**Application Engineering**

**Introduction:** This week we go thru 3 topics that we need to know, to engineer(design) solutions using MongoDB.

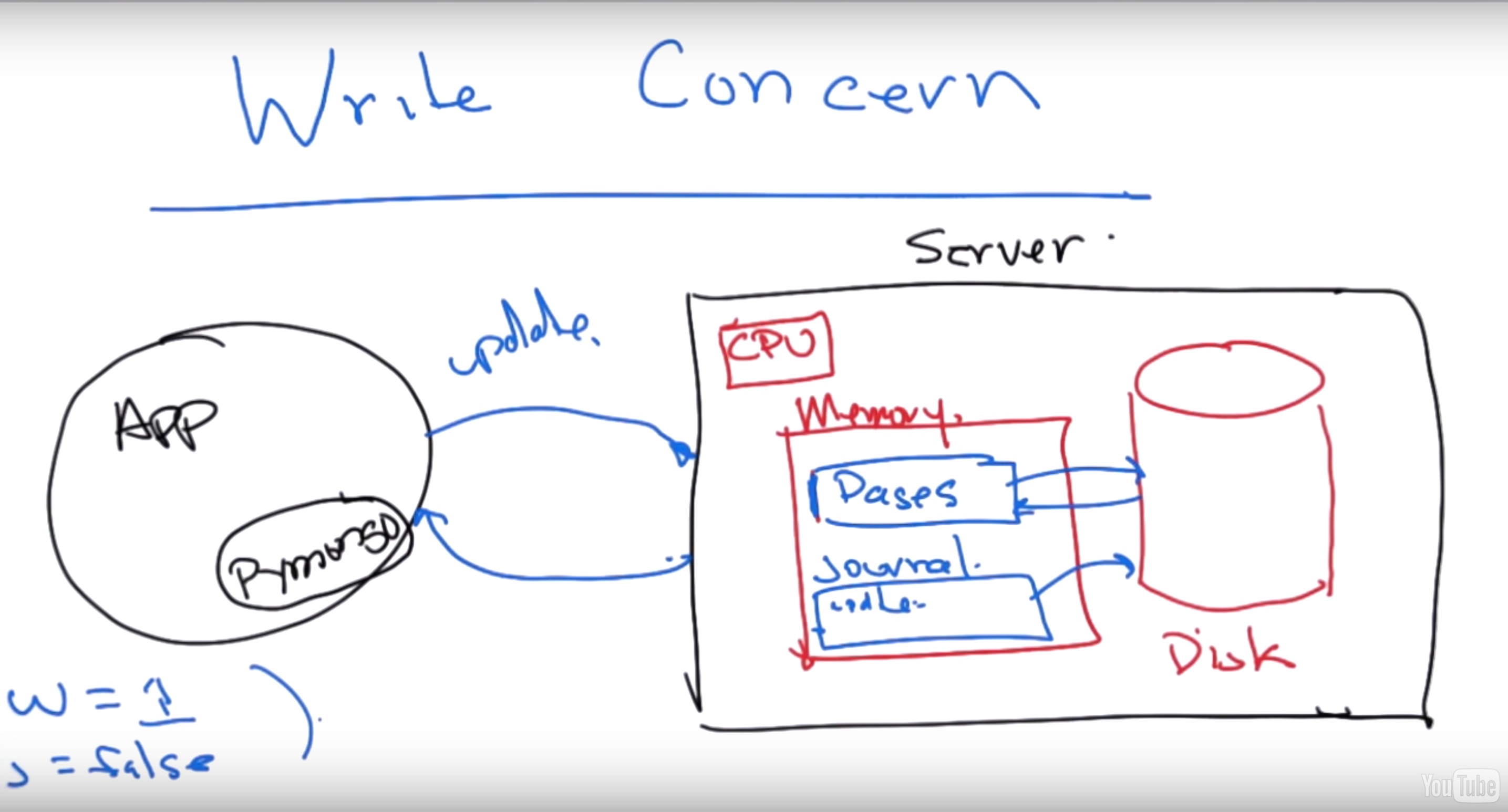
1. Durability of writes: How to we know that data is persisted into hard disk.
2. Replication : MongoDB approach to fault tolerence and durability.
3. Sharding: Using sharding, we can distribute a collection on multiple servers to gain greater throughtput.

Now this is a course for developers, we gonna talk about how we gonna replicated sharded system and does not show into how to configure the replication and sharding.

The [durability](https://en.wikipedia.org/wiki/Durability_(computer_science)) property ensures that once a transaction has been committed, it will remain so, even in the event of power loss, [crashes](https://en.wikipedia.org/wiki/Crash_(computing)), or errors.

**2) Write Concern :**

Lets talks about how we make sure that the writes which we make to the database actually persist, so that when we read from the database, we can see them.



Now we have out appication, and this application is going to be using driver , i.e MongoJava driver, which is part of the application and appication is talking to a database server, which is represented as a rectangular box of the above image. In this lesson, we gonna assume there is single server but in the later lesson, we are going to talk about the replicated servers.

This server has several parts. It has CPU, and the CPU is running the mongoD program. And there memory and then includes the persistence disks.

Now, the database is writing to memory. Let us consider a wiredTiger, where there’ll be a cache of pages in the memory that are periodically written and read from the disks, depending on the memory pressure.

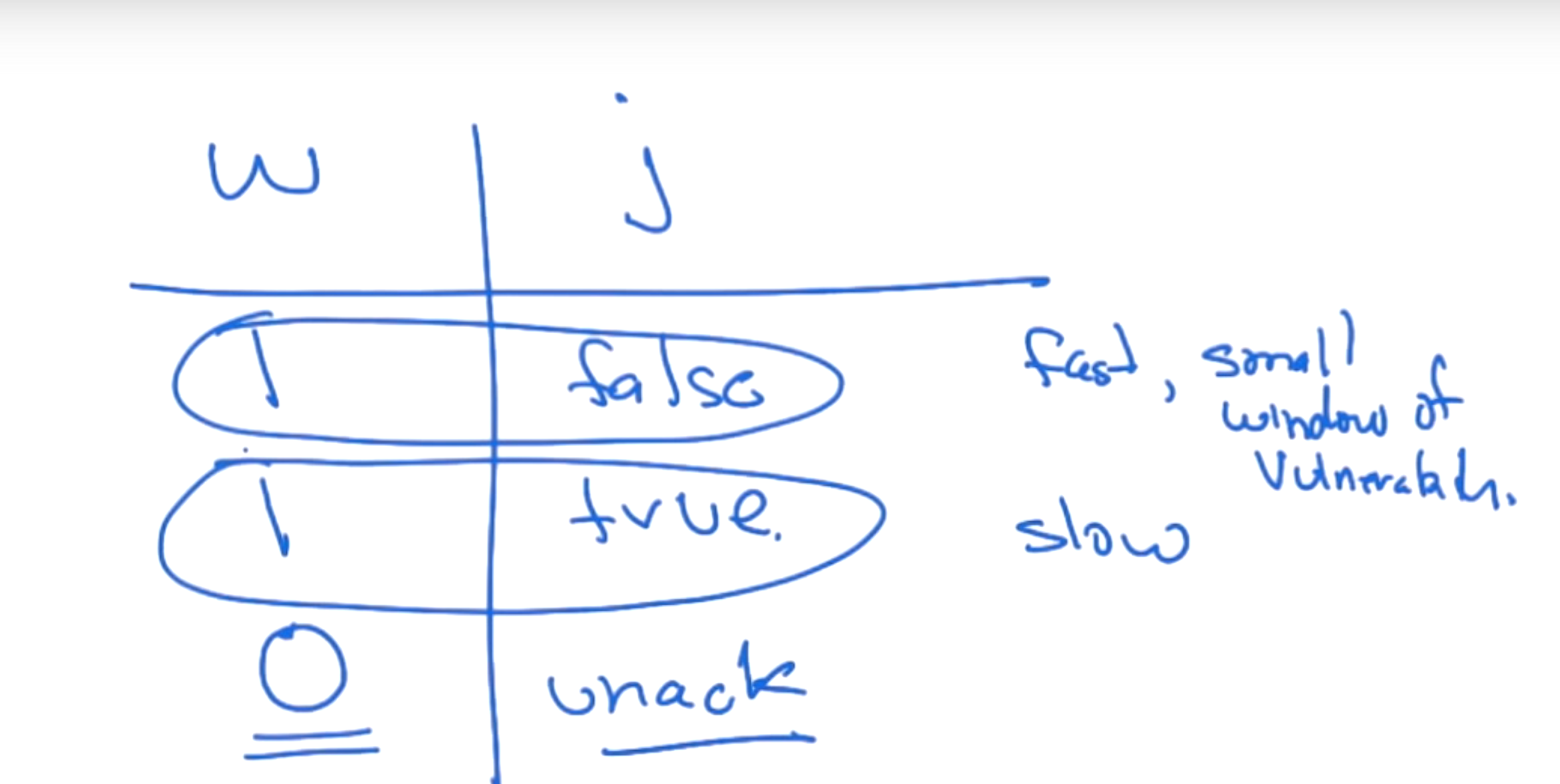
Now the secondary strucutre that called a Journal, which is a log of every single thing that the database processes. Journal is in the memory and writing back this data to the disk only complete the update or insert of data in the disk. So updating the data and logging activity in the journal to disk is the complete circle of write activity in MongoDb.

So, when we do a update or a insert then we are goint to connect to the server via TCP connection. And the server gonna process the update/insert and it write into the memory pages and written back to the disk based on the memory pressure. It gonna simultaniously write to journel.

By default, in the driver, when we do and update or insert, then we gonna wait for the response which is a acknowledged update or insert. But we don’t wait journal to be wrtten into the disk. The journal may not be written to memory for some more time.

The value that represent that we gonna wait for this write to be acknowledge by the server is called w, by default it is 1, which means wait for this server to response to my write. By default j value is false, which is the value for journal , which means we have to wait for this journal to be written to disk, before we gonna continue.

The inmemory data can be lost if the server is crashed for some reason. These value helps us to logs the data accordingly.



if the journal is saved correcly then it will allow to do the update/insert on data again even if the data is lost from memory.

First is default and is small but have a small window of vulnerablity and 2nd will be slow but no vulnerablity.

3rd is not recommended which is a un acknoledged write, where we don’t wait for any update.

Finally, the value W has more value in the replicated env.

J value represent true or false where W value is numeric value.

## Quiz: Write Concern

Provided you assume that the disk is persistent, what are the w and j settings required to guarantee that an insert or update has been written all the way to disk.

Top of Form

 w=0, j=0

w=1, j=1

w=2, j=0

w=1, j=0

Bottom of Form

**3) Network Errors**

As we have discussed the write concerns and also J and W value, we are pretty much sure that we know the status of update/insert . But unfortunatly it is not quite true.

Lets say j=true and w = 1 and do an udpate/ insert to mognoDb. And now if we get a affirmative response, then we know that it definitly happened.

What if we don’t get a response, we assume that it didn’t happened. But it might have happened and network errors the message not reached. Which results in an errors in the driver code.

Generally, This case is okay if the objectID is generated by the mongo Drvier. If we wanted to re-insert then simply it will unique key contraint and will not be inserted.

However, in update we have the problem. Assume that we $inc to increment the value. If we are not sure about the earlier value and then retrying the same update statement, which may increase it by twice . In practice if the network is working well then this scenario is very rare.

However, we can avoid this by turning all the updates as inserting new docs. We have fetch the old docs and then modify the column and then insert it again.

**4) Replication :**

We have talked about how we get a durability on single node by waiting for the write, let say, to go to the log.

How we get

1. Availability :: What is ment here is that, even if the node goes down then also we still be able to use the system.
2. Fault Tolerence :: if we completely loose the primary because of some fire, how we have to make sure that the data is not lost in the backups

And what we do to solve both the problems is replication and this is how it works.

In MongoDB, we have the concept of Replica Set, and the replica set is the set of mongo nodes. For this example, let us consider it as 3. All these are mongoD that act together and all mirror each other in terms of data.

There is one primary and others are secondary which is dynamic. And data that is written to the primary will asynchronously replicate to the secondaries.

Our application and drivers stay connected to the primary and will write to the primary. If the primary goes down, then the remaining nodes will perform the election to elect the new primary. To elect the primary we should have a strict majority of the original number of nodes. So since the original number of nodes are 3 then we need 2 nodes to elect the primary and that’s the number we have. After this, then one of 2 server will become primary and this point, the application would reconnect to the new priamry for right, through the drivers. All this is done transperently.

This concept of replica set will provide us avaibality and fault tolerence in mongoDB.

When the actual primary came back up then it will act as one of the secondary by rejoining the group.

The minimum no of nodes for replica set is 3. And the reason is, if we have fewer than 3, what would remain is not the majority of the set. So there is no way to elect a primary. Which means we have go without a primary. i.e if one goes down and only 1 is left and this cant be made primary because there is no secondary to back up the data in primary, then no concept of replica set.

**5) Replica Set Election**

In this section, we gonna discuss about the election that goes on when the primary goes down and the types of nodes that gonna present in a replica set.

1. Regular: this is the node, which has data and can become primary. And it’s the most normal type of the node. It can be a primary or secondary.
2. Arbitary: This node is just there for voting purpose. And we have lots of reason that we gonna use one of these.If we want to have a even number of replica set nodes, and then we have to make sure that there’s should be an aribitary node, to make sure that there a majority to select the primary if one goes down.
3. Delayed node: A disaster recovery node which can be set to 1 or 2 hours behind the other node and it can participate in the voting but it cannot become the primary. And to achieve this its priority is set to 0.
4. Hidden node: which is used for analytics. It cannot become the primary node. And so it priority is set to 0 and can participate in elections.

How to decode the votes for each of this nodes have?

We will assume that every node has one vote because in reality it is not correct to give more than one vote. When the primary goes down, then the other nodes will decide the primary node and this is trasperent to the application.

## Quiz: Replica Set Elections

Which types of nodes can participate in elections of a new primary?

Top of Form

Regular replica set members

Hidden Members

Arbiters

Lawyers

Bottom of Form

**6) Write Consistency.**

In the replication of mongoDB, there’s only a single primary at any given time. And in the default configurations, our writes and reads go to the primary. Now, our writes has to go to the primary but your reads don’t have to go the primary. The reads can go the secondary if we like to do so.

But if we allows writes go to the primary then we will have a strong consistency of reads with respctive to writes. And what this means, among other things is that we don’t read stale data. That if you write something, then you will be able to read it and othere applications servers that read it will also be able to read what we wrote after we wrote it. Providing we make sure that write is completed through journalling.

Now, we can, if we prefer, allow our reads to go to your secondary, but we may be reading stale data from secondaries, relative to what we wrote or somebody wrote in the primary. And the lag between any 2 nodes is not guaranteed because the replication is asynchronous.

We will go thru the different read preferences that we can set in terms of the drivers to decide wheater or not to accept reads from the secondary. And the reasons, why we might want to read from the secondaries because we want to scale the reads through the replica set.

On the other hand when the failover occur, briefly during that time, theres is no primary and you can’t complete a write and this continues still the primary is elected.

This is contract to some other system that compete with mongoDB that have a weaker form of consistency which is called “Eventual Consitency”, which means eventually, we will be able to read what we wrote, but there is no guarantee that we will able to read it in any particular time frame. The problem with eventual consistency is that it is very hard to reason about, because when we write and most application servers are stateless, and saving the session data and fetching back may get different data and then have to reconcile what that means.

SO mongoDb does not offer eventual consistency in its default configurations when we read and write to the primaries. But if we want to from a eventual consistency we can read it from the secondary, which will give you evetual consistency.

## Quiz: Write Consistency

During the time when failover is occurring, can writes successfully complete?

Top of Form

 Yes

No

Bottom of Form

In this case exception is throw in the program. We have catch these exception and retry. Usually the failover is about 3 sec and not so common.

**7) Creating a Replica Set:**

Let us start creating a replica set on just single machine. Usually in real world, we keep each mongoD of replica set on different servers so that there is real fault tolerence.

In this example, we are running mongoD on a single node with different ports number so that they will not conflict with each other.

mkdir -p /data/rs1 /data/rs2 /data/rs3

mongod --replSet m101 --logpath "1.log" --dbpath /data/rs1 --port 27017 --oplogSize 64 --fork --smallfiles

mongod --replSet m101 --logpath "2.log" --dbpath /data/rs2 --port 27018 --oplogSize 64 --smallfiles --fork

mongod --replSet m101 --logpath "3.log" --dbpath /data/rs3 --port 27019 --oplogSize 64 --smallfiles --fork

1) First we have create 3 directories for each of the mongoD instance.

2) Then we have to start the mongoD with option as –replSet which sets the group of replicaset and –logpath is the location of the log file and –dbpath is the location of data to store, --port is port.

These will run with bash shell. With windows this may change a bit. This will start all the servers but these will not have idea of each other. TO make them in a replica we have to run command from mongo shell.

config = { \_id: "m101", members:[ { \_id : 0, host : "localhost:27017"}, { \_id : 1, host : "localhost:27018"}, { \_id : 2, host : "localhost:27019"} ] };

rs.initiate(config);

rs.status();

In order to tie all the instance of mongoD to a replica set we have to run the about command.

rs.initiate(document) : This command should be called with the configuration documents. We can’t run the initiate command on a host that does not become a primary.

document include the \_id value which should be unique and also it represents the name of replica set. And then we have members array which gonna name three members, and these id are arbitary. We can also set the priority for the members using priority:0 and slaveDelay: 5 to set the delay latency.

Mongo –port 21018 :: allows us to connect to shell of particular port if mongod is running on multiple port.

rs.status: this is used to check the status. This will return the number of nodes in the replica set.

Now we can write to primary and login to secondary and try to read. MongoD will throw an exception when we are trying to read from secondary because the secondary servers are not configure for read by default.

rs.slaveOk(): this will allow the particular secondary slave to query it but not write.

**8) Replica set internals.**

Now we have a three-node replica set, each of these a mongoD and each of this has within it an oplog collection and the oplog will be kept in sync by mongo.

So if one of these is primary, then the writes are done to the primary. And our secondary will always reading the content of the oplog of the primary.

when we do a write to primary then we gonna written to the oplog and then the secondary is going to be creating what’s new in the oplog and applying the same operation on the secondary. And when the election occurs and new primary will be elected and other when comes up then that will become the secondary.

Now let us this on live:

To the check the mongoD process running we can run the shell command as follows

ps -ef | grep mongod

Open the primary node by checking the status and login with

mongo —port xxxxx

now write something to primary.

db.people.insert( {‘name’ : ‘andrew’} )

now let us the check the oplog collection. oplog is in the local database.

use local

show collections.

which will have collection name oplog.rs

so lets see the collection.

This collection includes the log of creating the collection of people and also the insert.

Lets go to secondary and check what going on there.

we can see the collection already available there with the same data and also oplog.rs in the local db is also updated to the latest.

Each of the database know how long they are sync and so it asked the primary for what ever it is new. if we check the status then it has the prop of optime and optimeDate and also syncingTo which is the primary.

Note: oplog is a cap collection and it will rollout after some amount of time. So we need to have big enough oplog to be able to deal with periods where the secondary can’t see the primary. so how long the secondary will be depends on how long we might expect there to be a bifurcation in the network and also how much data we are writing and how fast oplog growing. So in a very fast moving system, the oplog may be very large oplog.

However, we can resyync the secondary by reading the complete database of primary but this is much much slower.

Oplog uses the statement base approach. where these are usually mongoDB documents . it does not matter which storage engine we are using and not even which version of Mongoldb is using. and so we can have mixed mode replica set. This allow us to do the upgrade of a system in parts but not all at a same time.

Quiz: Replica Set Internals

Which of the following statements are true about replication. Check all that apply.

Top of Form

a) You can write to a primary or secondary node and the database will forward the write to the primary.

b) Replication supports mixed-mode storage engines. For examples, a mmapv1 primary and wiredTiger secondary.

c)A copy of the oplog is kept on both the primary and secondary servers.

d) You can read from a primary or secondary, by default.

e) The oplog is implemented as a capped collection.

Bottom of Form

**9) Failover and rollback.**

in this we gonna discuss about a failover situation where some data which is committed previously will result in a rollback.

Imagine we have 3 servers running in a replicaSet and one of the these is primary and others are getting replciated out with a delay latency of few sec. And all a sudden the primary node goes down and other one will become the primary. But there are writes on the previous primary node which is not yet replicated on to other servers.

Then after a period of time, the older primary will come up and become the secondary and it will now retry to sync the data from the present primary and realises that it has some more write then the present primary and then roll back these write which cause in the rollback of the actually committed data. But however, these rollback write are put into a file which can be applied manually if required and but they won’t be part of dataset.

We have a way to avoid this particular scenario for the most part by waiting for the write until majority of the nodes have this data and hence this vulnerability won’t exist. This will avid most the cases.

So, by seeing the write concern (w) value to majority so that majority of the nodes have this value then it wont happen. So in default case of w=1 and j=1 , it is perfectly possible to loose some writes by rolling them back.

while rolling back, it will keep the operation to apply manually but in most cases it will be complicated to apply and so we usually avoid it.

Lecture Notes

While it is true that a replica set will never rollback a write if it was performed with w=majority and that write successfully replicated to a majority of nodes, it is possible that a write performed with w=majority gets rolled back. Here is the scenario: you do write with w=majority and a failover over occurs after the write has committed to the primary but before replication completes. You will likely see an exception at the client. An election occurs and a new primary is elected. When the original primary comes back up, it will rollback the committed write. However, from your application's standpoint, that write never completed, so that's ok.

Quiz: Failover and Rollback

What happens if a node comes back up as a secondary after a period of being offline and the oplog has looped on the primary?

1. The entire dataset will be copied from the primary
2. A rollback will occur
3. The new node stays offline (does not re-join the replica set)
4. The new node begins to calculate Pi to a large number of decimal places

Bottom of Form

**10) Connecting to ReplicaSet from Driver**

In this chapter, we gonna check how to connect to replica set using mongo driver.

Lets assume that we have set a repleset with name as replset and we have 3 instances running. The first on port 27017 and this is the primary and there are 2 secondaries at 27018, 27019.

We can connect to the primary directly using the driver with the following code.

MongoClient client = new MongoClient(new ServerAddress("localhost",27017));

But assume that 27017 instance is down for some reason and in this case, we have to check that and then change the port to the new primary, let say, 27018. Other wise, the driver will be waiting for a primary connection . we can use rs.stepDown() in primary terminal to simulate the instance down.

MongoClient client = new MongoClient(new ServerAddress("localhost",27018));

Mongo Replica set are automatic failovers, we would like to advantage of that when using the driver. We can do it simply by passing the list of servers address to mongoClient.

MongoClient client = new MongoClient(Arrays.asList(new ServerAddress("localhost",27018)));

if you pass the list of mongo servers and if the servers are part of a replicaset then it will go into replica set mode and dynaically look for the primary and any secondaries. In this case, 27018 is secondary but it is part of replica set and so the driver will figure out the primary automatically as 27017 and insert data.

This solution is ok when the servers are stepDown but if the process are killed then drivers will not be able to reach it because the process is down. So this does not work. To safeguard this situation the best practice is we have to specify the members of the replica set as list to mongoClient. MongoClient will automatically figure the primary from the list.

MongoClient client = new MongoClient(Arrays.asList(new ServerAddress("localhost",27017) ,

new ServerAddress("localhost",27018),

new ServerAddress("localhost",27019)));

If we enable info level logging, we see interesting info in the logs.

1. Driver frist creates a cluster. This is the client side view of the cluster with the specified host in the list and mode as multiple.
2. Then it will slowly discover the servers 27017, 18,19.
3. Then it will connect to the each individual server and then discover the type of the cluster as replicaset and the primary of all the servers.
4. Then starts inserting the data.

**Note:** One last bit of protection we can give is to pass the name of the replicaSet as parameter to MongoClient and this will check the all the member , which are passed as list, are part of the same replicaSet. If not, then it will throw a error and does not connect. This will avoid us from connecting to other replica servers. We can do this mongoClientOptions

MongoClient client = new MongoClient(Arrays.asList(new ServerAddress("localhost",27017) ,

new ServerAddress("localhost",27018),

new ServerAddress("localhost",27019)),

MongoClientOptions.*builder*().requiredReplicaSetName(“replset”).build());

**11) What Bad things can happen to the good nodes:**

In any distributed system i.e MongoDB, things may go wrong.

Let us simulate a situation, where java driver continiously inserting the documents and at the same time, kill the parent node. In this case, Java driver throw a socketTimeOut exception. As SocketTimeoutException is a non-catchable exception, so java compiler will not cause any issue even if the code is not wrapped in try-catch block.

So when this happen, the mongo client will get disconnect with the server and execution will stop.

To resolve the issue, we have to just catch this exception while inserting the document.

When we catch the exception, then the MongoDriver will automatically check the new primary that is elected and then connect to it automatically.

The insert at the time of exception will not be reached by the server but the other inserts will be fine because new connection made.

Thus, if we catch the exception while inserting, then the driver will reconnect to the new primary even if the old primary is lost. This small code will help to recover from such situations.

**12) Re-visiting write concern.**

The W parameter determine how many nodes we have to wait for before we move on when we do an insert.

In the simple case, where we have 3 nodes in a replica set,

Then setting w=1 , then will wait still the primary acknowledge the write. And setting it 2, then we gonna wait for primary and then one other node to ack the write. When w=3, will wait for all 3 to ack the write.

Then setting J = 1, then the journal is written to disk.

Now, How long we wait is called wTimeOut. This also includes the time that we need to wait for the secondaries to ack the write. This can be set in the drivers.

The value of w,j and wTimeOut are called writeConcern.

These values can be set at 3 places

1. on a connection
2. On a collections inside the drivers
3. While configuring the replicaSet. These are defaults i.e if no values come in then the replicaset will handle it this way. This is the safest way to do in the point of system adminsitrator. Bcz we cant expect developer to be using the default that we except them to use.

But in any event, we can do any of these 3 types.

There is couple of ways to use the “w” value

W: majority :: It waits for the majority of the nodes in the replicaSet will ack the writes. This is wait need to avoid roolback when failover is caused. While writing to disks and suddenly primary went down then the node which has already ack the present write, will become a primary in the replicaset.

J is only for the primary node, we don’t wait for it to commit to the nodes of the replicaset.

Lecture Notes

Write concern (w) value can be set at client, database or collection level within PyMongo. When you call MongoClient, you get a connection to the driver, but behind the scenes, PyMongo connects to multiple nodes of the replica set. The w value can be set at the client level. Andrew says that the w concern can be set at the connection level; he really means client level. It's also important to note that wtimeout is the amount of time that the database will wait for replication before returning an error on the driver, but that even if the database returns an error due to wtimeout, the write will not be unwound at the primary and may complete at the secondaries. Hence, writes that return errors to the client due to wtimeout may in fact succeed, but writes that return success, do in fact succeed. Finally, the video shows the use of an insert command in PyMongo. That call is deprecated and it should have been insert\_one.

**13) Read Preference**

Home Work:

## Homework: Homework 6.1

Which of the following statements are true about replication in MongoDB? Check all that apply.

Top of Form

 The minimum sensible number of voting nodes to a replica set is three. MongoDB replication is synchronous.

By default, using the new MongoClient connection class, w=1 and j=1.

The oplog utilizes a capped collection.

Bottom of Form

## Homework: Homework 6.2

Let's suppose you have a five member replica set and want to assure that writes are committed to the journal and are acknowledged by at least 3 nodes before you proceed forward. What would be the appropriate settings for w and j?

Top of Form

 w=1, j=1

w="majority", j=1

w=3, j=0

w=5, j=1

w=1,j=3

Bottom of Form