Release: 20250117

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)

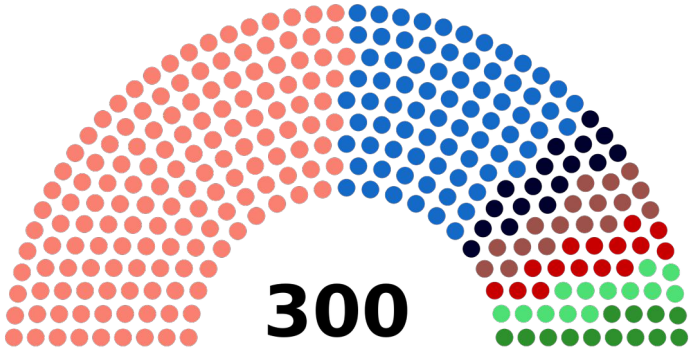
- There cannot be another group that has more than half
- The majority can make a decision knowing no-one else will
- A member is never sure if it's in the majority - it can just vote

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 visible secondaries have a higher operation time; copies newer operations if they exist
- 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 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 voting replica set members

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)

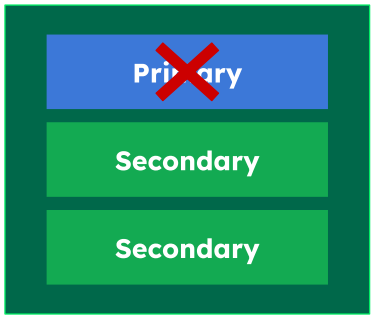
There can be non-voting servers for extra tasks, not HA

Exercise - Primary election

For each of the **next slides**, identify which host (if any) will become the new primary.

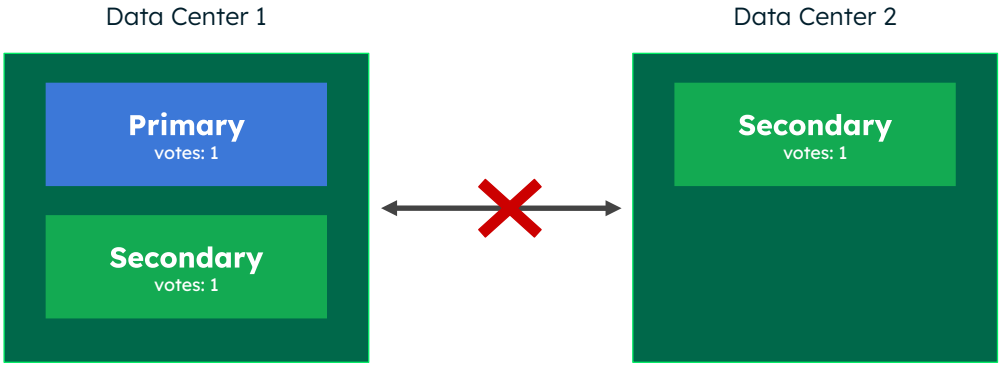
3 Nodes, 1 data center

Data Center 1



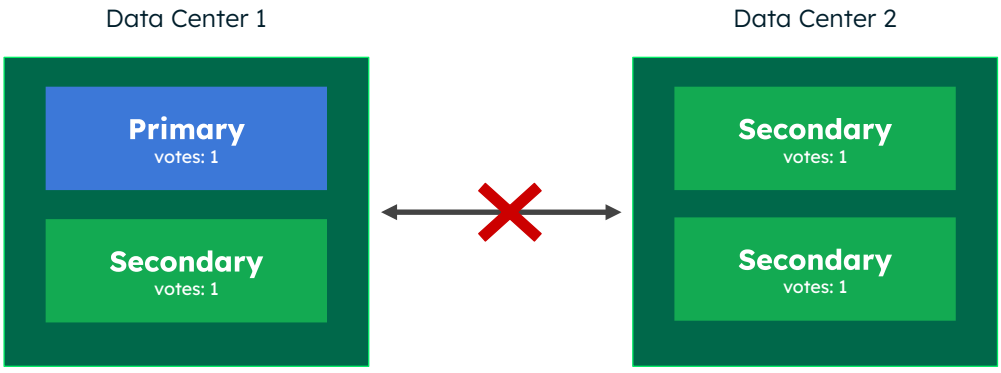
In this scenario - If the Primary goes down which secondary, if any, will become the new primary ?

3 Nodes, 2 data centers



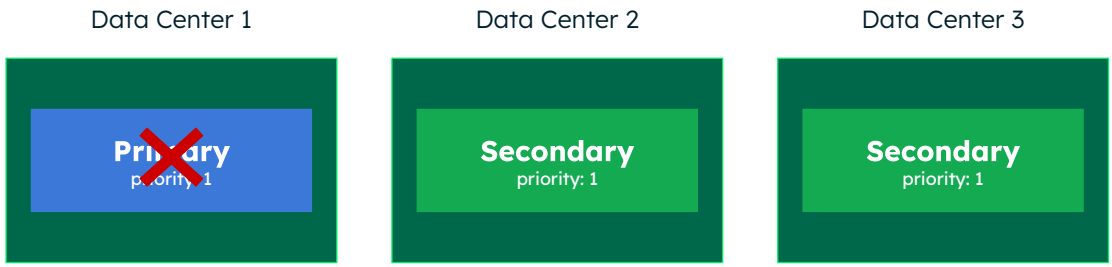
In this scenario - If the Network goes down which secondary, if any, will become the new primary ?

4 Nodes, 2 data centers



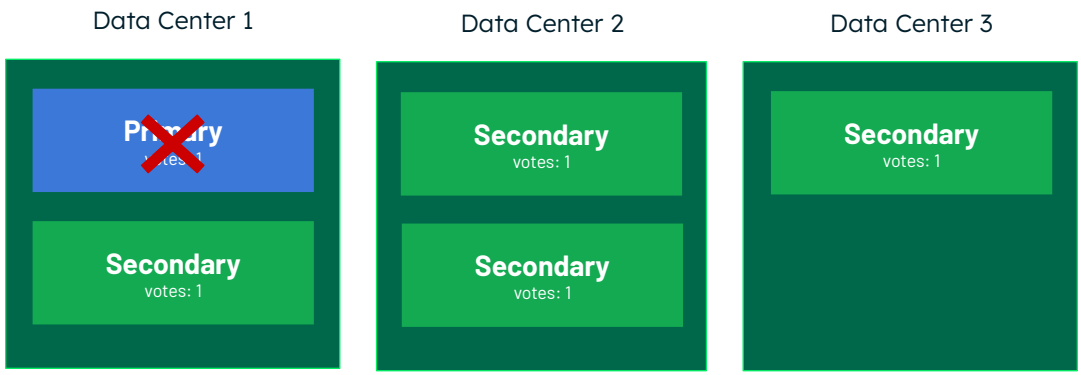
In this scenario - If the Network goes down which secondary, if any, will become the new primary ?

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

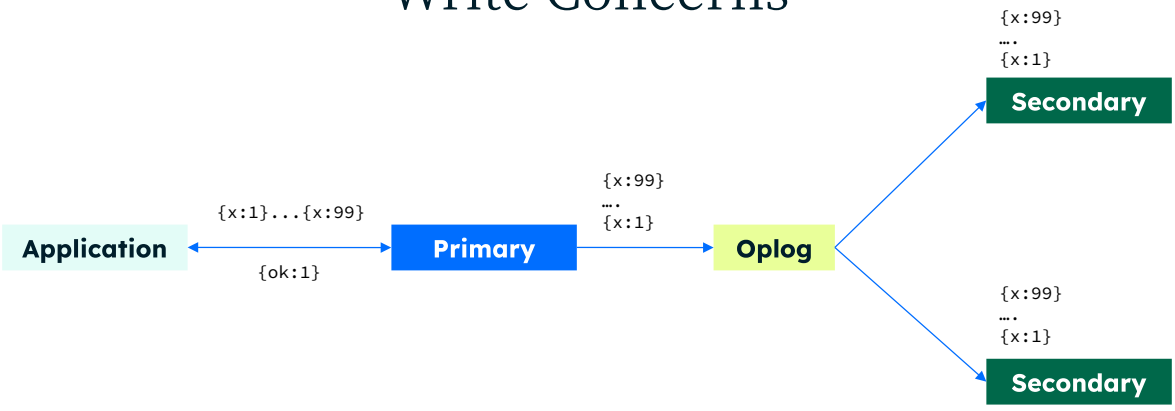
Developers should be aware of replication and how this affects applications.
MongoDB gives you strong tools to control and fine-tune tradeoffs among safety, speed, and correctness

Write Concerns - Questions

When writing to the database:

- What does **durable** mean to you?
- What does a write acknowledgement from the database mean to the application?
- Is it okay if writes are lost? If so, what types of writes?
- Who decides what data must never be lost?
- Is there a difference between knowingly lost and unknowingly lost?

Write Concerns



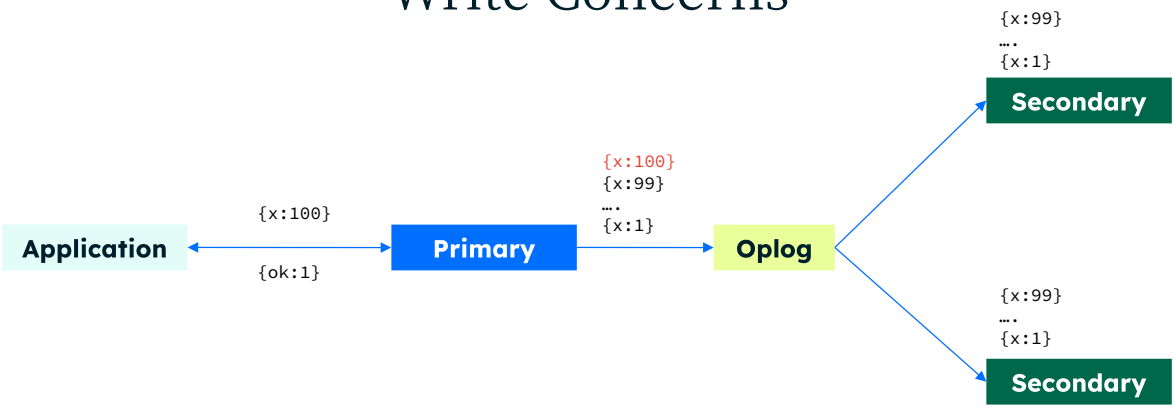
Writing some data.
In this example, we have { w : 1, j : true }

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

The w option to request acknowledgment that the write operation has propagated to a specified number of mongod instances or to mongod instances with specified tags.

The j option to request acknowledgment that the write operation has been written to the on-disk journal

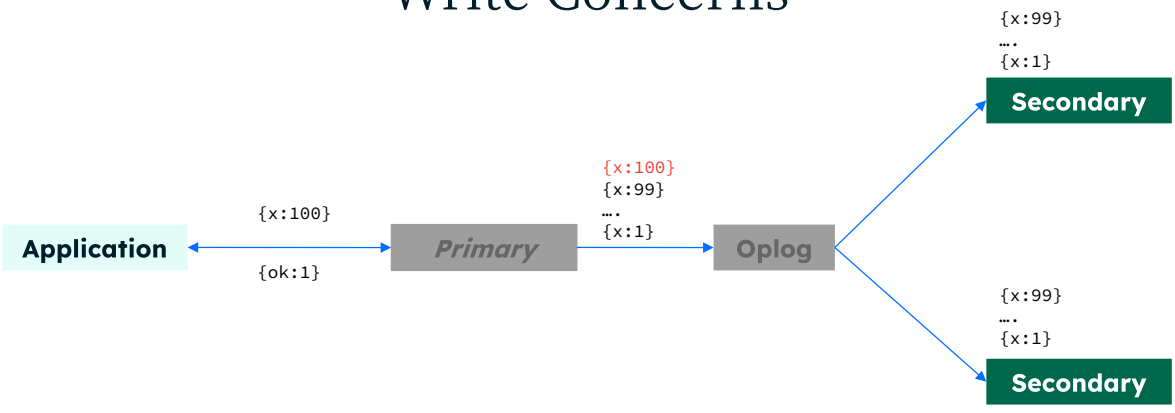
Write Concerns



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)

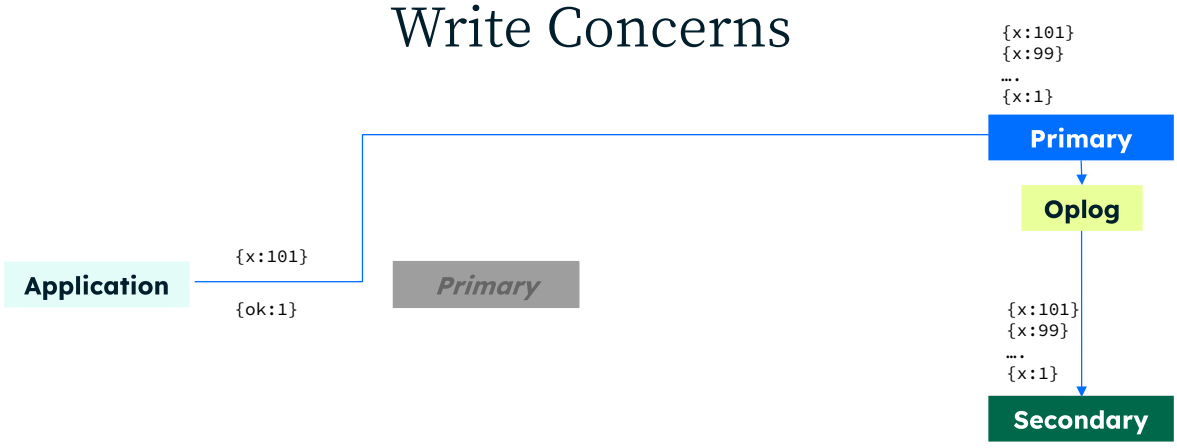
Write Concerns



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.

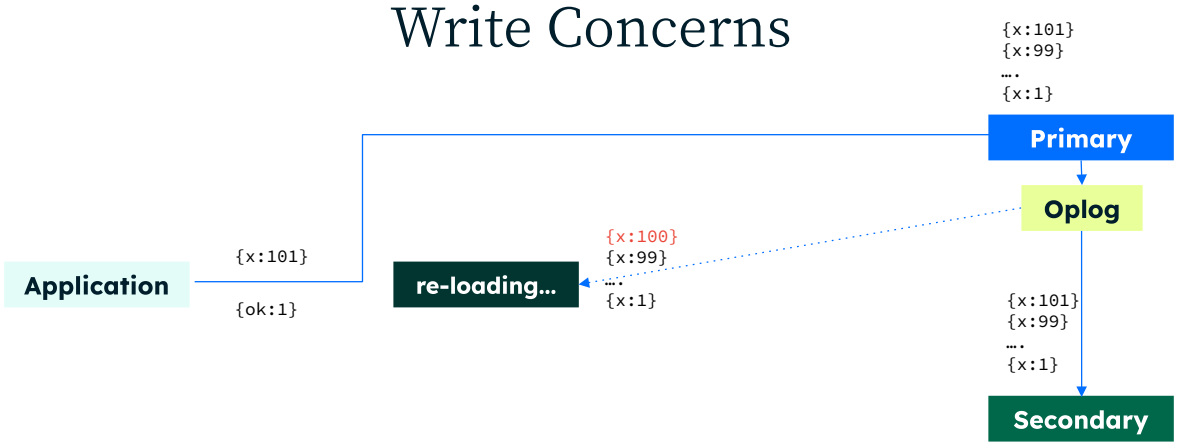
Write Concerns



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.

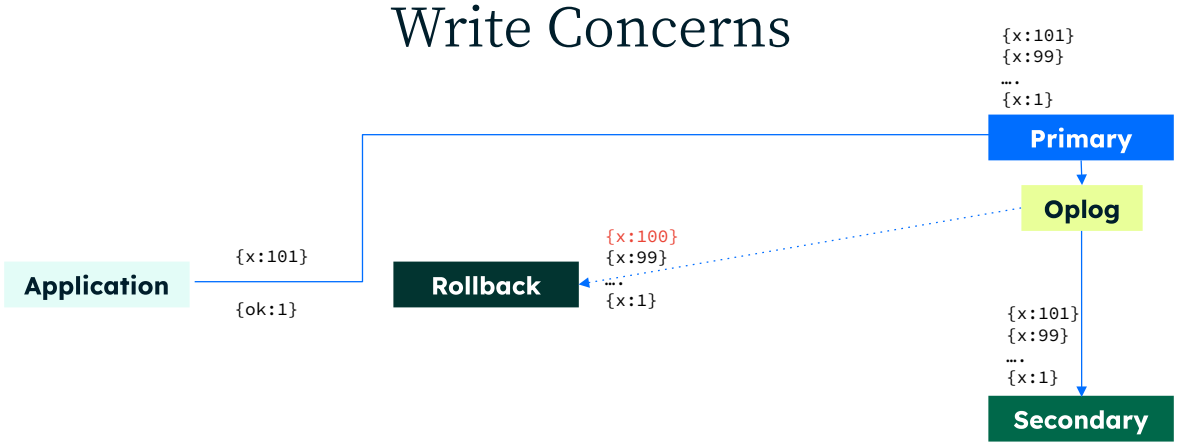
Write Concerns



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.

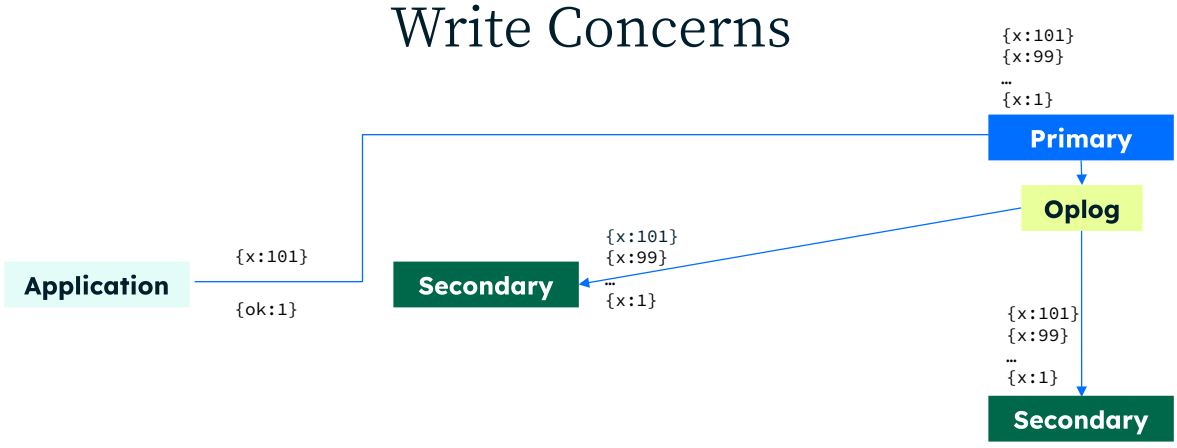
Write Concerns



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.

Write Concerns



Old primary becomes a Secondary

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:true`)
- 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

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

The Write Concern can be specified for individual write operations

Run the code in the notes to test the speed of various write concerns

What has more impact on the write speed: batch size or write concern?

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


```
var wc = { w: 0 }
totalrecs = 1000
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

- Acts as a tie-breaker in elections
- Stores no data
- Cannot become a Primary
- Is strongly advised against in production systems

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

In a 2-member replica set, a server requires 2 votes to win an election. An Arbiter added to such a replica set will have 1 vote and will help elect a primary if one of the data-bearing servers goes down.
However, an arbiter can't store data, so if one data-bearing node is down, writes with a write concern of w:majority will not succeed.
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

All of the mentioned options are involved in the decision of where to send a write via a write concern, outside the database itself in that the db will execute the command given it by the decision makers.

#2. What three 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 three 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

Fully durable data means if we were to take away the power from one server, we could find the same dataset in another. The read the data that is fully durable - the read concerns are 'majority, snapshot and linearizable'

#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.

To avoid data loss - it is advised the data is written with a write concern higher than 0 or 1 or even a specific number lower than the majority in the RS. In this notion, using majority will prevent data from being lost regardless how many nodes your RS has or which nodes in the RS are available.

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

```
var wc = { w: "majority" }
var start = new Date()
for(x=0;x<10000;x++) {db.test.insertOne({a:1,b:"hello"},{writeConcern: wc})}
var end=new Date()
print(`${end-start} milliseconds`)
```

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

```
var wc = { w: 0}
var start = new Date()
for(x=0;x<100;x++) {
recs = []
for(y=0;y<100;y++) {recs.push({a:1,b:"hello"})}
db.test.insertMany(recs,{writeConcern: wc})}
var end=new Date()
print(`${end-start} milliseconds`)
```