TDT4225 Exercise 4

Hauk Aleksander Olaussen

November 1, 2021

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?**

As Kleppmann says: "It rarely makes sense to use multi-leader replication within a single datacenter".

Usecases for this approach then makes us turn our head toward operations distributed over multiple datacenters. This will make each datacenter have its own leader, which will be the one to recieve the writes, but it will share and recieve data from other leaders in other datacenters. This will improve performance as a client may connect to any of the leaders in any datacenter (probably the one closest geographically), and not just one designated leader in one datacenter somewhere in the world. It will also tolerate a whole datacenter failing due to some fault - be it network problems or outages. The database will still be operating while the faulty datacenter is catching up to the rest.

Still, multiple datacenters is not the only usecase for multi-leader replication. They are appropriate to use when needing an application to work without internet connection, and then "sync" up when reconnected, making each device act as a datacenter/leader updating itself when it gets data from another device. Kleppmann uses the calendar on our phones and other devices as an example for this. It may also be used in applications where collaborative editing is a thing. Multi-leader replication will allow these applications (users) to share their local replica with other users, and with some conflict handling, allowing them to write in the same document simultaneously.

When not having the operation distributed over multiple datacenters, using leader-based replication with just a single leader will propably be a better approach. One of the reasons for this is that we do not need to worry about conflicts when it comes to writing. All the followers will update themselves based on the leader, and nothing else - meaning that the leader has absolute control. This means, however, that should the leader fail, a user can not write to the database. This is dealt with by assigning one of the followers as the new leader, a process called *failover*. For smaller scale applcations, the single leader approach will probably be a better choice.

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

We would not want to just replicate the SQL statements given to the leader if it includes non-deterministic functions, as this would mean that we write data to the follower not neccessarily equal to the leaders. Here, log shipping would be a better approach. Another case is if the statements depend on data already existing in the

database. This would mean that the SQL statements must be executed in the same order as they did on the leader to ensure that the data is the same.

As we can see, just replication the SQL statements on the follower will not neccessarily give us the result we want, and we must then use another approach - namely different kinds of log shipping.

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

Read-after-write consistency (read-your-writes) essentially means that a user should always see the data they themselves have submitted. It ensures this by always reading data that the user has modified from the leader, which always has the most recent data. If most of the things in the application are able to be edited by the user, this approach will not do, as most of the data will have to be read from the leader. Another way of ensuring this is to save time of the last update, and after a set time (Kleppmann uses 1 minute in his example), all the reads will be made from the leader. One can also save the timestamp and make sure that the read is handled by a replica sufficiently up do date, or wait until the current one is.

There are other related problems to this - namely *monotonic reads* and *consistent prefix reads*, which considers the user always seing data it has already seen, and always viewing the data that makes causal sense, respectively.

2. Kleppmann Chap 6

a) **Why should we support sharding / partitioning?**

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

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