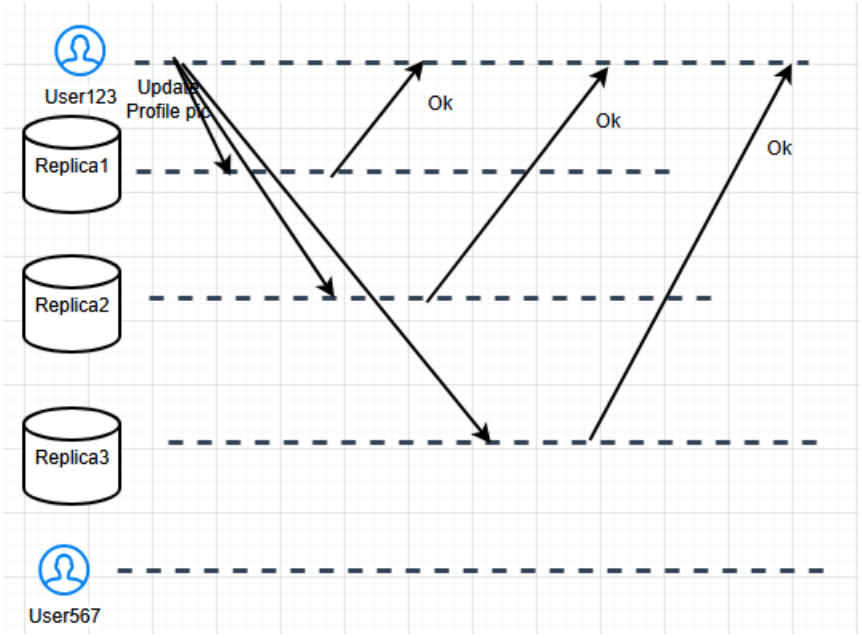
Replication5

There are conflicts in multi-leader application we got to handle

Leaderless replication:

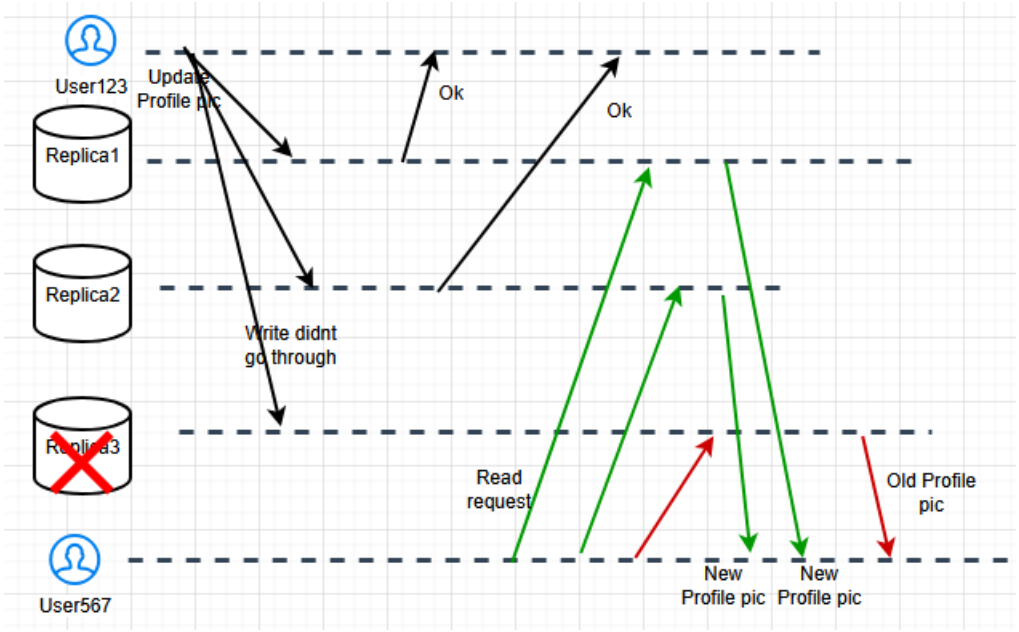
Here, we are talking about entire dataset that need to be replicated.
In Leaderless replication, there is no leader assigned to any node. That means, we can write to any node and we can read from any node since there is no follower also.
- we don’t have any Leader or Follower
- It is used in mostly all organization including Amazon Dynamo, Cassandra, Riak, Voldemort



- Write to multiple replicas
- Read from multiple replicas

Drawback is, it will have certain amount of latency
Chance of getting stale data also

Imagine Replica3 is down and it is not available. There could be multiple reasons like server down, network failure, no memory, etc. So our operation cannot be performed on Replica3. Replica3 is offline and only two Replicas are up and running.



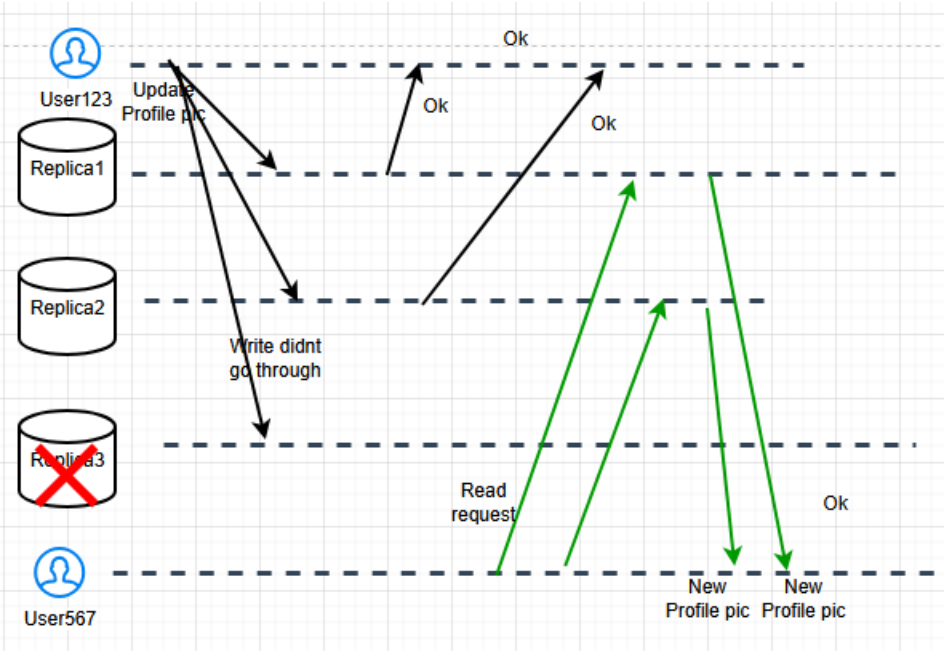
Now User567 is reading from Replica1, 2 and he sees Updated picture but he sees the old pic from Replica3, which was down during the write/update operation earlier.

Updating Offline DBs:
- Read repair: in Read repair, when we read from Replica1 and 2, we will automatically update Replica3. Or we update the node from which we are not getting correct values. While reading from updated nodes, we will send an update query to update stale information in the node that was offline. Whenever we do the Read operation, we will update the other non-updated nodes. It works for values that are frequently updated. For example, Voldemart uses this

- Anti-Entropy Process:
Instead of user operations, it will be background processes where Replicas update themselves in the background.

We cant read from all the Replicas, especially if we have a large system, there is a fair chance that one or the other Replica will go down. To handle such situations, we have something called as “Quorum”

Quorum:
Lets say n = number of nodes
w = number of nodes on which will confirm the write operation
r = query at least from this count of node



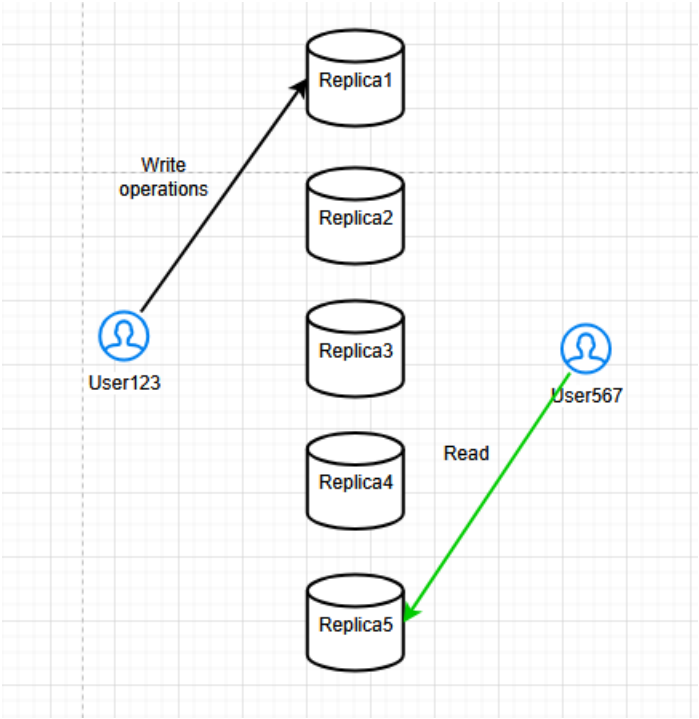
In this case, n = 3, we got response only from 2 nodes while writing so w = 2 and r = 2
Case 1: n = 3, r = 3, w = 1
We are reading from every node and writing to one node only. In this case, write operations will be faster. For Read operation, it will be slower because we read from every Replica. When we say we read from 3 nodes, it means that only if all 3 nodes respond Ok we will show the data to user, otherwise, we will retry. If any one of the nodes doesn't respond, we will roll-back the entire operation.

Case 2: n = 3, r = 1, w = 3
Read will be faster, Write will be slower.
Similarly, for Write operations, we will confirm the user that the data is updated only if all 3 nodes respond Ok, otherwise we keep retrying, which is time-consuming. If any one of the nodes doesn't respond, we will roll-back the operation.

Case 3: n = 3, r = 2, w = 2
What it means is, we stop reading nodes, even if we get Oks from 2 out of 3 nodes.
Both Read and Write will be faster in this case, compared to previous cases

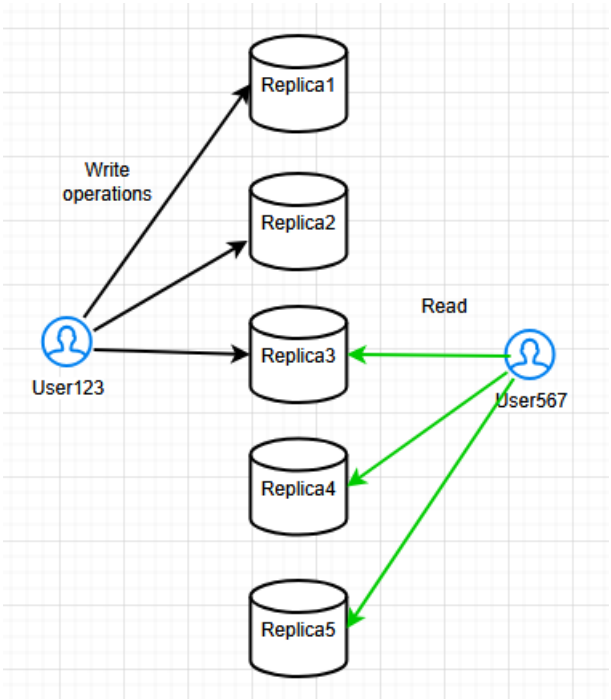
Quorum theorem states that if you have 'n' number of nodes then your r + w > n (read + write should be greater than number of nodes) then it is a good system

in another case, we say r > n/2 and w > n/2



How about the case? r=1, w=1. We respond ok even if we get one Ok . We could read from Replica5 while write is happening in Replica1 that means we could be reading stale data and that's ok.

Consider the second case: n/2=5/2, r=3, w=3. r > n/2 and w > n/2. In this rule, when it is > n/2 atleast one node will give us the updated data. Previous case doesn't work, thats why we add the second case. Replica3 is the common updated node.



How do we know Replica3 is the updated value? Because other two values are different. Maybe, we will have a Timestamp also. Lets say Write operation was performed at Timestamp: X, we see whether

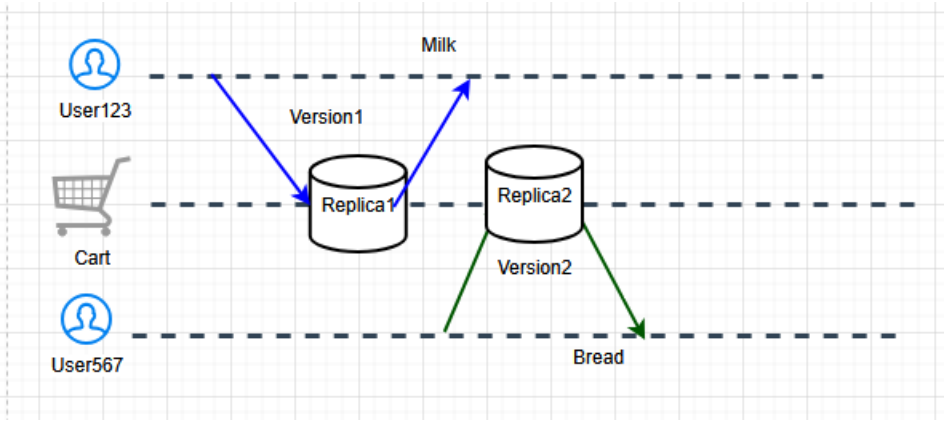
we have any updates in values since X in different Replicas. Whichever Replica has the most recent operation since the Timestamp, we pick that value. We are not considering Timestamp of Read but we are considering Timestamp of Write. However, if the common updated node goes down, then we will have some issues.

Concurrency

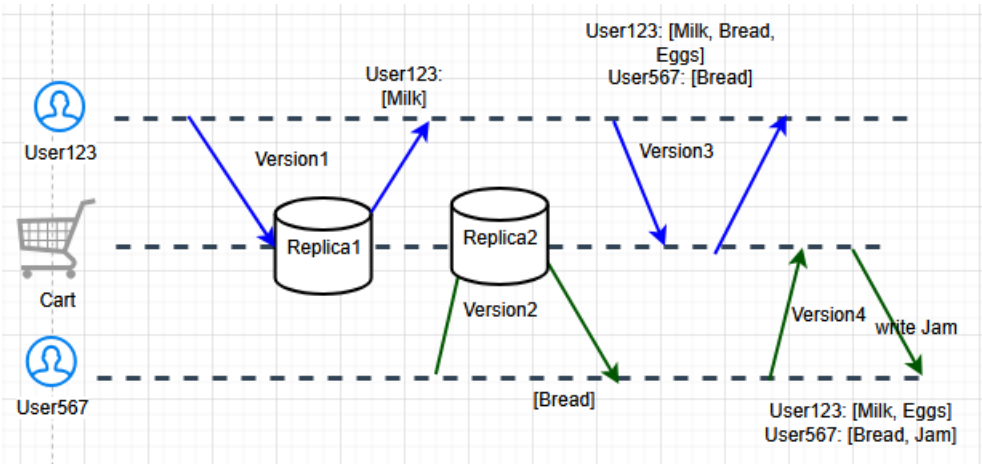
How do we know the data we are fetching is in progress of getting updated/changed?
How do we know the data we are fetching is not changing, at the time of read?

Write was performed way before Read was performed.
1. W < < < R ===> in this case, Happy to read anything. We will always read the latest value only
2. R < < < W < < R ===> we performed Read long time back then Write then again Read. While performing Read operation and we know that no frequent Write operations have been performed on that chunk of data then we can safely say we will always Read the updated value only.
3. R + W = at the same time ===> Concurrency issue

Bit of a complex issue to wrap your head around. Say two users share the same Instacart account and adding items to the cart. User123 added Milk to the cart and Write operation happened on Replica1 and User567 added Bread to the same account cart but the Write operation happened on Replica2 and Replica2 update didn't happen while the second Write was performed. That means, based on the Timestamp of the second Write operation, Milk is now removed from the Shopping Cart, which is incorrect.



This is more like the Version control problem, say in Version1 we have only Milk, Version2 only Bread and in Version3 we have Milk, Eggs and Bread. All are different
To manage Concurrency, we will have a VersionVector



We have a VersionVector for User123 and User567. Then when there is no changes for sometime, we take both VersionVectors and merge them into one. [Milk, Eggs, Bread, Jam].