Assignment 4

Task 1.a

When we have multiple datacenters locations with replicas to be closer to the user, then the normal leader-follower is not as efficient. There will be a lot of performance issues if everyone is writing to the same leader, however with multi-leader this delay is hidden from the user. However the big downside with multi-leader is that data can easily have conflict and create issues with auto generated ids etc.

Task 1.b

The reason we should use log shipping instead of SQL statement replication is because SQL statements are non-deterministic, meaning that the same statement may give different values. An example of this is the command NOW(). Further autoincrementing is dependent on already existing data and is difficult to keep consistent.

Task 1.c

When talking about read-your-writes we mainly want to focus on data the user has updated. Say I have changed my facebook page, and if I reload these changes should be visable immediately, but i dont need to see other people changes in realtime. This means that the user should write to the leader which updates all the followers, howeven when the user is trying to access data about itself get this data from the leader, the rest is gotten from the followers. There are ofcoure optimizations here where the user can keep track of last update with a timestamp and if a replica is not up to date with this timestamp, read from leader.

Task 2.a

We mainly want to support partitioning because of scalability, since the partitions can be stored across many disks which means multiple processes for the query load.

Task 2.b

According to Kleppmann the best way of re-partitioning is by making the fixed number of partitions. It is a fairly simple solution and if done correctly it can be highly effectiv in most circumstances. Since everything is fixed there is quite few moving parts that makes managing this system quite simple.

Task 2.c

When the partitions are split such that you dont need to gather data from multiple partitions with the same query we can use local indexing since there wont be an index collision. However if there is a need to f.ex. get all items from multiple partitions and these needs to be merged we need to have global indexing to distinguish each item.

Task 3.a

Read committed:

Transaction 1: Reads initial value from DB A

Transaction 2: Writes new value to DB A

Transaction 2: Writes new value to DB B

Transaction 1: Reads initial value from DB B (transaction 2 has write lock and this is not committed yet)

Transaction 1: Commits

Transaction 2: Commits

Snapshot isolation:

These transactions may happend paralell since Snapshot creates copies of the original.

Transaction 1: Reads initial value from DB A

Transaction 2: Writes new value to DB A

Transaction 2: Writes new value to DB B

Transaction 1: Reads initial value from DB B (there are no locks since the transaction is done on a copy)

Transaction 1: Commits

Transaction 2: Commits

Task 3.b

Transaction 1: Reads initial value from DB A

Transaction 1: Reads initial value from DB B

Transaction 1: Commits

Transaction 2: Writes new value to DB A

Transaction 2: Writes new value to DB B

Transaction 2: Commits

Task 3.c

Write skew happens when two processes query the same object concurrenty and then procceeds to do updates based on this. An example is the two doctor problem where two doctors look at who is on call. Since there is only need for one doctor and both doctors sees that there are two doctors on call right now they take them selfs of. Now instead of both doctors being on call there are none and we have a problem.

Task 4.a

There are a number of possible things that could have happend. A node might have beed cut off from the rest of the network, the fault detection does not detect that there is a problem in the network. If the system is using timeout detection then the system wont know that the node is dead and the message is lost. There might be queues in the system and the request is taking longer then normal.

Task 4.b

The reason its dangerous to use clocks for last write wins is becase when speaking about networks things dont happend instantaneously. There is a travel time for the message to go through the system and when talking about multiple systems communicating this timedelay may be different from machine to machine. The local times on each machine may also not be perfectly in sync or may be at a faster/slower rate. Because of this small timevariation is can cause problems when multiple machines are trying to update the same object using last using time. This is because if machine A is writing to object at 00:02, and 5ms later machine B writes to the same object but its internal clock is 00:01 the new update wont go through because it happend "before"

Task 4.c

The problem with this method is that its using multiple system clocks to compare time and this can a problem if the clocks are out of sync by a couple of second. Secondly there is the problem with code running perfectly every time. Since there are few garanties that a thread never pauses the time from a lock creation to the process running may exceed the designated timewindow and can cause problems as well

Task 5.a

A connection might be the leader elections process where we need linearizability because we only want one leader. Therefor there is only one variable and of course time importance of order where process A might effect process B therefor process A needs to be orderes before process B. Since this is all linear the order is even more important. Lasty we need consensus to make sure that the leader is has the propper permissions given by the nodes and that previous leaders dont try to write.

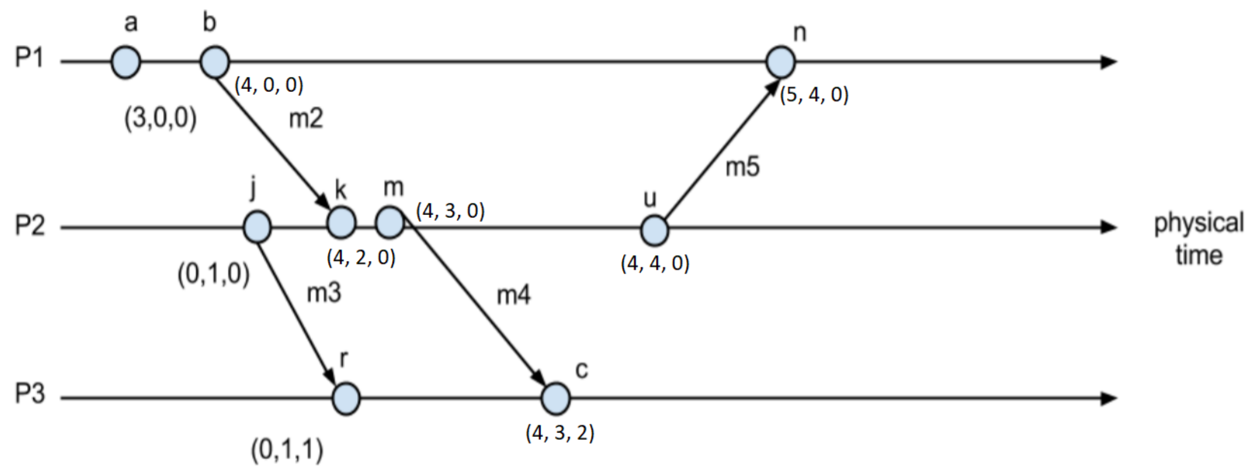
Task 5.b

There may be data systemt that are usable even tho they are not linearizable. I would belive that any system that is rapidly changing ex. stock prices is non linearizable since the data may not be accurate all the time, however with a simple refetch of the data this issue can be fixed if needed. Therefor this system is still useable but may not be precise.

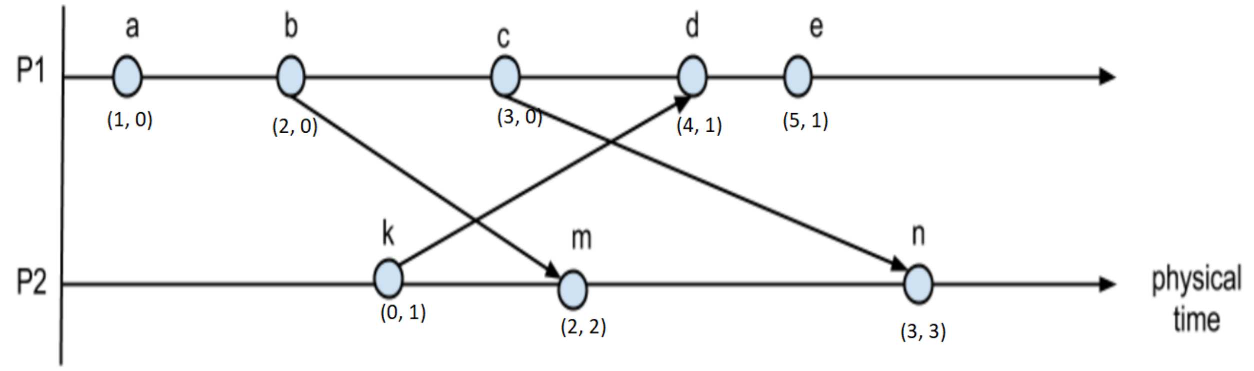
Task 6.a

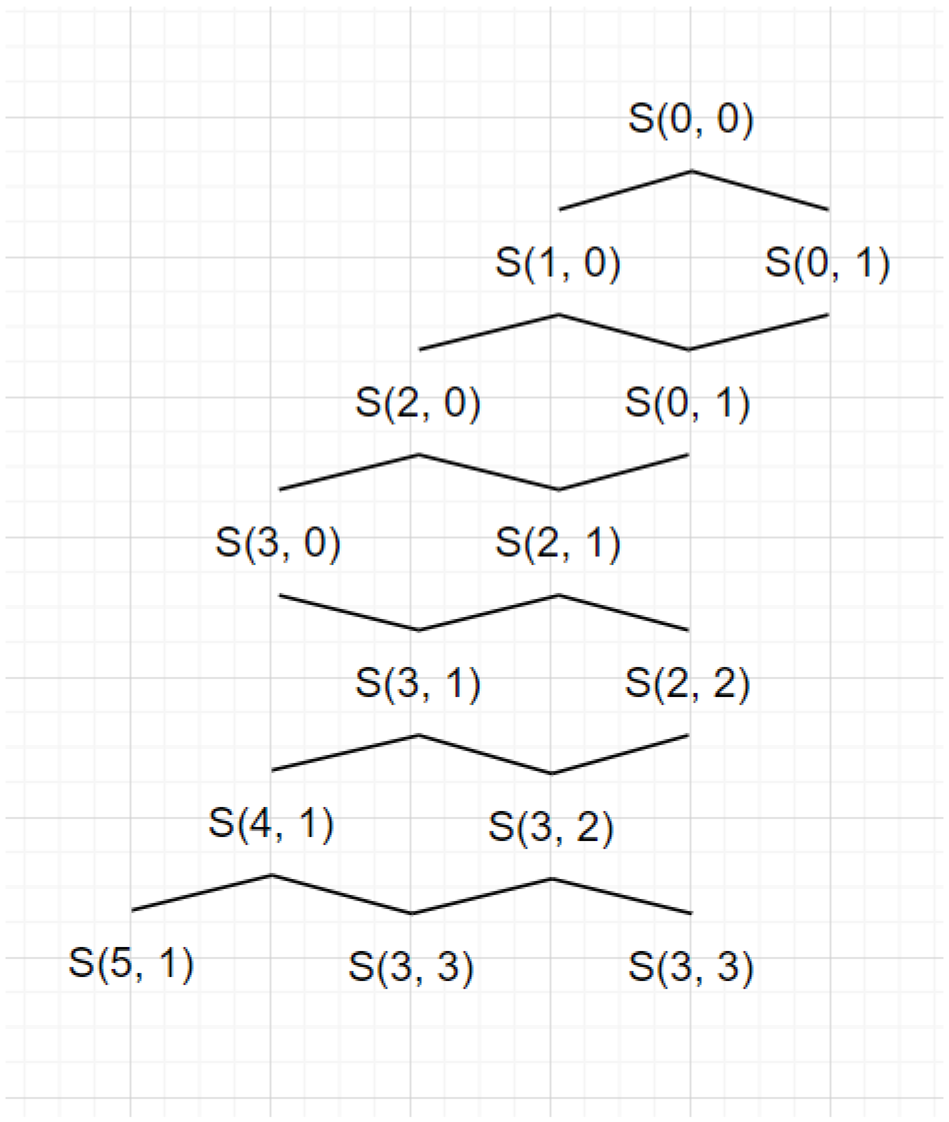
We can not be sure that e happend before f is $L(e) < L(f)$ because the logical clock is a local timer and if e and f are run on seperate processess e may have been run after f but is sooner in a different sequence. On the other hand using vector time we can conclude thst $V(e) < V(f)$ means that e is before f by inducing the length of a sequence of events.

Task 6.b


P1
P2
P3
a
b
n
(3,0,0)
(4, 0, 0)
m2
(5, 4, 0)
m5
j
k
m
(4, 3, 0)
u
physical time
(4, 2, 0)
(0,1,0)
m3
m4
(4, 4, 0)
r
c
(0,1,1)
(4, 3, 2)

Task 6.c


P1
P2
a
b
c
d
e
(1, 0)
(2, 0)
(3, 0)
(4, 1)
(5, 1)
k
m
n
(0, 1)
(2, 2)
(3, 3)
physical time



Task 7

When a new leader is elected, it starts getting client requests that need to be added to a log. This log contains a state machine command as well as a term number for when the entry was recieved by the leader. This number is used to detect if there is inconsistencies inbetween the logs. Finally all the logs have an index attached to identify its posisiton in the log.

To ensure that the logs are coherent there is implemented a Log Matching Property that follows the following rules:

1. If two entries in different logs have the same index and term, then they store the same command

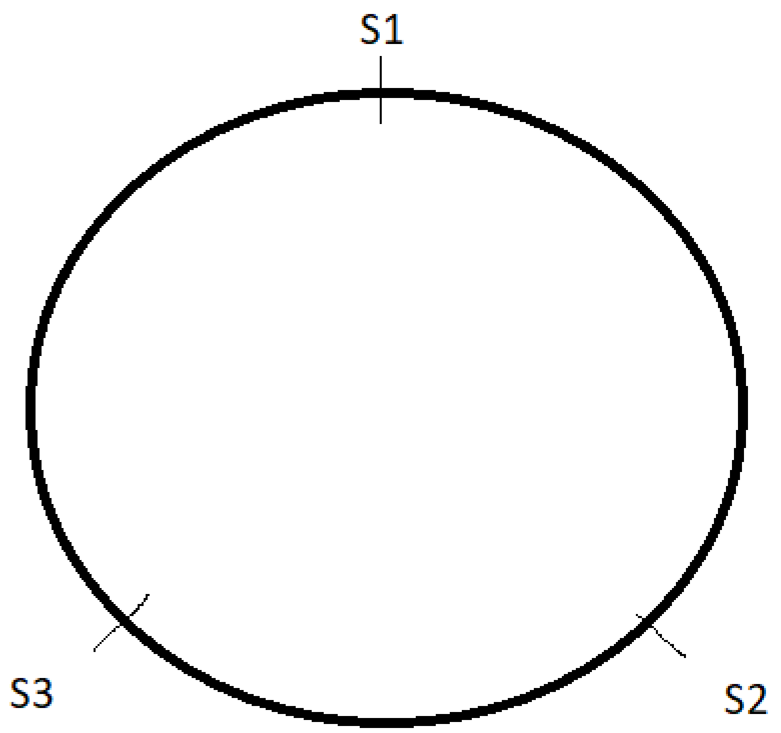
2. If two entries in different logs have the same index and term, then the logs are identical in all preceding entries

When the leader sends an AppendEntries RPC, the leader includes the index and term of the entry just before the new one. If the follower does not have this entry it can refuse the new entry. This check acts as an induction step and helps the leader ensure that the followers logs are identical to its own log.

There are rearly any conflicts here under normal circumstances. But in the event of a leader crash there has to be a new leader that is elected. And this leaders logs may not be up to date. This problem can have a domino effect where multiple leaders crash and the final new leader is so far behind on the logs that its basically useless. But with some elegant RAFT magic the new leader forces all the followers to duplicate its own logs.

Task 8.a

consistent hashing: Solves the problem of rehashing by providing a distribution scheme without having to worry about scaling. This is done by having a hash ring which is an abstract circle with data connected to a server. This ring can be scaled as much as pleased without effecting the rest of the system.



Now in consistent hashing when a server is removed or added then the keys from this server is moved to the next server. Say we remove S3, then all the keys are moved to S1, but the keys in S1 and S2 are untouched. This gives incremental scalability

vector clocks: This is a data structure used to capture causality between different versions of the same object. The clock is a list of versions containing a node, counter pair.

sloppy quorum and hinted handoff: mechanisms for handing temporary failures, and provides high availability and durability guarantee when some of the replicas are not available.

merkle trees: Solves the problem of having to use the entire dataset all the time. Since the tree is a hash tree where the leaves are hashes of the values of individual keys. The parent of these nodes are hashes of there children. By doing this we can easiely check data independently. This is mainly used for anti-entropy and synchronizes divergent replicas in the background.

gossip-based membership protocol: Is used to propagate membership changes and maintain an eventually consistent view of memberships. Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.