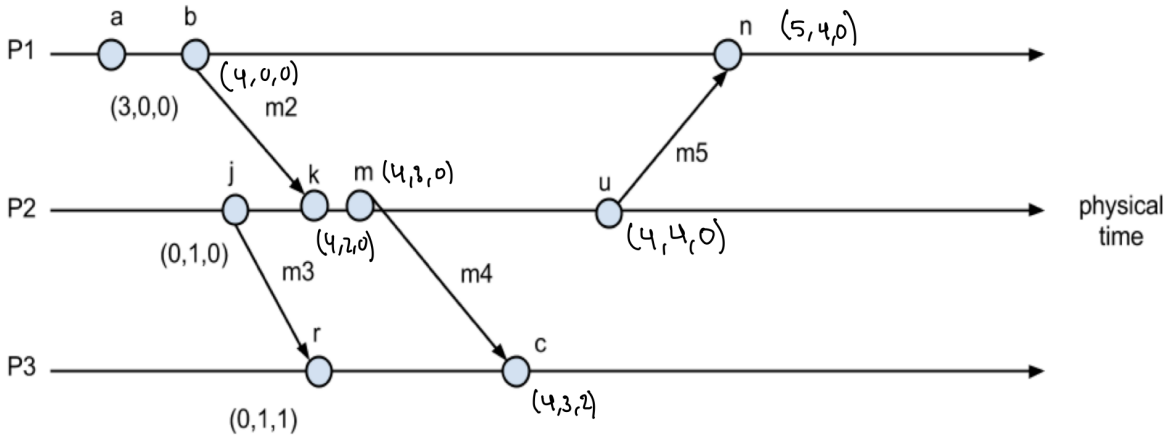
Leaderless and multi-leader replication systems usually do not require global consensus, and are therefore not necessarily linearizable. In these versions we rather have to cope without linearizability, and learn to work better with data that has branching and merging version histories. They are still perfectly usable, they just have to work without it.

Task 6 - Coulouris Chap 14

a) Given two events e and f. Assume that the logical (Lamport) clock values L are such that $L(e) < L(f)$. Can we then deduce that e "happened before" f? Why? What happens if one uses vector clocks instead? Explain.

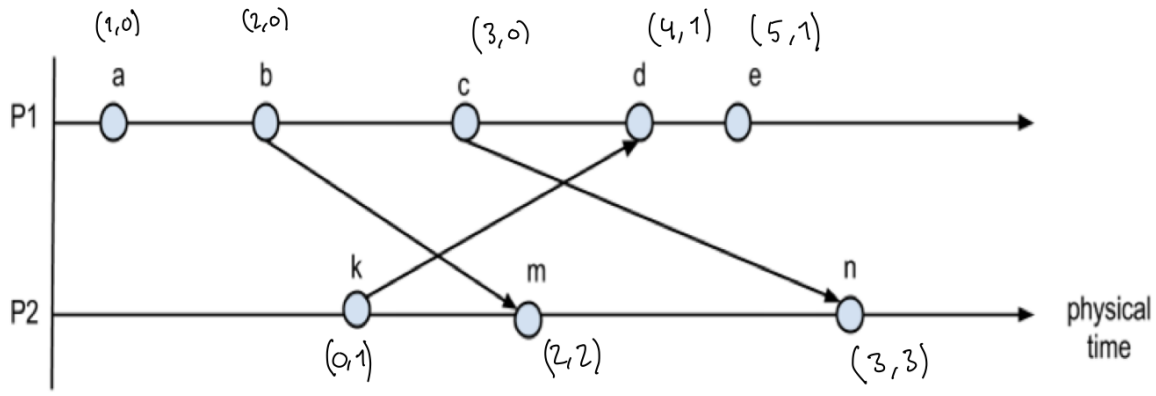
Logical clocks (Lamport) values only shows whether or not an event happened before or after another on a single timeline/process. When it comes to whether or not we can deduce that e happened before f, logical clocks only get you so far. The problem is that if the events are happening on different processes, they may be concurrent. It is possible if on the same process or we have a message in between, but if they are concurrent we simply cannot obtain this information. This is where one of them using a vector clock could come in handy. If you have access on one of the events vector clock, you may deduce if it happened before based on the value of that process against the lamport value. It is safe to say that for figuring out more about event order, we need to store/transfer more info than in logical clocks. This is why we should use vector clocks.

b) The figure below shows three processes and several events. Vector clock values are given for some of the events. Give the vector clock values for the remaining events.

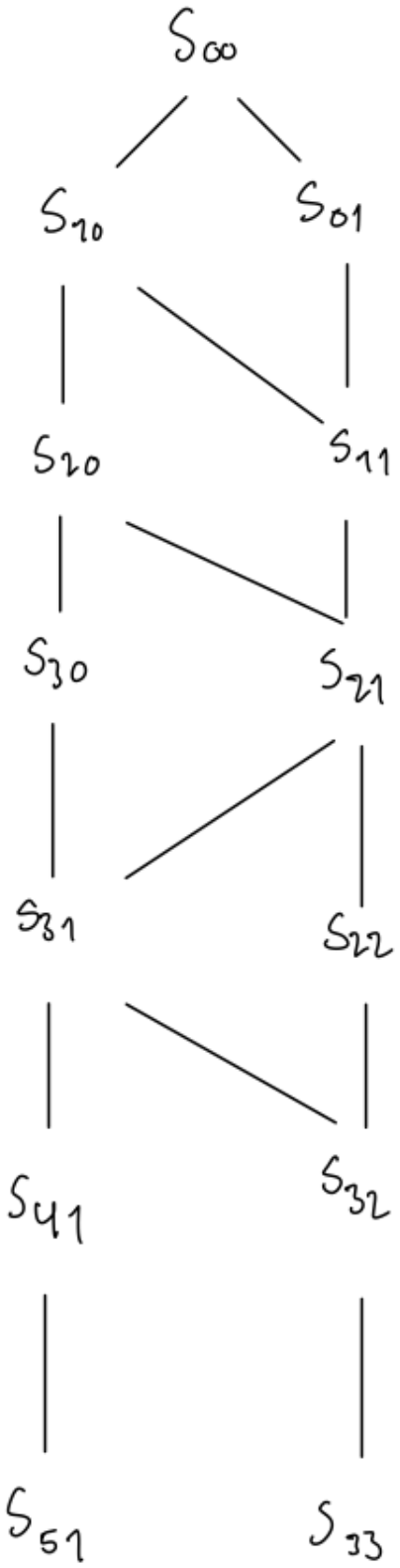


c) The figure below shows the events that occur at two processes $P1$ and $P2$. The arrows mean sending of messages. Show the alternative consistent states

the system can have had. Start from the state S_{00}. (S_{xy} where x is in $P1$'s state and y is in $P2$'s state)



As long as it has no effect without cause in a cut, it is considered a consistent run


S_{00}
S_{10}
S_{01}
S_{20}
S_{11}
S_{30}
S_{21}
S_{31}
S_{22}
S_{41}
S_{32}
S_{51}
S_{33}

Task 7 - RAFT

a) 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?

When a Leader is elected, it is responsible to update the Followers' log to the Leader's one. When updating a Follower, it finds the first matched `<entry, term>` backwards, and update the Follower with the following logs. RAFT ensures that the logs before the matched `<entry, term>` are the same using the property of the RAFT algorithm called *Log Matching*. When sending an AppendEntries RPC, the leader includes the index and term of the entry in its log that immediately precedes the new entries. If the follower does not find an entry in its log with the same index and term, then it refuses the new entries. The consistency check acts as an induction step: the initial empty state of the logs satisfies the *Log Matching Property*, and the consistency check preserves the *Log Matching Property* whenever logs are extended. As a result, whenever AppendEntries returns successfully, the leader knows that the follower’s log is identical to its own log up through the new entries.

Therefore, one can say that if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

Task 8 - Dynamo

a) Explain the following concepts/techniques used in Dynamo:

- **consistent hashing**

  Consistent hashing tries to solve a partitioning problem by assigning nodes a random responsible area of the hash ring, ending up with departure or arrival of a node only affecting its immediate neighbors. This causes incremental scalability.

- **vector clocks**

  One problem which is addressed in Dynamo is high availability for writes. This is done by implementing vector clocks with reconciliation during reads. The advantage of this is that version size is decoupled from update rates.

- **sloppy quorum and hinted handoff**

Sloppy quorum and hinted handoff are mechanisms for handling temporary failures - which provides high availability and durability guarantee when some of the replicas are not available.

- **merkle trees**

  Merkle trees are in Dynamo used in anti-entropy for recovering from permanent failures. The advantage is that the system synchronizes divergent replicas in the background, which is made possible because of the use of merkle trees.

- **gossip-based membership protocol**

  The last problem Dynamo addresses according to the lecture, is membership and failure detection. This is solved by the technique of gossip-based membership protocol and failure detection, which preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.