In this part, I am going to show some result of the MongoDB team did on exploring the different latency under different write concern levels.

In real developing environment, there is nothing ideal, we must make trade-offs. In this very context, the trade is being made on weather you want strong durability guarantees or low latency.

They setup 3 types of scenarios on 3-node replica sets using different geographical distributions of replica set members to demonstrate. Each experiment performed 100 single-document updates. All operations specified that journaling was required in order to satisfy the given writeConcern.

The experiment is targeting 2 variables, writeConcern values and different client locations, where applicable, utilizing the variable control method to analyze the impact on the latency due to variable change.

First, they are testing the latency due to different writeConcern, all replicas set members and the client were in the same AWS Availability Zone (roughly the same datacenter) and Placement Group (roughly the same rack). All replica sets members were running MongoDB 4.0.2 with SSL disabled.

------------------------------------------------------------------------------

( 看图 )

Chart

Description automatically generated with medium confidence------------------------------------------------------------------------------

Requests acknowledgment that write operations have propagated to the majority of voting nodes

------------------------------------------------------------------------------

（explain）

Average latency measured across many operations is not the best metric. Developers who want to ensure a consistently great, low-latency experience worry about the worst performing queries in their deployment. High latency queries are measured at the 95th and 99th percentiles – where observed latency is worse than 95% or 99% of all other latencies. (One could argue these are insufficiently precise – most web sessions involve hundreds of requests, and so it is very likely that most users will experience latency at the 99th percentile during their session.)

（<https://www.mongodb.com/blog/post/performance-testing-mongodb-30-part-1-throughput-improvements-measured-ycsb>）

------------------------------------------------------------------------------

Second experiment, all replica sets members were in the same AWS Region (the same geographic area), but they were in different Availability Zones. Client 1 was in the same Availability Zone as the primary, and Client 2 was in

the same Availability Zone as a secondary. All replica sets

A picture containing chart

Description automatically generatedmembers were running MongoDB 4.0.3 with SSL enabled.

A picture containing table

Description automatically generated

------------------------------------------------------------------------------

（explain）

------------------------------------------------------------------------------

8.2: Quantifying w:1 Write Loss In a FailStop Model

As discussed previously, users may choose to trade off latency for consistency depending on their application requirements. We designed a simple workload simulation to estimate the degree of w:1 write loss in a replica set under standard failure modes. We developed an insert-only work- load that runs against a 3 node replica sets deployed on AWS in the US-EAST-1 Availability Zone. It has 30 concurrent writer threads that each insert documents into a test collection, which is initially empty. All writes are done at w:1 write concern. The workload client inserts as many documents as possible until exceeding a specified workload time limit, which was set at 1 hour for our experiments. To simulate failures, the MongoDB process for each replica set node is forcefully killed and restarted at periodic intervals. The processes are shut down with a SIGKILL signal.

The time intervals we need to identify is

(1) the time between a restart and subsequent shut down

(2) the time between shut down and restart, which

can be viewed as the "time to repair".

The MongoDB team using the result of an existing paper which state that the probability of fail over with time goes by following the Weibull distribution. And then they scale it down to let the single workload running in a reasonable time so that instead of days the team is using of scale of seconds. This gives a mean time between shutdowns of 54.16s.

We induced artificial network latency by using the Linux traffic control(\tc") tool to delay IP traffic between the replica set hosts. No latency was added between the workload client and the replica set hosts. This will also cause extra lost since the replica hosts still need to time to acknowledge the failure of the primary node, select the new primary node and broadcast it to other nodes to work.

Table

Description automatically generated

8.3: Failover Data from MongoDB Managed Replica Set Deployments, they were not able to directly measure the number of node crashes or restarts with the data collected, but we analyzed the number of unique terms seen

by each replica set over this time period. Recall that terms are monotonically increasing integral values maintained on each replica set node and are used as a way to totally order elected replica set leaders. There can be at most one leader for a given term. Thus, by examining the set of all unique terms over some period of time, it is possible to get a sense of how many elections that replica set experienced. In many of these managed deployments, it is likely that one of the most common sources of term changes is an election due to a planned maintenance event, e.g. upgrading a server's binary version. In these cases, an election is necessary since the node undergoing maintenance (e.g. the primary) must eventually be shut down, but a planned stepdown will wait for a majority of replica set nodes to be caught up to the primary before the stepdown succeeds. Thus, all writes that were present on the primary at the start of the stepdown should become majority committed, reducing the loss of any w:1 writes.

Notice that the mongod process dropped connections as a result of losing its Primary status. This is expected and normal. Dropping connections forces all currently connected clients to reconnect and thus refresh their understanding of the replica set status. This prevents clients from errantly sending writes to a former primary.

[（https://www.mongodb.com/blog/post/your-ultimate-guide-to-rolling-upgrades）](（https:/www.mongodb.com/blog/post/your-ultimate-guide-to-rolling-upgrades）)

The method then initiates a catchup period where it waits up to secondaryCatchUpPeriodSeconds, by default 10 seconds, for a secondary to become up-to-date with the primary. The primary only steps down if a secondary is up-to-date with the primary during the catchup period to prevent [rollbacks.](https://www.mongodb.com/docs/manual/core/replica-set-rollbacks/)

If no electable secondary meets this criterion by the end of the waiting period, the primary does not step down and the method errors.

Once the primary steps down successfully, that node cannot become the primary for the remainder of the stepDownSecs period, which began when the node received the method.

[（https://www.mongodb.com/docs/manual/reference/method/rs.stepDown/#:~:text=After%20the%20primary%20steps%20down,a%20secondary%20to%20catch%20up.）](（https:/www.mongodb.com/docs/manual/reference/method/rs.stepDown/%23:~:text=After%20the%20primary%20steps%20down,a%20secondary%20to%20catch%20up.）)

Chart

Description automatically generated