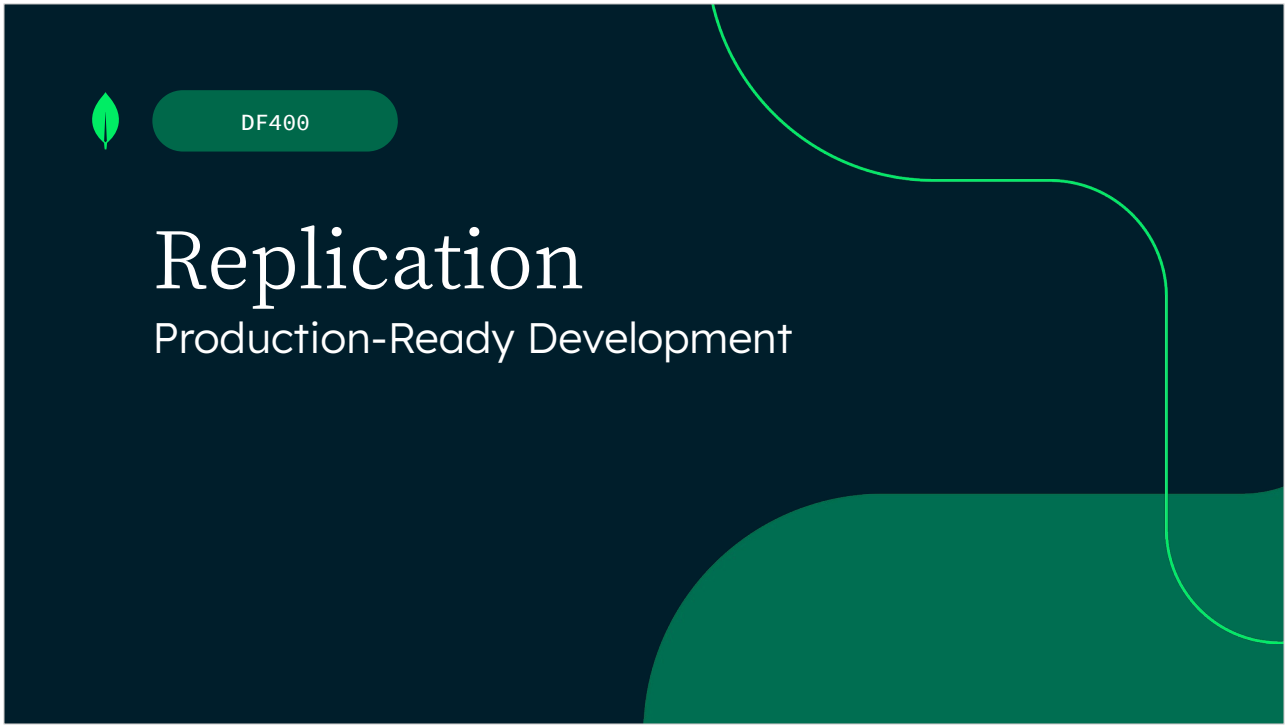


Release: 20230414

Topics we cover

Replication

Type of Nodes in Replication

Concept of Majority

How elections happen

Read concerns and preferences

Replication

- Multiple copies of each collection
- On different physical servers
- As far away from each other as possible
- Nature is the best example of replication for resilience

We want our data to be physically independent, ideally at least 100KM (60 miles) between copies.
We do need to consider latency and practical availability of hosting locations though.
We have multiple copies of the data, so if one is destroyed, the others are safe.

Reasons for Replication

High availability

Reducing read latency

Supporting different access patterns

These are explained in detail in the next few slides.

High Availability (HA)

Data still available after:

- Equipment failure (e.g., server, network switch)
- Datacenter failure

Achieved through automatic failover:

- Remaining servers have an election
- High availability is not 'Active Active' but fast failover

In simple terms - a server going offline suddenly is not noticeable to users
Anything up to 50% of the total infrastructure can fail, and the system keeps working.

Reduced Read Latency

Reading from a copy that is geographically near you

Speed of light matters

Some MongoDB global replica sets are up to 50 servers in 20+ countries - this supports a large base of application users who want to log in and read their preferences or personal data locally. Updates must always go via the master though so this model only allows geographically close reads.

Different Access Patterns

If 90% of the users look at the latest 1% of data - small enough to remain in cache = fast

If the rest (10%) look at all data (Analysts)

- Don't need such a fast response
- Shouldn't be bad neighbors - Their usage could affect the 90%
- Protect the cache and resources from these heavy users

In this situation, Analysts could read from secondaries to prevent affecting the "working set" of the primary.

Replica Set components

Primary Member
Secondary Members
Non-voting Members

A number of nodes can vote and become primary if appropriate.
The primary is what applications talk to to write and usually read from.
Secondaries are there as hot standbys.
The nodes choose the most appropriate Primary between them.
Some nodes can be flagged secondary and not eligible to become primary or vote.
Usually there are three voting members in a replica set. Sometimes there are five (max seven) - MUST be odd to avoid hung elections and no clear winner.
There can be up to 50 total members in a replica set - but no more than 7 voting members or the voting process would take too long.

Primary Member
- Elected by consensus
- Handle all write operations and most reads

Secondary Members
- Handle only read operations
- Help to elect a Primary
- May not have the latest data

Non-voting Members
- Hold additional copies for analysis or similar
- Do not define what will be the Primary - provide an odd number of voting Members

Secondary Server

A Secondary maintains a copy of the Primary's data set

Type of Secondary

- Priority 0 Replica Set Member
- Hidden Replica Set Member
- Delayed Replica Set Member

Note: Now Hidden and Delayed are considered Bad

Priority 0 nodes cannot become Primary

Hidden nodes are secondary nodes that an application cannot access by connecting to the replica set as a whole.

A client must connect directly to a hidden secondary by its hostname/IP - they cannot be used in a sharded cluster.
Hidden nodes must be priority 0, but may vote.
Hidden nodes are **not recommended** these days but are occasionally still used for making backups.

Delayed nodes delay playing replication operations to maintain a copy of the data which is not up to date.

The original purpose was to protect against user error - for example, dropping a collection would not immediately happen on the delayed node.

Delayed nodes are **no longer recommended** as a means of protecting your database as delayed nodes proved impractical to manage and use.

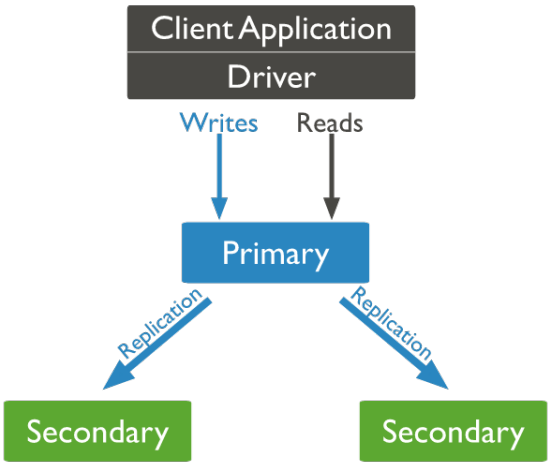
Drivers and Replica sets

When coding with MongoDB, Drivers keep track of the topology

- Drivers know where and how to route requests
- You don't need to know what is a primary or secondary at any point
- You can specify where you want a read to go logically
- When upgrading the server, it is important to check driver compatibility

Although the topology is discoverable after connecting to any node, the connection string needs references to at least one up and running node. The best practice is all voting nodes (although a majority will work). Over time, a replica set can evolve, nodes can be added and removed; the connection string you specify should keep up with that. Otherwise, one day, you might have only one original working node that still in the Replica Set, and when it's down, you would not be able to connect. Using an SRV record is preferable.

Replication process



- MongoDB achieves high availability by the use of a replica set
- A replica set is a group of mongod instances that host the same data set
- In a replica, one node is the primary node that receives all write operations
- All other instances, such as secondaries, apply operations from primary asynchronously.

Replication process steps

1. Applications write all changes to the Primary
2. Primary applies changes at time T and records them in its Operation Log (Oplog)
3. Secondaries are observing this Oplog and read those changes up to time T
4. Secondaries apply new changes up to time T to themselves
5. Secondaries record them in their own oplogs
6. Secondaries request information after time T
7. Primary knows the latest seen T for each secondary (This is important)

Oplog

The Operation log (Oplog) is a read-only capped collection of BSON documents

Each time a write changes a document, it's recorded in the Oplog:

- Records changes like dropping a collection or creating an index
- Is independent of the database's binary form
- Is in a database called 'local'; the collection is 'oplog.rs'

- The Oplog is a read-only collection of BSON documents it lives in the **local** database
- The system adds to it but not users
- The Oplog is a capped collection
- It has a fixed size; older data is automatically deleted (Starting in MongoDB 4.0, the oplog **can** grow past its configured size limit to avoid deleting the majority commit point)
- Each time a write changes a document, it's recorded in the Oplog
- There is one entry per document changed
- The entry refers to the document being changed by its _id
- The entry describes the new value of fields $set, not $inc
- Meaning, the oplog can be played again and get to the same place.
- The Oplog also gets other writes like dropping a collection or creating an index.
- The Oplog is independent of the database's binary form. It is logical statements not chunks of files So you can use it to replicate between different formats/versions.
- This is a big advantage for upgrading/downgrading compared to disk based or binary replication.

Oplog - one entry per change

Example: `db.foo.deleteMany({ age : 30 })`

This is represented in the oplog with several records (one per document):

```
{ "ts" : Timestamp(1407159845, 5), "h" : NumberLong("-704612487691926908"),
"v": 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 65 } }

 { "ts" : Timestamp(1407159845, 1), "h" : NumberLong("6014126345225019794"),
"v": 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 333 } }

 { "ts" : Timestamp(1407159845, 4), "h" : NumberLong("8178791764238465439"),
"v": 2, "op" : "d", "ns" : "bar.foo", "b" : true, "o" : { "_id" : 447 } }
```

Every entry in the oplog refers to an individual document by its _id or is a bigger change like 'create and index' or 'drop a collection'

Oplog entries are Idempotent

Oplog operations can be played multiple times as they do not depend on previous ones

Examples:

- `{$inc:{a: 2}}` becomes `{$set:{ a: 5}}` assuming a was previously equal to 3
- `{$push:{b: "cheese"}}` becomes `{$set:{ "b.8" : "cheese" }}`

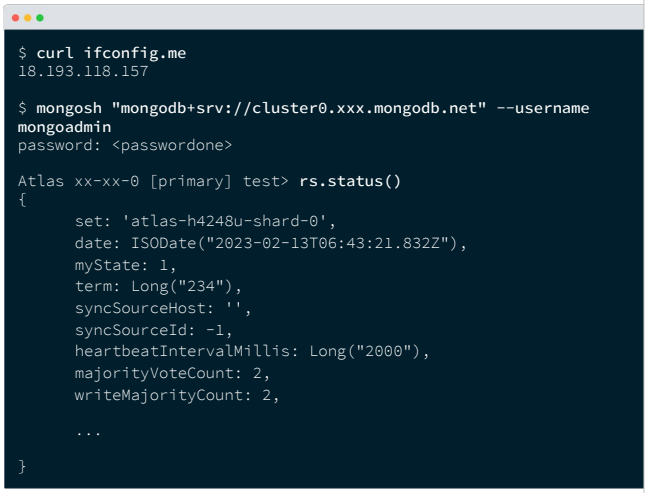
Each operation in the oplog is idempotent.
Whether applied once or multiple times, it produces the same result.
Necessary if you want to be able to copy data while simultaneously accepting writes.
If the database is restored from Time T and then you play the oplog to get it to T+5, this is required.
Otherwise, you would need to only play the oplog from exactly Time T - this way you can play from T-X to T+5 and still get the same result.

Exercise - Oplog and Replication

Log into your cluster using mongosh

Use rs.status() to see details of the Replica Set

Look in the members array for the hostname of a secondary



```
$ curl ifconfig.me
18.193.118.157

$ mongosh "mongodb+srv://cluster0.xxx.mongodb.net" --username
mongoadmin
password: <passwordone>

Atlas xx-xx-0 [primary] test> rs.status()
{
      set: 'atlas-h4248u-shard-0',
      date: ISODate("2023-02-13T06:43:21.832Z"),
      myState: 1,
      term: Long("234"),
      syncSourceHost: '',
      syncSourceId: -1,
      heartbeatIntervalMillis: Long("2000"),
      majorityVoteCount: 2,
      writeMajorityCount: 2,

      ...

}
```

Make sure you have an Atlas M0 cluster up and running in the same region as your Strigo host otherwise you are looking at global network latency.
This must be on Amazon AWS in one of the following regions (Singapore, Oregon, Virginia, Frankfurt) - Your instructor will tell you which is correct.

Make sure you have a user created and the firewall (Network Access) open to your host.
Ensure that you have the sample dataset loaded.

When using Atlas you are always using at least one Replica set, if not multiple, in a sharded cluster.
Even in the free version you have high availability with seamless failover.
You can peek at the configuration for this (but not change it in Atlas from the command-line)
Log into your Atlas Cluster using mongosh and view the configuration.

Using the srv address returns all the hostnames and details and connects you to the 'Set'
When you connect to the 'Set' your commands and responses come from whatever is the Primary (exceptions apply)

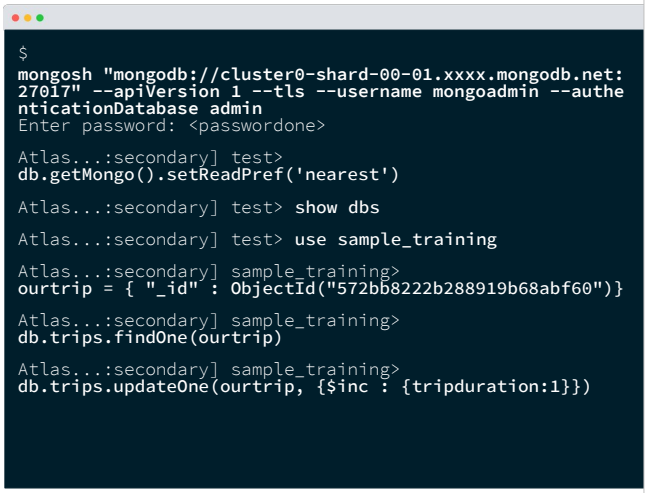
Log directly into a secondary

Log into a specific single host

Also need to specify TLS as that's specified by the SRV record only

List Database

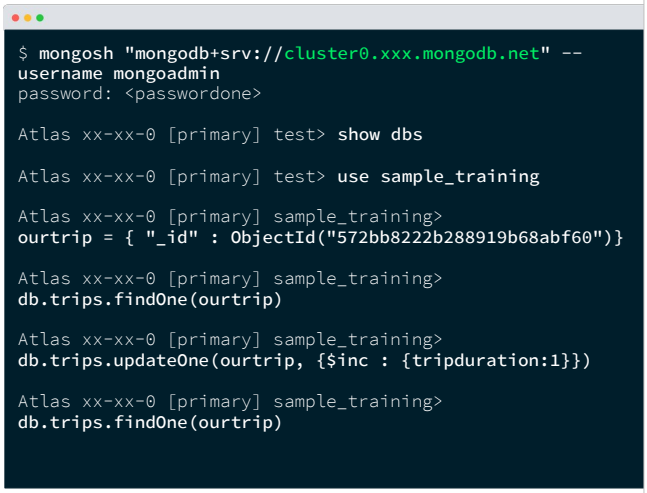
We cannot write to a secondary in mongosh



We can log into a single instance rather than the set
We normally don't do this except for maintenance tasks (like shutting it down)
On a Secondary we cannot do anything until we acknowledge it is a secondary in the original shell - in monogsh you don't need to. However, we cannot write to a secondary ever.
In the legacy mongo shell, a method called `rs.secondaryOk()` is available that provides us the acknowledgement so that data can be read from the secondary.

Make the Change on the Primary

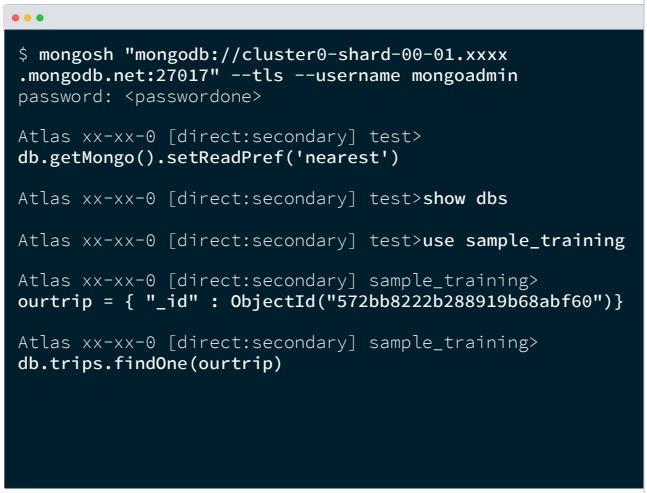
Go Back to the terminal on the Primary



We could not make a change on the secondary- but we can make a change on the primary and see it happen on the secondary.

Look again on the Secondary

Go Back to the terminal on the Secondary and re-run findOne()



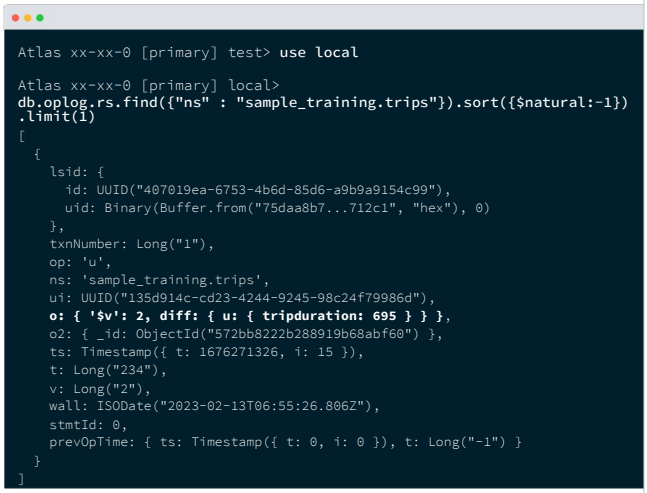
Now when we look on the secondary that change has propagated via the Oplog.

Look at the Oplog on the Primary

Go Back to the terminal on the Primary

Search the oplog - sort in reverse for the last entry

The secondary has an entry like this too



We can see our change in the oplog on the Primary - but it's now an idempotent change.
Even if our Query had not been by _id - the oplog entry would refer to a each individual change by the _id of the record.
The oplog is never indexed and is large so searching and be slow and disruptive - but if we scan in reverse {$natural:-1} and stop after one record it's quick.
This is an op of type "u" - update, it has an operation "o" and a target object "o2" and both an internal timestamp and wallclock time.
"uid" is the primary key for oplog records - they do not have _id fields. These are the only MongoDB documents that don't have an _id field.
You cannot edit the oplog.

Oplog Window

Oplogs are capped collections (fixed-size in bytes)
Guarantee the preservation of insertion order
Support high-throughput operations
Once a collection fills its allocated space:

- Makes room for new documents by deleting the oldest documents in the collection
- Like circular buffers

Sizing the Oplog

The Oplog should be sized to account for latency among members

Default Oplog size is usually sufficient - Make sure that your oplog is large enough:

- Oplog window is large enough to support replication
- Large enough history for any diagnostics you might wish to run

Initial Sync

- New / Replacement Replica Set members need a full copy of the data
- As do any that have been down too long where the oplog has rolled over
- Can take a long time and be relatively fragile on large systems

Recent changes (4.2 / 4.4)

- Can auto restart on non-transient error
- Resumable on transient error
- Copy Oplog at the same time
- Overall more resilient
- Can use a secondary as source

The idea of Majority

In distributed systems, the idea of a majority or quorum is important:
Group that together have more than half the total members (the majority)

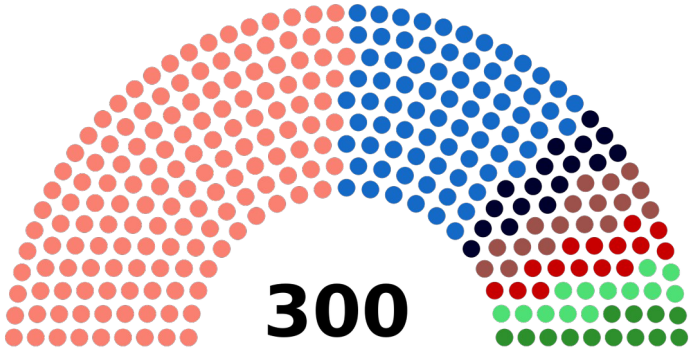
- There cannot be another group that has more than half
- The majority can make a decision knowing no-one else will
- A member is never sure if it's in the majority - it can just vote

Majority is an essential concept in distributed systems

A simple idea - you can only ever have one majority so if a majority of nodes agree on something, then even without knowing what you other nodes want - you do know that they do not have an overall majority.
But you must have an odd number to start with otherwise you could get an even split and no decision made.

Elections

Find the most suitable candidate as a leader



Unlike a political election - you need a majority of all eligible voters to be in charge - not just a majority of those who voted.
Each node will vote for or against the proposed candidate. Because you need a simple majority of all voters 'nay' votes are equivalent to not voting (abstaining), except that 'nay' votes can hasten the completion of an election since the node running the election can stop waiting for votes to come in (if it is destined to lose). A node also stops waiting for votes immediately if it gets sufficient 'yea' votes.

Elections

How to choose a Primary (Leader)

- Agreeing on a primary is surprisingly hard
- There are many academic papers on how to do this
- MongoDB uses the RAFT protocol
- Only one Primary is chosen at a time - The chosen one (new Primary):
  - Must be able to communicate with a majority of nodes
  - Should be the most up to date with the latest information
  - May prefer some over the others due to geography

- An election should result in a new primary being chosen
- There are a few factors as to how a primary is chosen

Elections - The Simple Version

A secondary determines:

- Has not heard from the Primary recently - Primary might be up, but not getting through
- Can contact a majority of secondaries, including itself
- Can vote and is not specifically ineligible

Secondary then proposes an election - with itself as Primary

- States its latest transaction time
- States the election term - that goes up by one every election
- Votes for itself by default

Heartbeats are sent to all members every 2 seconds. The default heartbeat timeout (or election timeout) is 10 seconds. If a heartbeat ping to a member (including primary) times out without response, that member will be marked as inaccessible.

Elections - The Simple Version

Any server that can vote, only votes for the candidate in an election if:

- The candidate is at the same or a higher transaction time
- Has not already voted in that election term

If it's currently the Primary, stands down

Once a candidate receives a majority of votes:

- Converts itself from a Secondary to a Primary
- Goes through an onboarding
- Starts to accept writes

There is actually a dry-run election first to avoid the primary standing down too soon!

Onboarding a New Primary

Once a node has been elected as a primary, it does some onboarding:

- Checks if any secondary, it can see, has a higher operation time - If so, copies over transactions from it
- Checks if any secondary has a higher priority than itself - If so, arranges a handover and passing of up to date transactions

All this typically happens in a very few seconds

Guarantees that any operation that a majority of nodes wrote and acknowledged is not lost

For the onboarding process, you can tune the parameters, timeouts, etc.

Primary stepping down

A Primary will become a secondary when:

- Sees an election happening that's later than the one it was elected in
- You explicitly tell it to step down - it does a good handover
- Can no longer see a majority of the secondaries

In Summary

There are multiple servers with the same data

Data is durable when it's on a majority of voting servers

Data takes time to propagate (normally milliseconds)

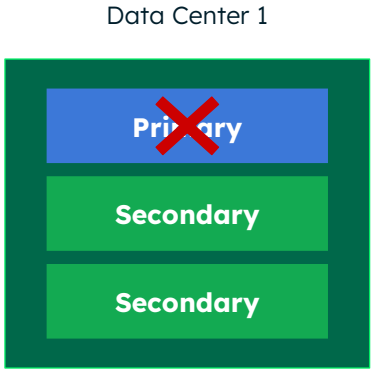
There can be non-voting servers for extra tasks, not HA

Exercise - Primary election

Look what happens when networks break

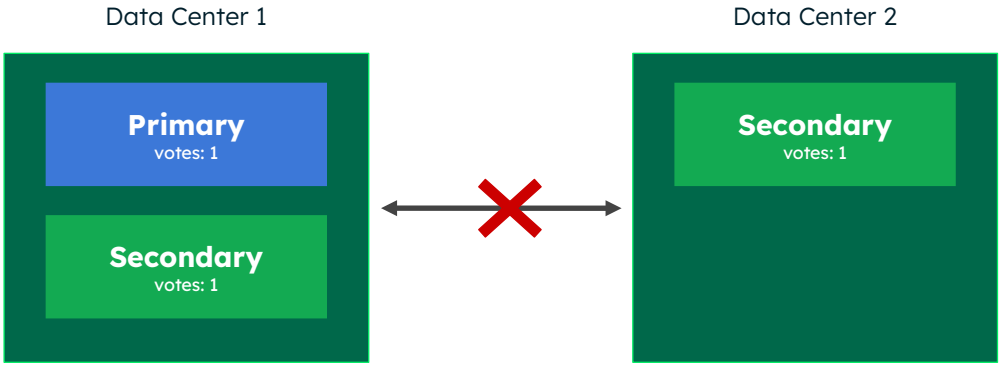
For each of the **next slides**, see which host (if any) becomes the new primary

3 Nodes, 1 data center



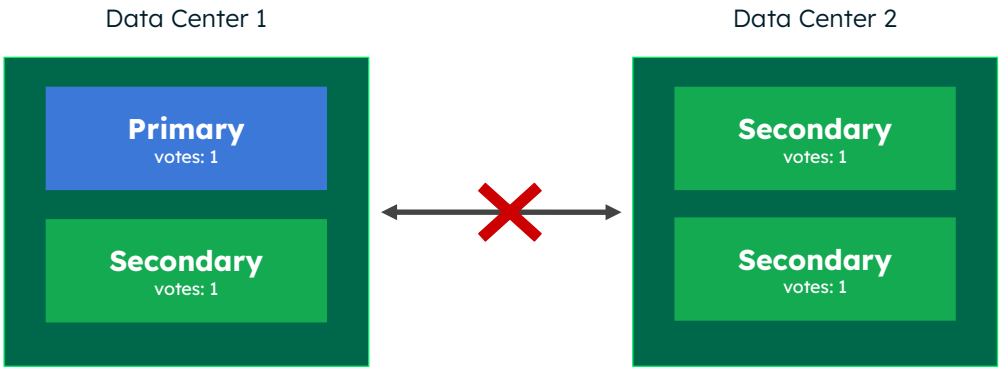
In this scenario - If the Primary goes down which secondary, if any, will become the new primary ?

3 Nodes, 2 data centers



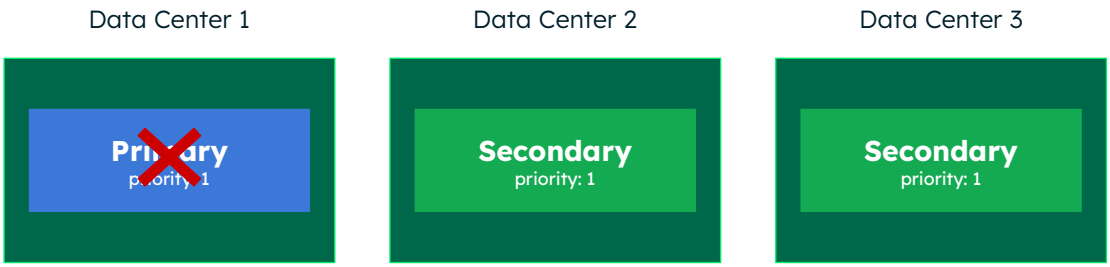
In this scenario - If the Network goes down which secondary, if any, will become the new primary ?

4 Nodes, 2 data centers



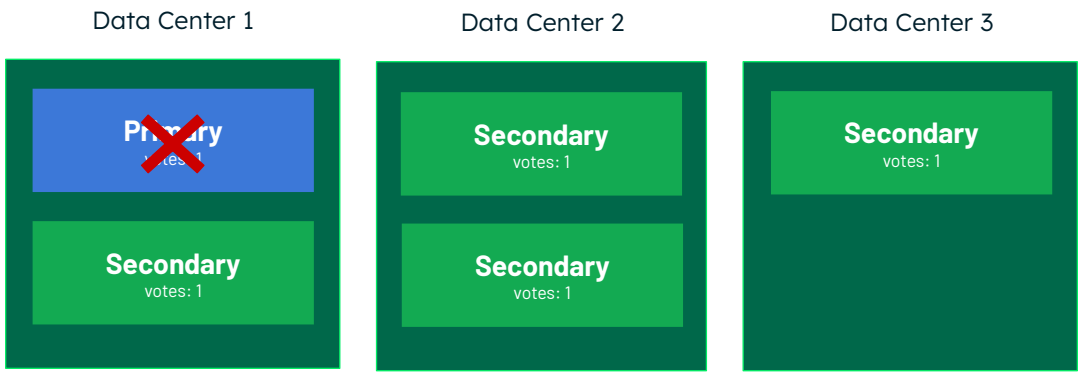
In this scenario - If the Network goes down which secondary, if any, will become the new primary ?

3 Nodes, 3 data centers



What will happen in this scenario? How long does recovery back to the full three nodes take?

5 Nodes, 3 data centers



What about this one? How long does recovery back to a full 5 nodes take?
Restoring on one data center is faster (and cheaper) than over a WAN.

Developer responsibilities

Replication doesn't seem like a developer issue

- Developers need to understand the implications
- The software must support it correctly
- Make decisions with business owners about speed versus durability
- In any multi-server system, safety, speed, and correctness must be considered

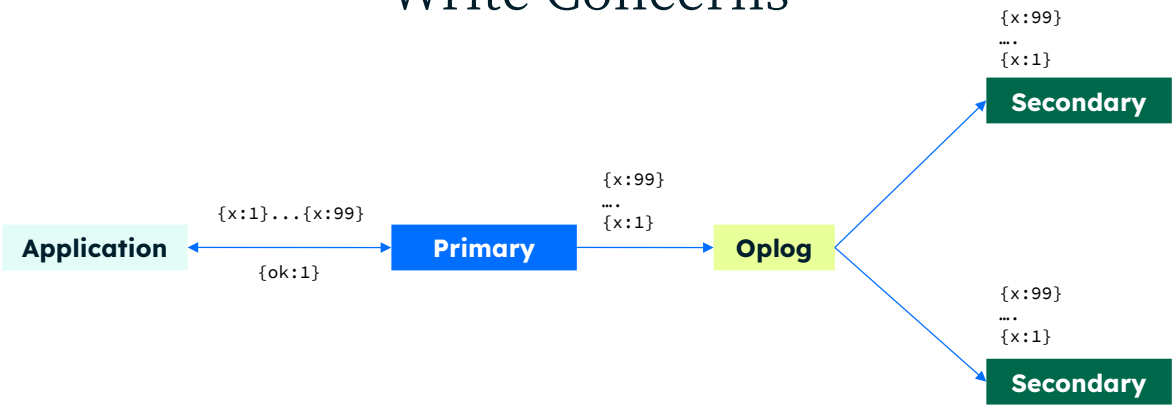
Developers should be aware of replication and how this affects applications.
MongoDB gives you strong tools to control and fine-tune tradeoffs among safety, speed, and correctness

Write Concerns - Questions

When writing to the database:

- What does Durable mean to you?
- What does 'OK - committed' from the database mean?
- What sort of data would not matter if the latest was lost in a crash?
- Who decides what data must never be lost?
- Is there a difference between knowingly lost and unknowingly lost?

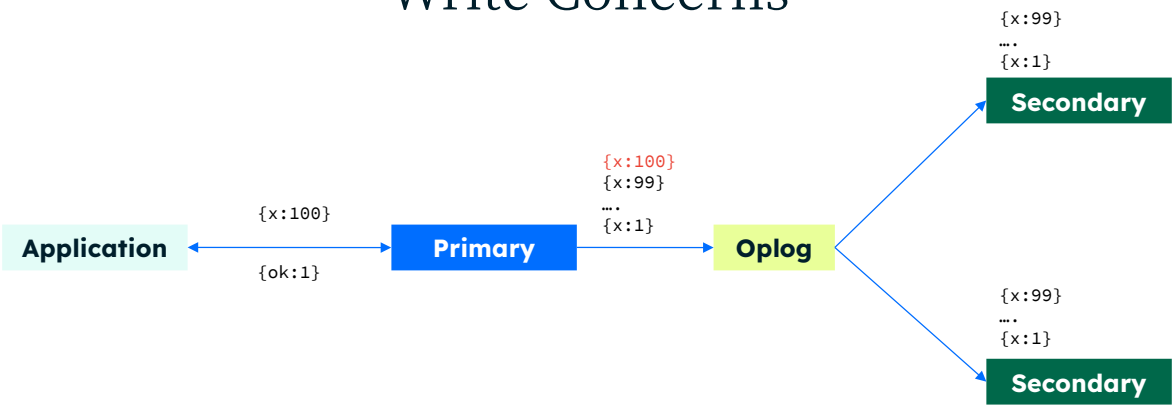
Write Concerns



Writing some data

We add 99 records to the database, with values X=1 to X=99 and they go through the oplog and get replicated everywhere.

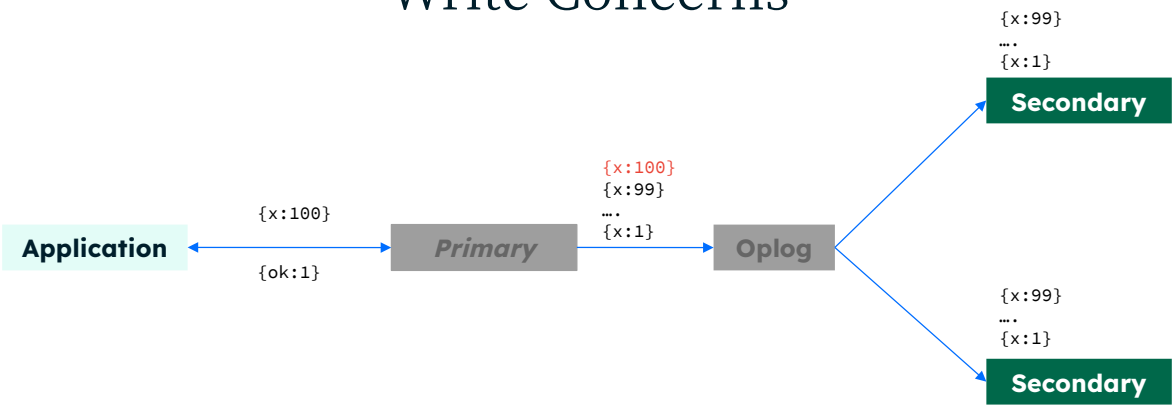
Write Concerns



Writing some more data

We send a new record (or a change to an existing one - this could just all be updates of X in a single record) this time X=100
We write to the primary and get an OK back as the primary has written it (for some definition of written)

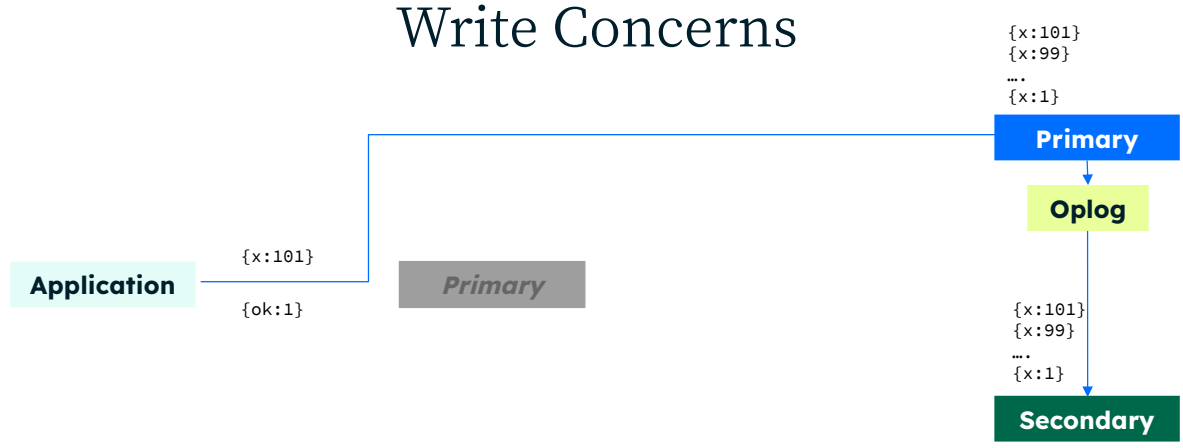
Write Concerns



Primary goes down

Now the primary crashes, after telling us but before either secondary took a copy. so X:100 is not on either secondary, and the primary is down.

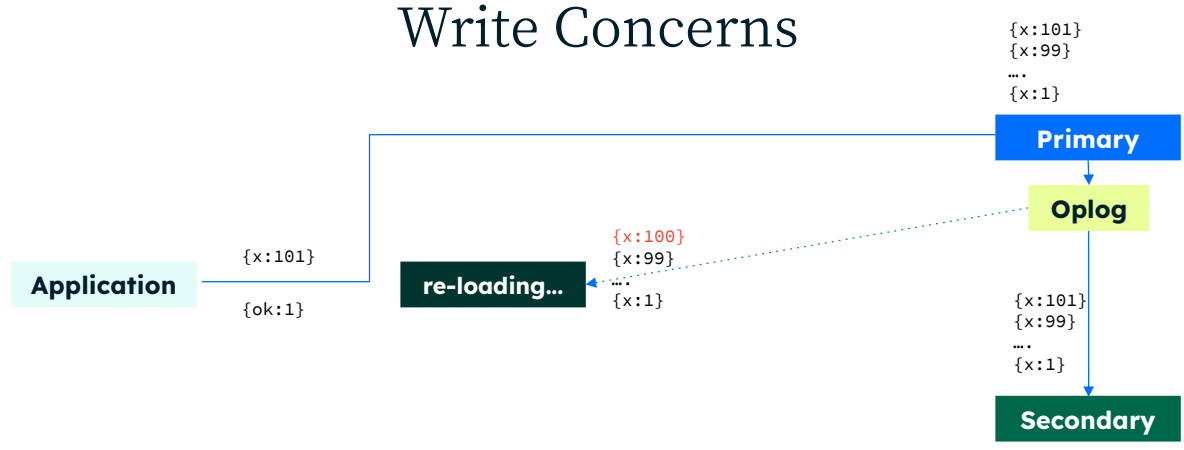
Write Concerns



New primary elected - Keep writing data

A new primary is elected from the remaining secondaries and now we write X=101 - both secondaries see that change and it's replicated.

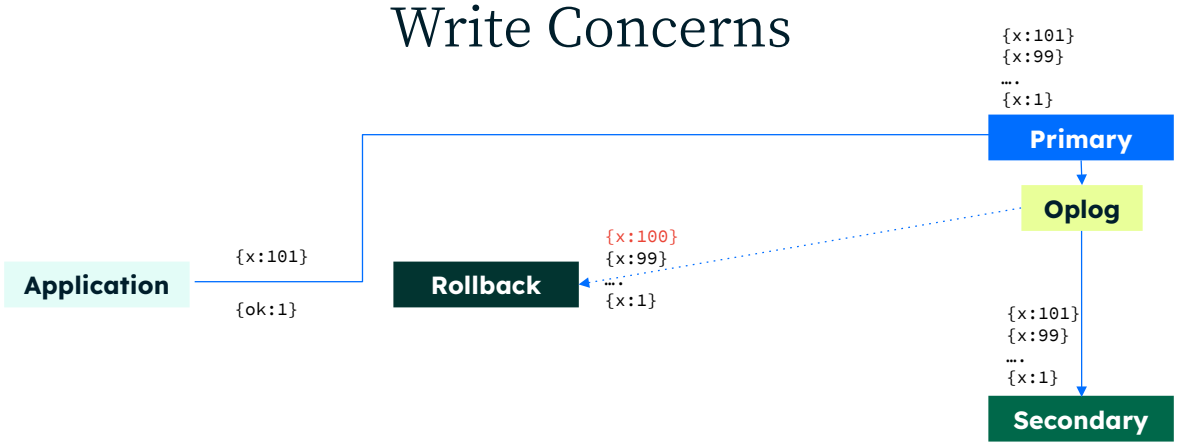
Write Concerns



Old primary coming back

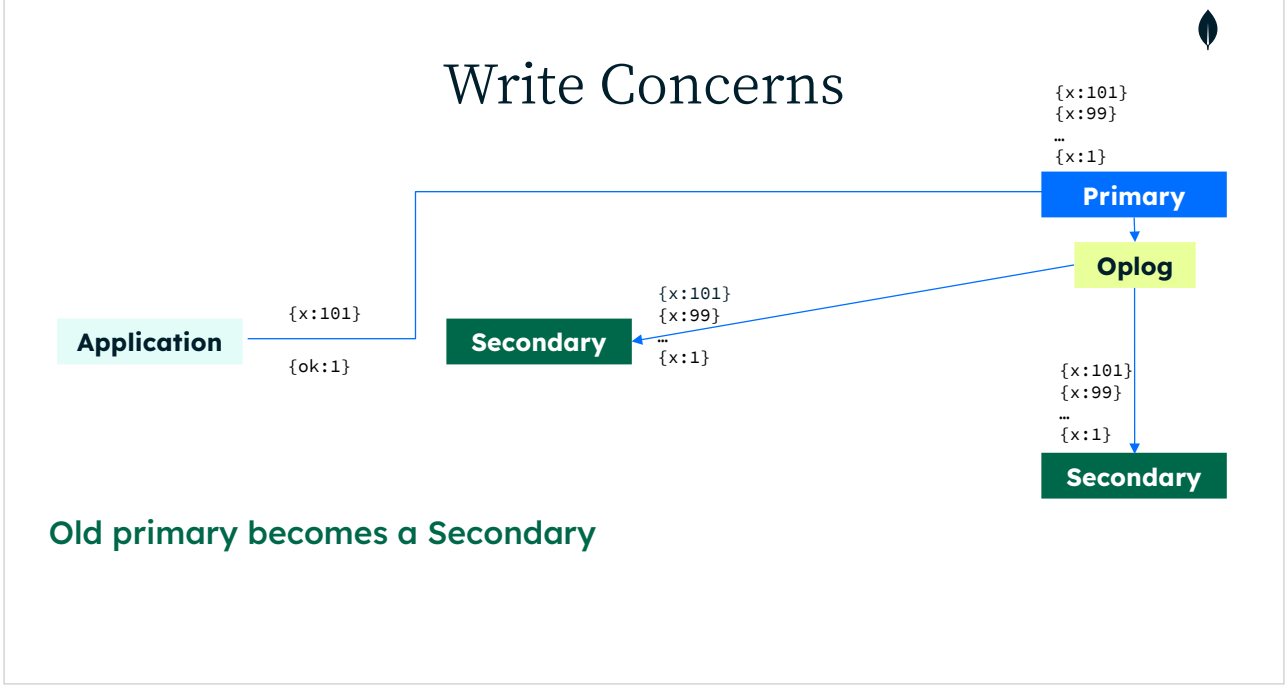
Our old primary comes back online and immediately sees that a new primary was elected after it so does not assume it's a primary.

Write Concerns



Resume Replication
But has to fix his state to resume

It sees the last common point in it's oplog, compared with the primary was the 99, that's the last time they are the same.
So it rolls back and removes all the changes after 99 - either deleting records or grabbing the latest version from the primary for changes
Then it plays forwards all the oplog changes in the new Primary after that last common point.



Finally it takes its place as a secondary.

Case for Majority writes

In the previous scenario:

- Data is written to the primary
- Primary acknowledges write to application
- Primary dies before secondary reads data
- Secondaries have an election
- All trace of the acknowledged write (x=100) was silently lost!

Majority Commit Point

Primary knows what timestamp each secondary is asking for, therefore it knows:

- They have everything before that durable
- Up to what timestamp exists on a majority of nodes
- Up to what point data is 100% safe

Until a change is 100% safe, the Primary will keep it in memory

In a system with automated failover, “Safe” means "If it fails over, this won't be silently lost"

If it runs out of memory, it spills these cached copies/cache snapshot out of the disk in a file called the LAS file.
Can you guess how we might accept writes, but then they can never get to a majority of nodes?
We will talk about that later.

Write Concerns

In MongoDB, use write concerns to define what 'OK, committed' means:

- Received by the primary over the network but not examined (`w:0`)
- Received and written by the primary - durable on primaries disk (`w:1, j:1`)
- Received and written by a majority (`w:"majority"`)

`w` is the number of servers, `j` is whether to wait for the next disk flush (default majority)

Write concerns can be specified in the application on:

- Any write operation
- A connection
- An object used to write

MongoDB waits until it achieves the request level or times out. If it times out, it may still have done some part of it. In the event of a timeout, you may need to confirm the state.

Why not just always write to a majority?

There are other write concerns too, but these are really the ones that matter.
TBH - always writing to a majority is a good plan a lot of the time.

Exercise - Write Concerns

The Manual page for insertOne shows we can specify the write concern using

db.col.insertOne(<document>,{writeConcern:{ w:<value>, j:<boolean>, wtimeout:<number>}})

	Durability Guarantee			
Batch size	w:0	w:1	w:1, j:true	w:"majority"
1				
100				

Run this code in the shell to measure the time taken to insert 10,000 small documents using different write concerns w and values of j

Use a for loop and Complete the table

How would the relative location of the client and servers impact this?

```
var wc = { w: 0 }
totalrecs = 10000
batchsize = 1
nbatches = totalrecs / batchsize
var start = new Date()
for(x=0;x<nbatches;x++) {
recs = []
for(y=0;y<batchsize;y++) {recs.push({a:1,b:"hello"})}
db.test.insertMany(recs,{writeConcern: wc})
}
var end=new Date()
print(`${end-start} milliseconds`)
```

Read concerns

When reading, choose how reads are impacted by what's durable

Read Local	What's the latest on the Replica set member we read from
Read Majority	**What's the latest that is 100% durable** Enabled by Majority commit point
Read Snapshot	**Read what was there when our query starts** This hides any changes whilst the query is going on But we need to keep data around whilst we do it
Read Linearizable	Wait until a majority catch up with my query time

On the Primary: “local” returns the data on the primary.

On a Secondary: "local" returns whatever that secondary has, "majority" still returns what has been replicated to a majority of hosts

Local is normally appropriate however majority avoids any chance of dirty reads at minimal extra cost.
There are few circumstances where Snapshot or Linearizable are required to be used.
Read Snapshot is only applicable when you are in a transaction before MongoDB 5.0. After 5.0 it can be used at any time.
There is one additional concern 'Available' - in a Sharded cluster it allows more tolerance of partitions at the risk of returning the same document twice during a chunk migration. It should be avoided in most circumstances.

Read Preferences

When do you think you would use each of the following?

- Read from Primary Only
- Read from Primary unless no Primary exists (primaryPreferred)
- Read from any Secondary Only
- Read from Secondary unless no secondary exists
- Read from nearest Geographically
- Read from a specific set of servers

Which read preference would you use and why?

Read preference specifies where the reads should be sent.
This is an interactive slide - The instructor will describe each - you have to say when you use it.

Arbiter

An Arbiter

- Does not have a copy of the data set and cannot become a primary
- Participates in elections for primary and acts as a tie-breaker
- Is strongly advised against in production systems

A system with Arbiters can be Highly Available OR Guarantee Durability - But not both

They introduce many, many edge cases.
Arbiter cannot answer any query.
With an arbiter you cannot meet majority if a writable node is down (i.e., you have a majority but one writable node out of 3)
Arbiters, therefore, raise questions about reliability guarantees.
We do not recommend **ever** using Arbiters in production - they are a false economy.

Quiz Time!

#1. Which of the following decides where to send a write for it to be considered durable?

A	The Database	B	The Driver	C	The Developer
D	The Operations Team	E	The Business Management		

Answer in the next slide.

#1. Which of the following decides where to send a write for it to be considered durable?

A The Database

B The Driver

C The Developer

D The Operations Team

E The Business Management

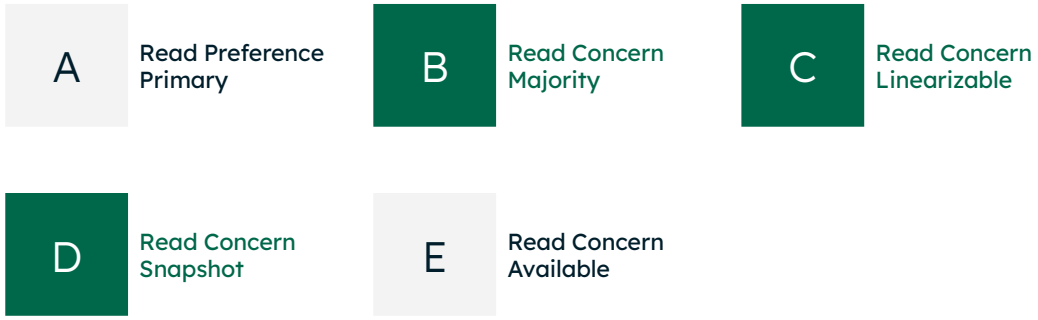
Answer: B,C,D, E

#2. What features can you use to read just the data that is fully durable?

A	Read Preference Primary	B	Read Concern Majority	C	Read Concern Linearizable
D	Read Concern Snapshot	E	Read Concern Available		

Answer in the next slide.

#2. What features can you use to read just the data that is fully durable?



Answer: B,D

#3. What should be done to prevent data loss before a new primary is elected?

A	Data must be written to all nodes before the election
B	Data must be written to a majority of nodes before the election
C	Data must be written to a secondary node before the election.
D	Data in the primary disk must have flushed before the election.
E	Data must have been backed up before the election.

Answer in the next slide.

#3. What should be done to prevent data loss before a new primary is elected?

A	Data must be written to all nodes before the election
B	Data must be written to a majority of nodes before the election
C	Data must be written to a secondary node before the election.
D	Data in the primary disk must have flushed before the election.
E	Data must have been backed up before the election.

Answer: B

Recap

Replication creates multiple identical copies of data - Typically used for High Availability

Writes to a Primary are replicated to Secondaries

The primary is elected by the cluster as needed

This replication happens via the oplog

A write concern of “majority” is the one that ensures data durability

Arbiters are not recommended

Exercise Answers

Answer - Exercise: Write Concerns

	w:0	w:1	w:1,j:1	w:"majority"
InsertOne	2184	8884	15279	50512
InsertMany	256	316	629	1195