NTNU

Kunnskap for en bedre verden

TDT4225 - Very Large, Distributed Data Volumes

Exercise 4

Author:
Martin Skatvedt

Date 09.11.2023

1. Kleppmann Chap 5

a

The main reason to use multi-leader replication is if there are multiple datacenters with replicas. This would reduce latency, because each datacenter could have its own leader, which a user could read and write from its local datacenter. In addition in a leader-based system, if the datacenter fails, the leader would have to promote a follower in a another datacenter. However when using multi-leader replication, each datacenter can work independently of the others. Lastly communication between datacenters usually works on the public internet, which makes is less reliable, than local internet. Single-leader replication is sensitive to such problems, because writes are made synchronously. Multi-leader replication (with asynchronous replication) on the other hand can usually tolerate this much better.

Multi-leader replication is more complex than normal leader-based replication, and comes with problems such as conflict resolution. Leader-based replication is often better to use if the data is not distributed over a large geographical area, and you want a less complex system.

b

Replication using SQL statements or statement-based replication, every INSERT, UPDATE or DELETE statement is forwarded to the followers, adn then executed there. This however has some big problems. Non-deterministic statements such as *NOW()* or *RADN()* will result in different values on the leader, and on the follower. Another problem is that if statesments use an auto-incrementing column, each statement has to be executed in exactly the same sequence as the leader. Statements may also have side effects, such as user-defined functions. Since statement-based replication has many problems, we instead use write-ahead log shipping. This log is an append-only log, consisting of bytes conatining all writes, commits and aborts from the leader. When a follower uses this log, it will build an exact copy of its leaders data-structure.

2. Kleppmann Chap 6

a

According to Kleppman Dynamic Partitioning is the best way to support re-partitioning. With dynamic re-partitioning, each partition is assigned to one node, and each node can handle many partitions. Each partition will then grow until a maximum, such as 10GB in Hbase, and then split into two partitions, where each partition has approximately half the data. Likewise is a partition shrinks below a threshold, it can combine merge with another partition. When a node has had many partition splits, it can transfer partitions over to another node, to balance the load. An advantage of this, is that is has a low overhead, since the number of partitions grows with the data. We also don't move data more than necessary.

b

Local indexing or document-based indexing, makes each partition separate from the other partitions. It only holds the secondary indexes for its own data and doesn't care about the other partitions. A downside of this approach is that all items with the same secondary index, isn't always in the same partition, which means you have to query all the partitions to get all the results. Which makes queries very expensive. Global indexing on the other hand keeps a global index which is then partitioned. This makes reads much more efficient, since we only have to query the partitions with the item we are looking for. However this makes writes much slower, since each write may effect multiple partitions. In many cases, updating a global index is asynchronous. You

should use global indexing when you have more than one secondary index, and you want to reduce read speed. Use local indexing when you want to reduce write overhead.

3. Kleppmann Chap 7

a

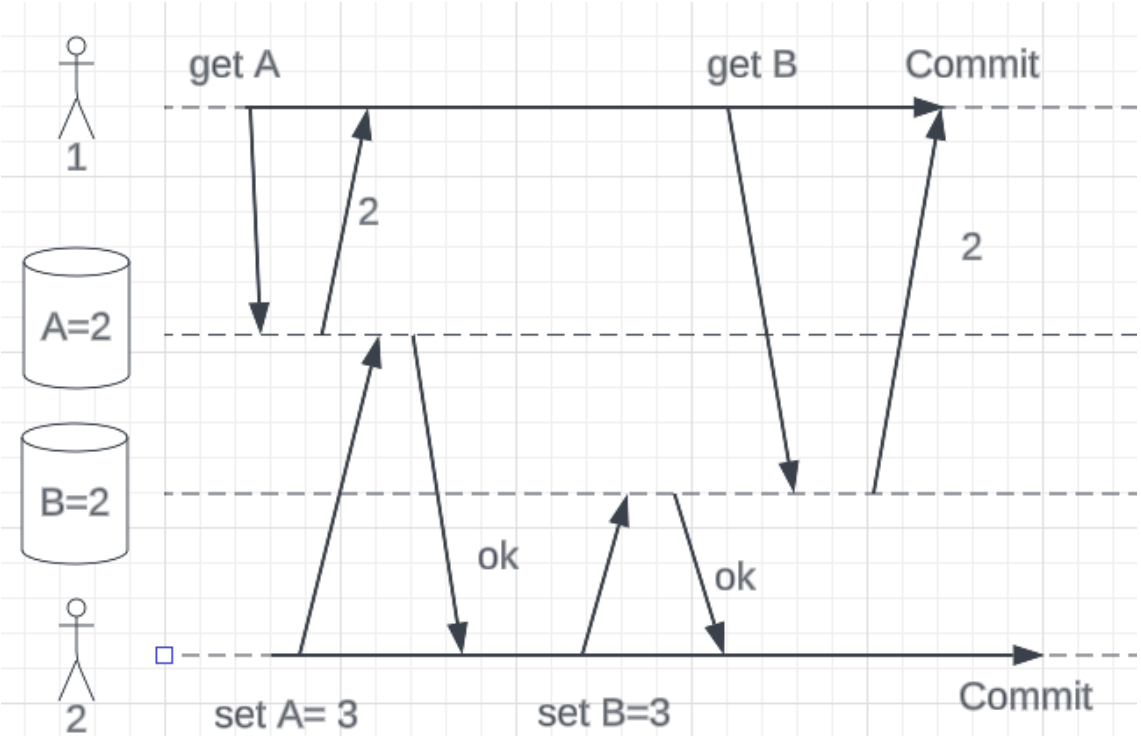

Figure 1: Read committed sequence

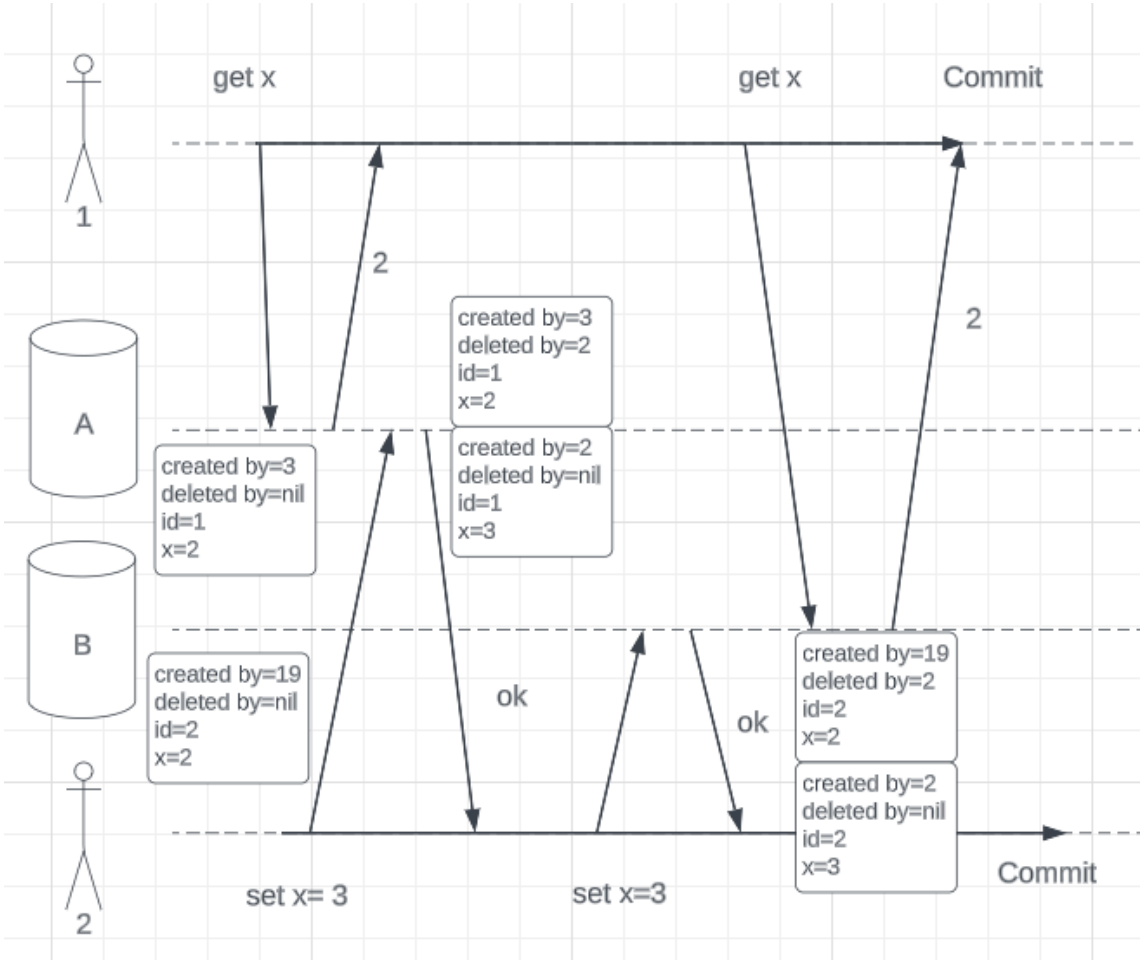

Figure 2: Snapshot isolation sequence

b

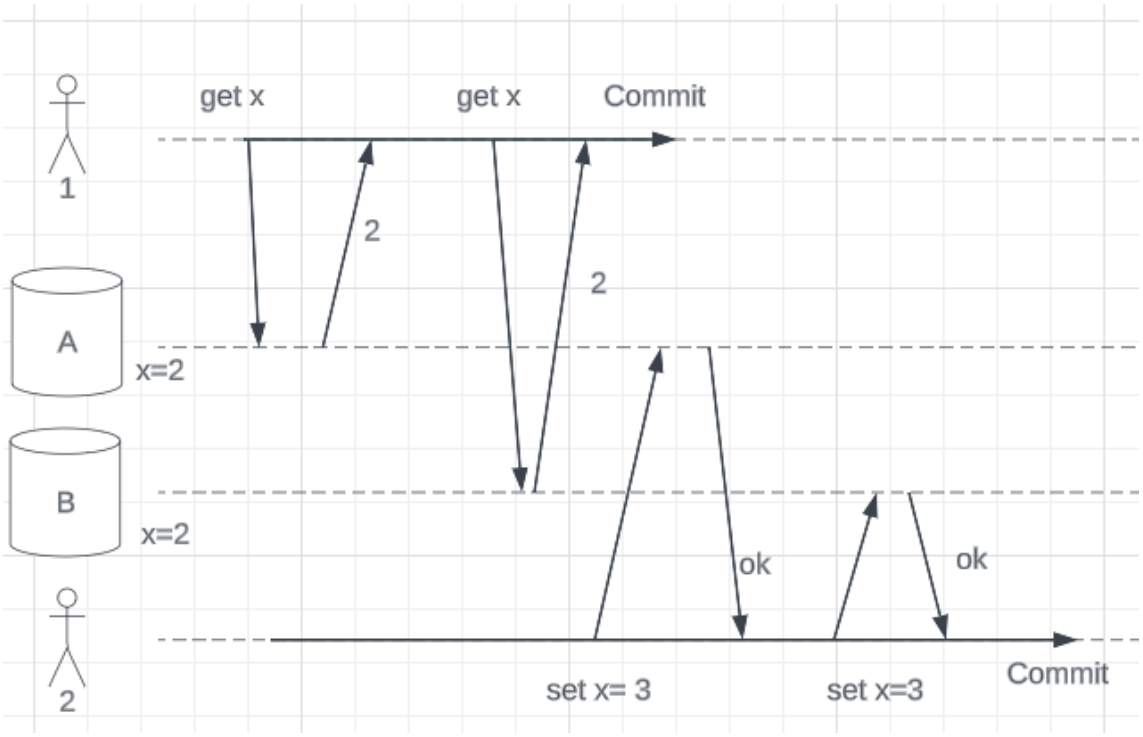

Figure 3: Two phase locking sequence

4. Kleppmann Chap 8

a

If you get no reply, here are some alternatives:

- Request has been lost
- Request is in queue, and is waiting to be delivered
- Remote node has failed
- Remote node have temporarily stopped working
- Remote note has processed request, but response is lost
- Remote node has processed request, but response has been delayed

b

Explain why and how using clocks for last write wins could be dangerous. Clocks, and especially time-of-day clocks can be dangerous when using last write wins, because clocks can be out of sync, and a previous write, may overwrite the actual last write. Clocks can often be unreliable and synchronized. Such as the quartz clock in a computer, can drift up to 17 seconds per day.

5. Kleppmann Chap 9

a

Linearizability makes the system seem like one copy of the data. Ordering gives a sequence of events or operations meaning, and introduces correctness and consistency. Consensus gives us algorithms for deciding when nodes have different data. They all give distributed systems a meaningful and correct order of operations.

b

Yes, there can be usable distributed data systems, which are not linearizable. Such systems can rely on eventual consistency. Eventual consistency rely on the idea that given enough time a system will eventually become consistent without the need for linearizability, which can be expensive.

6. Coulouris Chap 14

a

We cannot deduce that e-¿f, event though L(e)¡L(f). This is because events could have happened on different processes. For example if e is the only occurring event on p1, it would have a value of 1. But if events happened before f on p2, f would have an value of 3. f could still have taken place long before e. However if we use vector clocks, we can deduce than e happened before f. This is because vector clocks, takes other processes into account, where each clock value is a n-long vector, where n is the amount of processes.

b

b = (4,0,0)

k = (4,2,0)

m = (4,3,0)

c = (4,3,2)

u = (4,4,0)

n = (5,4,0)

c

Could not figure this out

RAFT

RAFT ensures that all participants in the event of a crash. Raft uses a leader-based approach for its consensus module. At any time, a participant is a leader, follower or a candidate to be the next leader. Time is then divided into terms, and under each term at most one leader is elected. The leader is responsible for sending out *heartbeats*, which signalizes that is it alive, if the electionTimeout elapses, a new election is started. The server then votes for a new leader, which is elected. This means that at some point, all candidates have won.

When the term has an elected leader, normal operation proceeds. A client will send a command to the leader, and the leader will append the command to its log. The leader proceeds with sending out a AppendEntires RPC to its followers.

If there any log inconsistencies, the new leader must either delete or fill in missing values in the followers logs.

This makes every log be consistent in case of a leader crash.

MySQL RAFT

One key takeaway was RAFTs performance in case of fail over, with the old semi sync the system did a fail over in 20-40 seconds, but RAFT typically does it within 2 seconds. It also simplified the operation and made the MySQL server take care of promotions and memberships. Lastly it introduced huger reliability, provable safety. All this with equal or comparable write performance.