MDB300

Replication Part I

MongoDB Production Readiness

Release: 20250117

Topics we cover

Replication

Replica Set Components

Concept of Majority

How elections happen

Read concerns and preferences

Replication

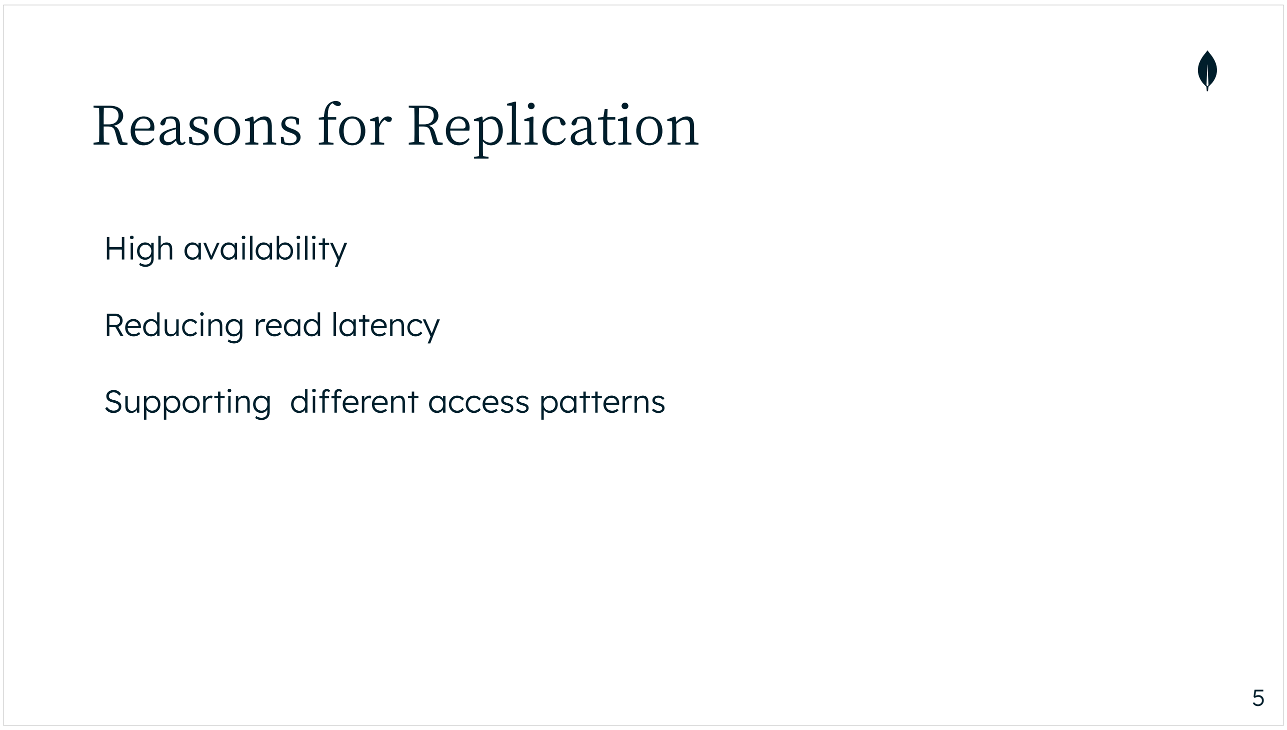
Multiple copies of each collection

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.



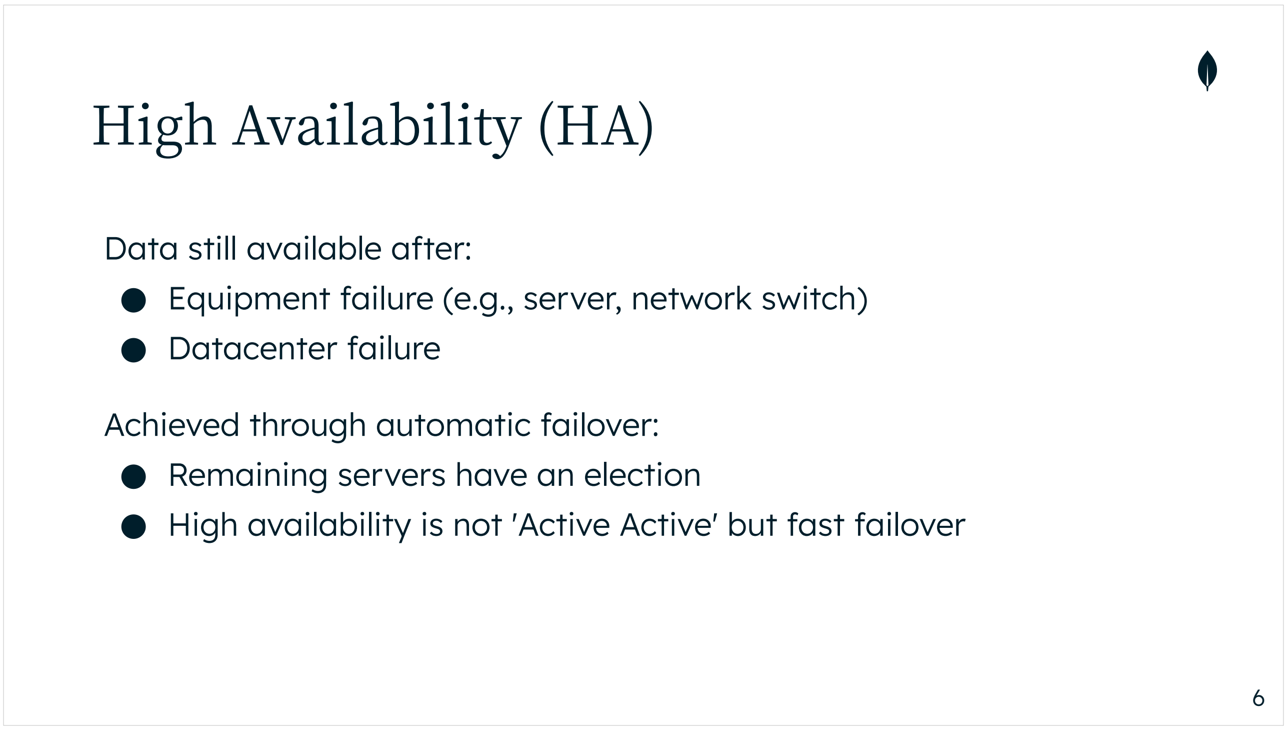
Reasons for Replication

High availability

Reducing read latency

Supporting different access patterns

These are explained in detail in the next few slides.



High Availability (HA)

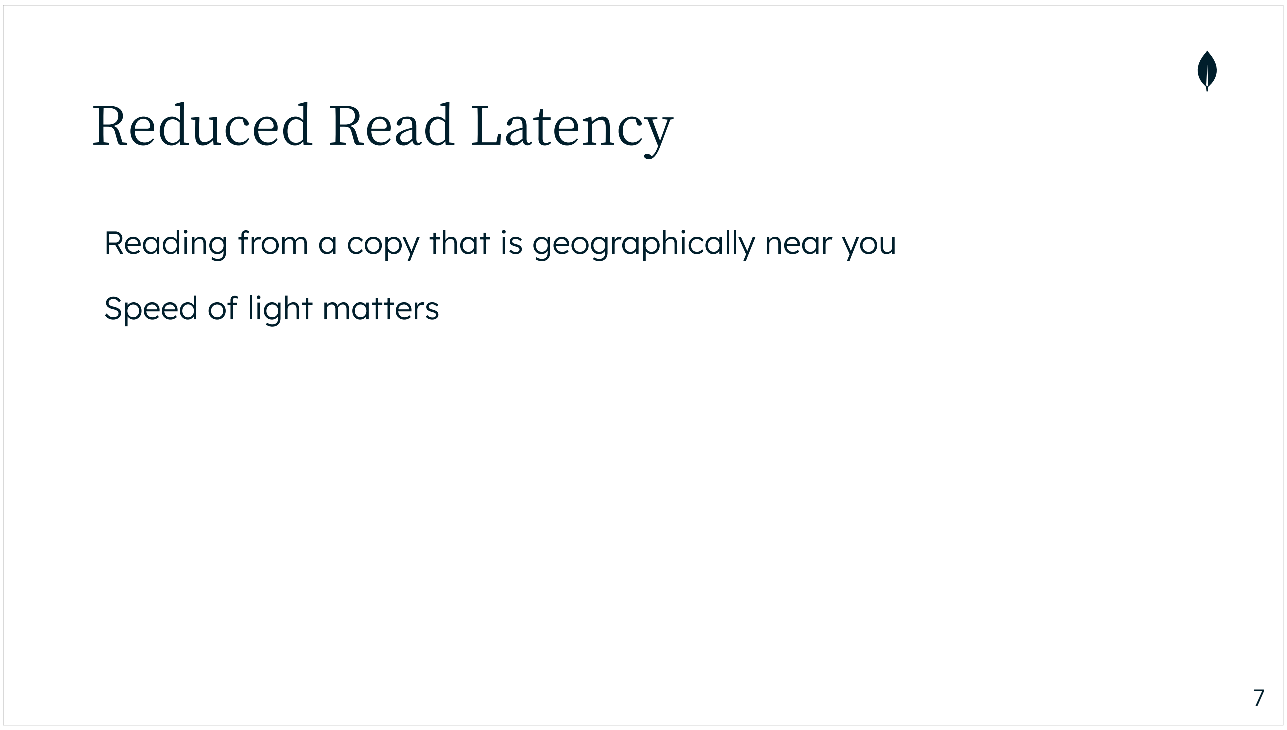
Data still available after:

- 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.

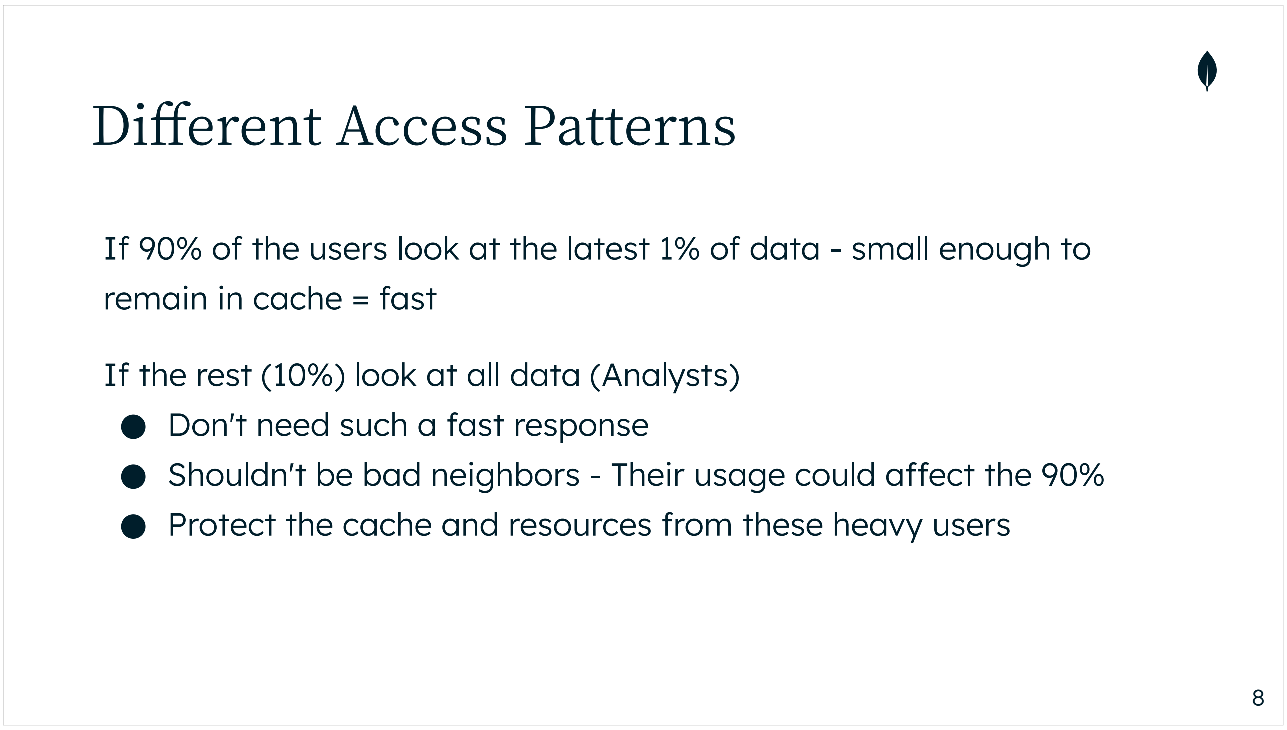


Reduced Read Latency

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 Primary, so this model only allows geographically close reads.



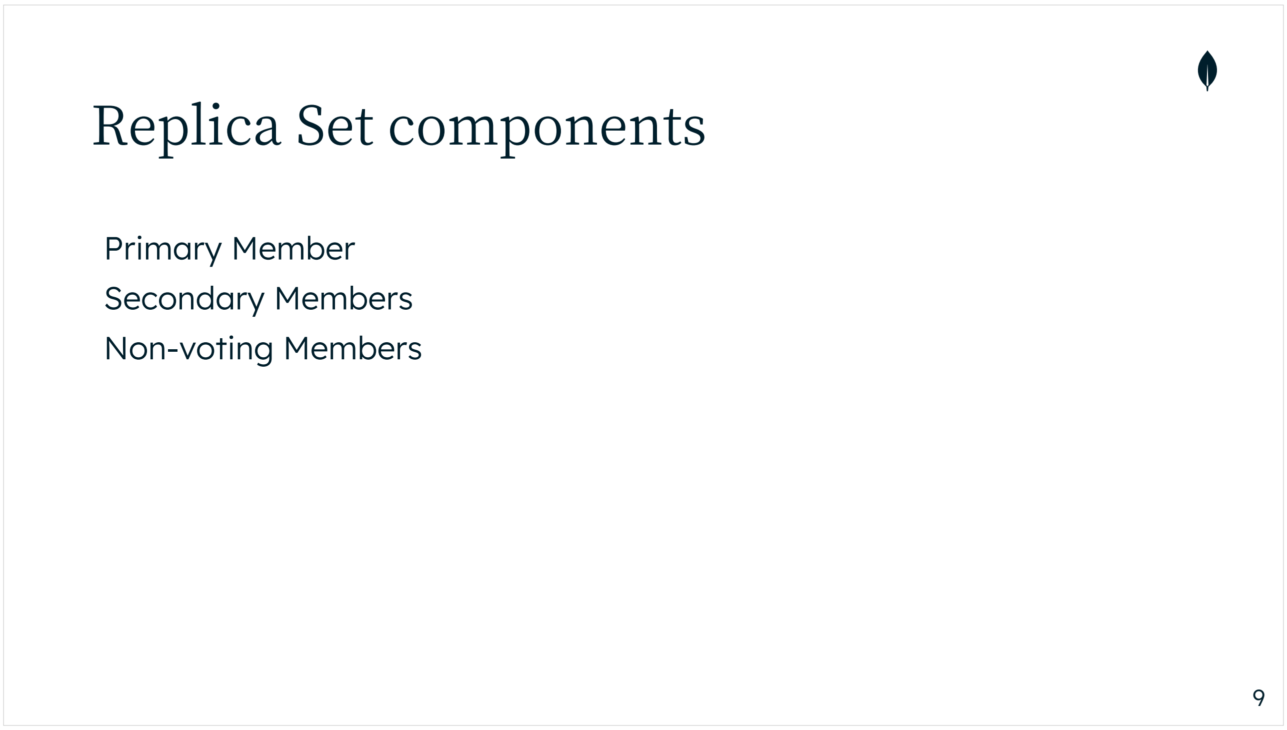
Different Access Patterns

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

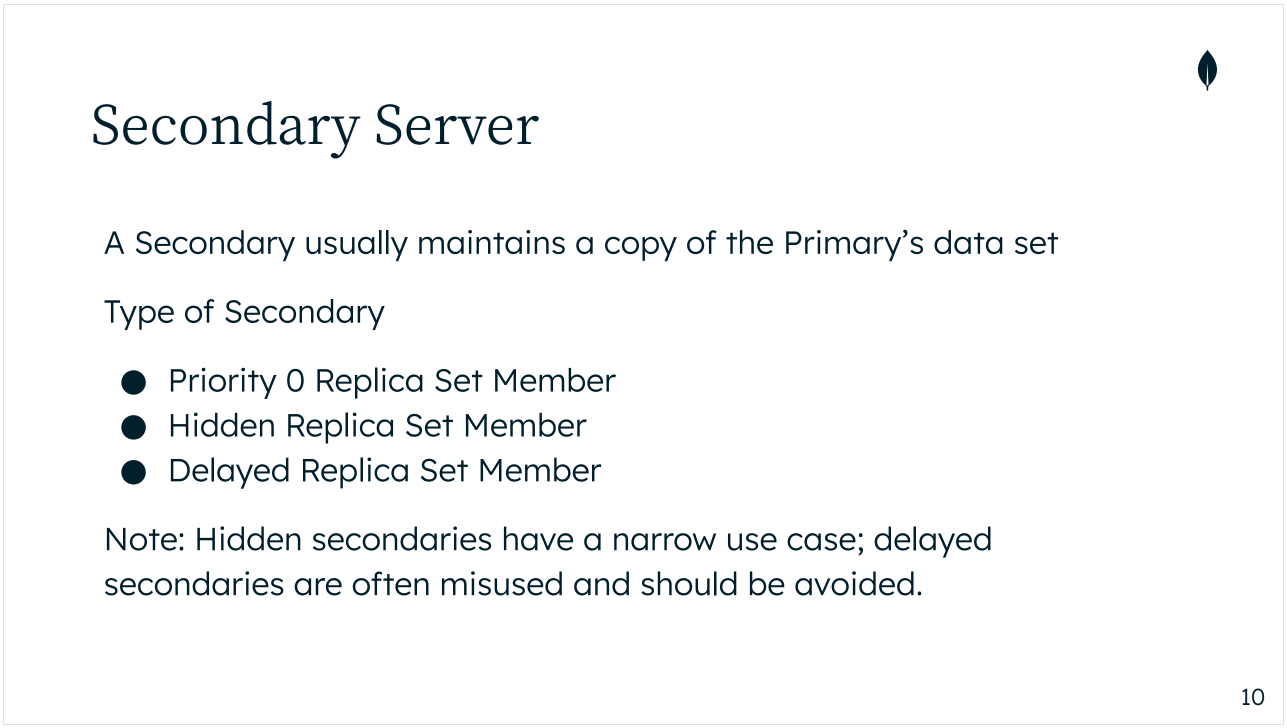
Primary Member
Secondary Members
Non-voting Members

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 usually 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: Hidden secondaries have a narrow use case; delayed secondaries are often misused and should be avoided.

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.

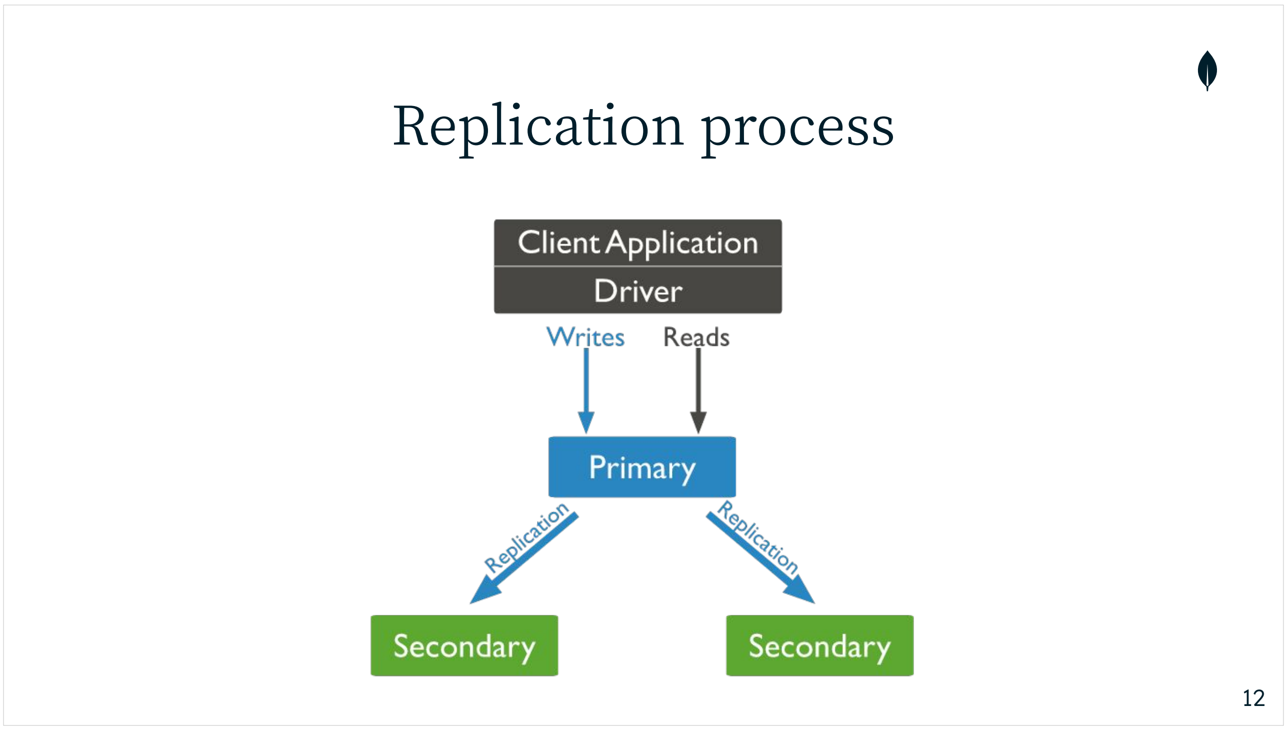
Chained replication means a secondary may maintain a copy of another secondary.

Drivers and Replica sets

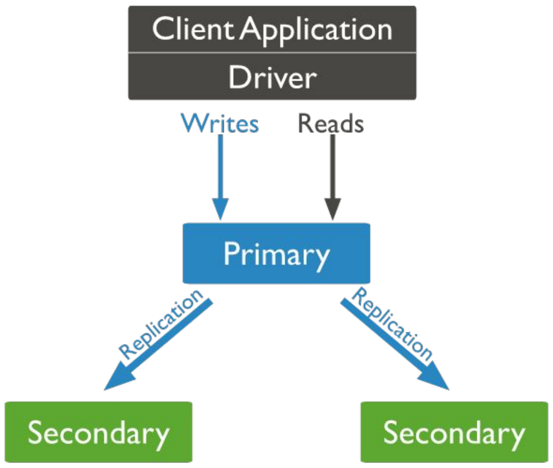
When coding with MongoDB, Drivers keep track of the topology

- 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.



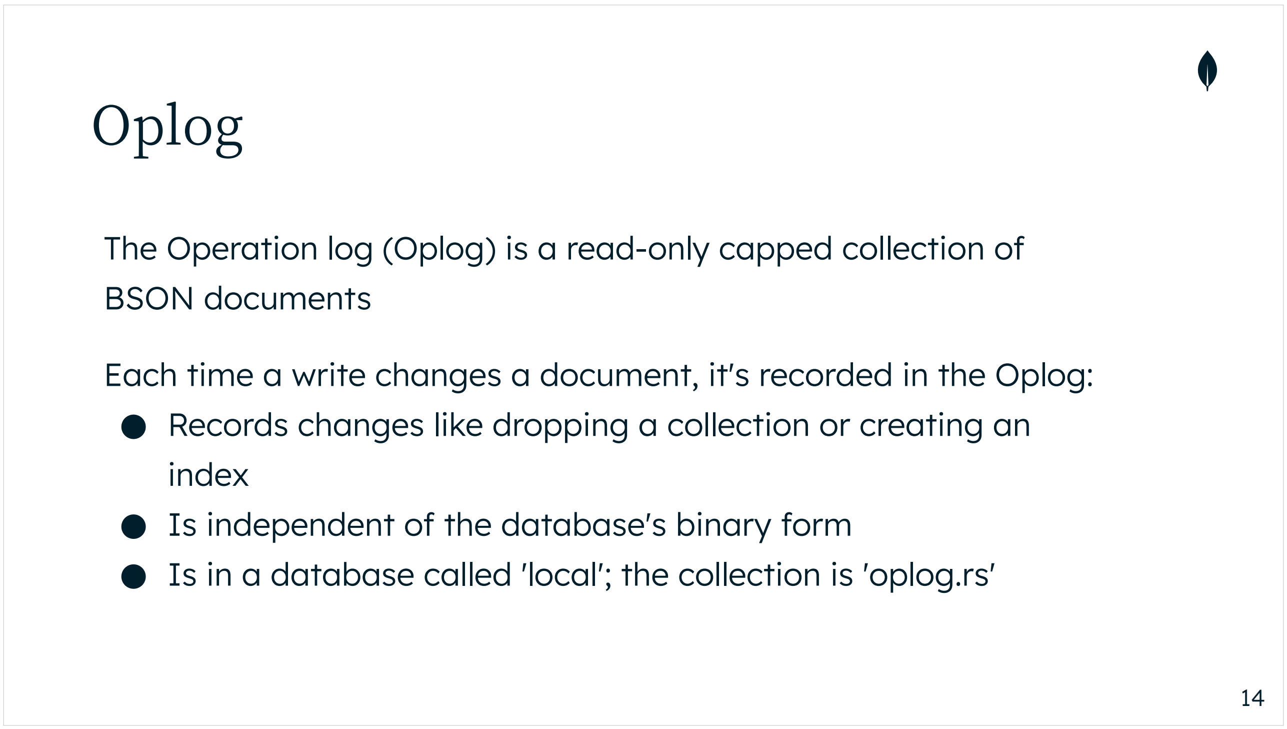
Replication process



- 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)



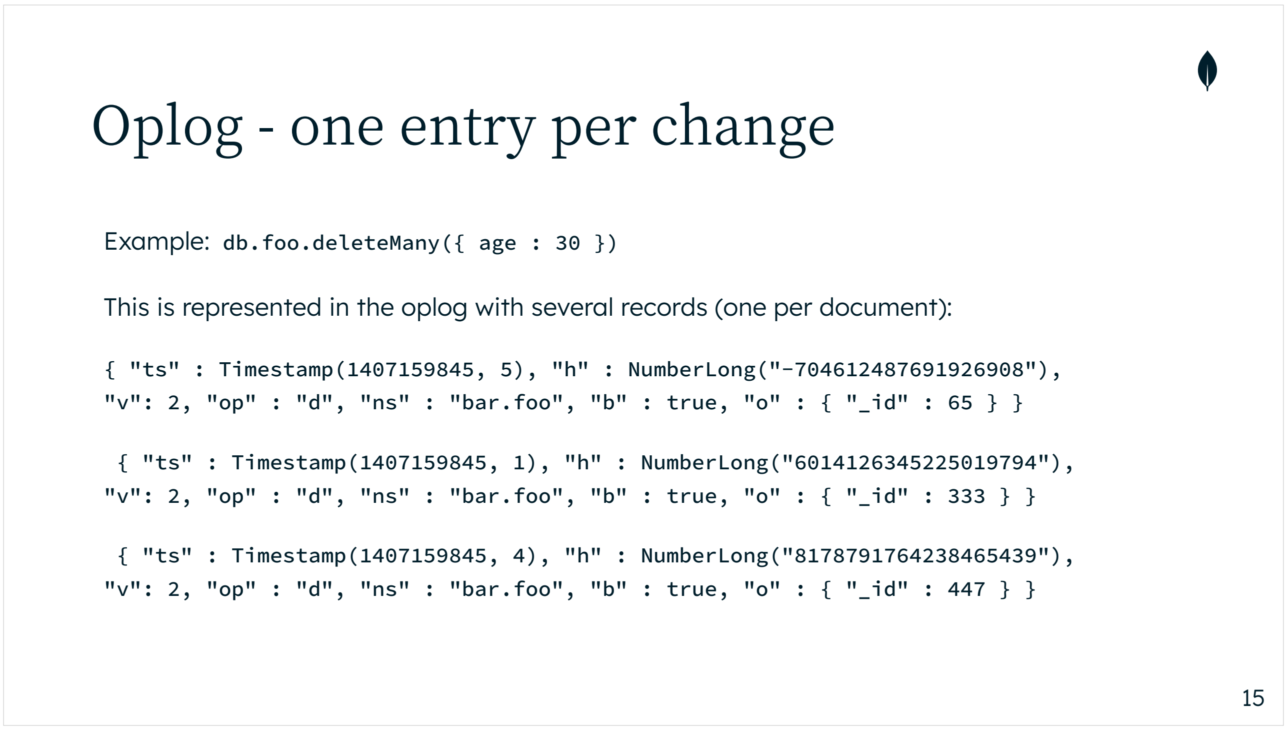
Oplog

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:

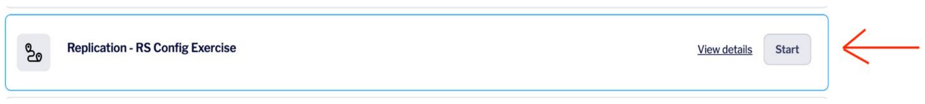
- `{$inc: {a: 2}}` becomes `{$set: {a: 5}}` assuming a was previously equal to 3
- `{$push: {b: "cheese"}}` becomes `{$set: {"b.8" : "cheese"}}`

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.

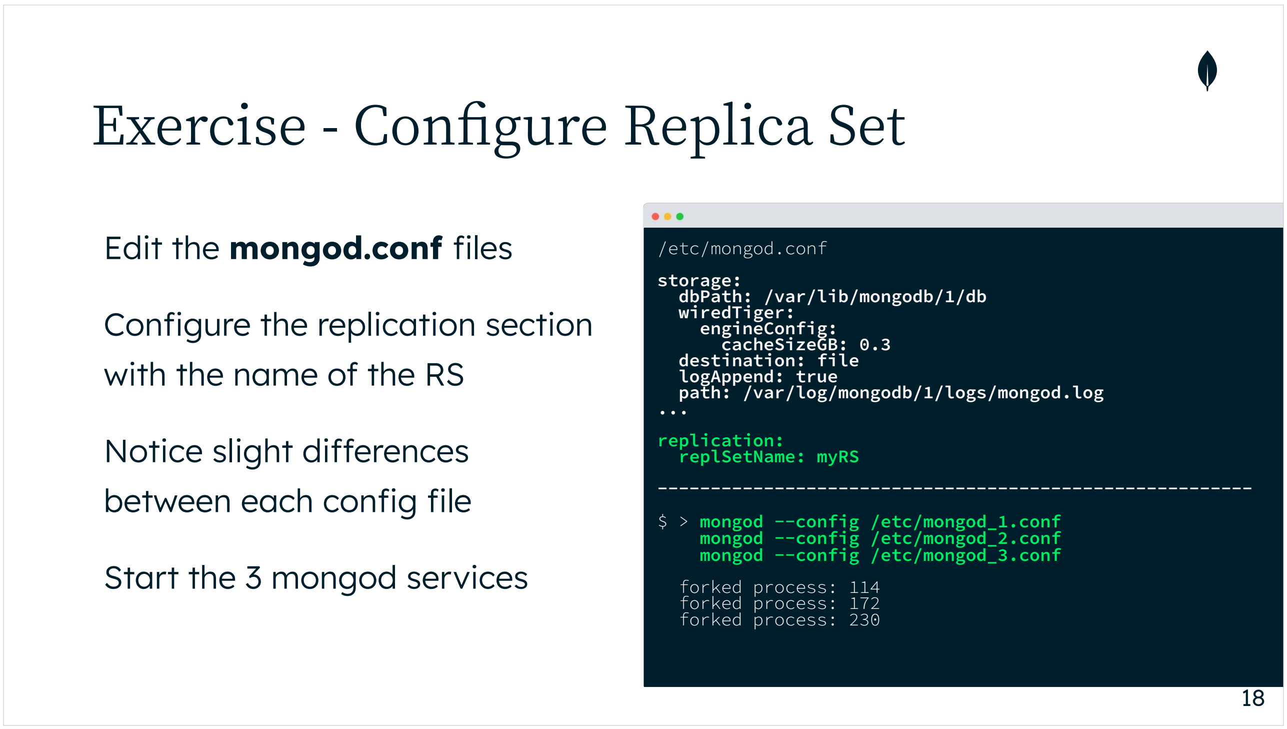
Exercise - Configure Replica Set

In this lab, you will be deploying your own replica set and will observe the oplog.

Start up the lab titled Replication - RS Config Exercise on Instruqt



NOTE! - the configuration and initialization portion of this exercise is intended to only be used in instruqt. If the instructor has not configured the training as a multi-track live event, then simply start a new event with the lab title ‘replication - rs config exercise.



Exercise - Configure Replica Set

Edit the **mongod.conf** files

Configure the replication section with the name of the RS

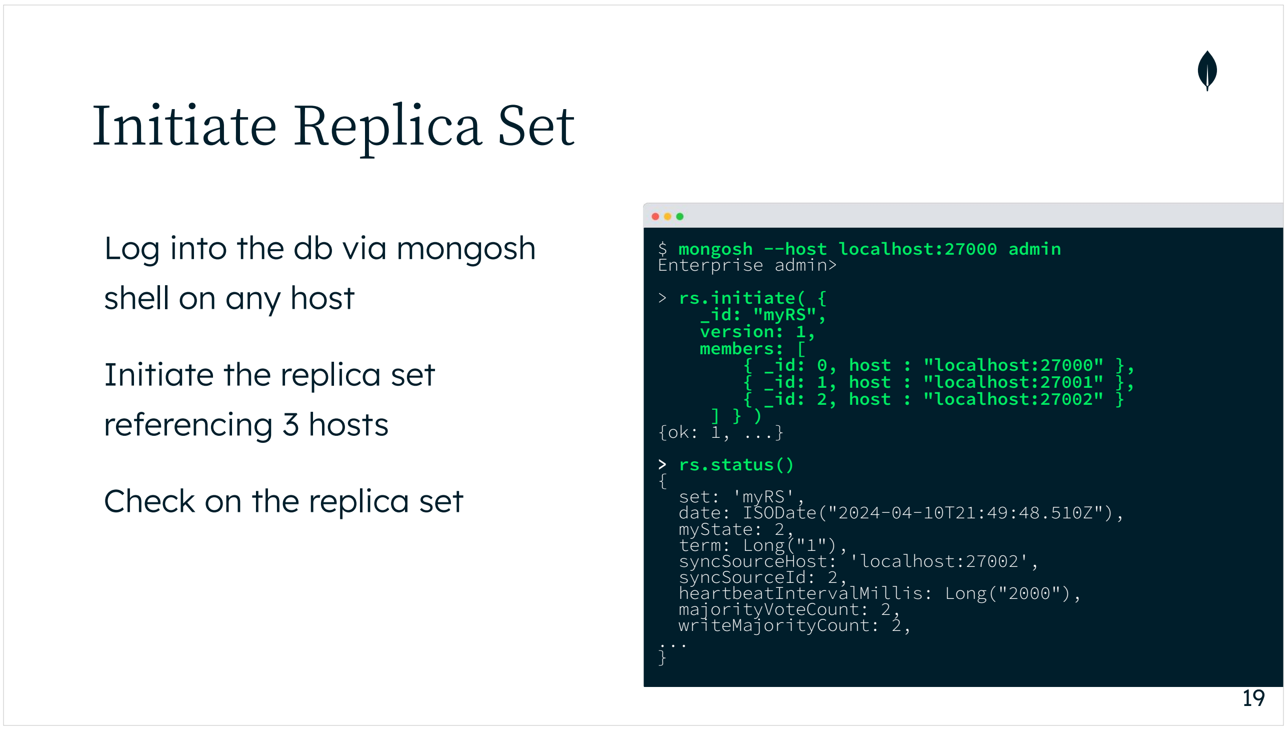
Notice slight differences between each config file

Start the 3 mongod services

```
/etc/mongod.conf

storage:
  dbPath: /var/lib/mongodb/1/db
  wiredTiger:
    engineConfig:
      cacheSizeGB: 0.3
  destination: file
  logAppend: true
  path: /var/log/mongodb/1/logs/mongod.log
...

replication:
  replSetName: myRS

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

$ > mongod --config /etc/mongod_1.conf
    mongod --config /etc/mongod_2.conf
    mongod --config /etc/mongod_3.conf

  forked process: 114
  forked process: 172
  forked process: 230
```

```
//spins up the mongod process based on the provided config file
mongod —config /etc/mongod_1.conf

//example of a config file
storage:
 dbPath: /var/lib/mongodb/1/db
 wiredTiger:
  engineConfig:
   cacheSizeGB: 0.3
...
systemLog:
 destination: file
 logAppend: true
 path: /var/log/mongodb/1/logs/mongod.log
# network interfaces
net:
 port: 27000
 bindIp: localhost,mongodb
# how the process runs
processManagement:
 timeZoneInfo: /usr/share/zoneinfo
 fork: true
#operationProfiling:
#todo - fill in replica set details
replication:
 replSetName: myRS
...
```

Security is being omitted here which is taught in a different section



Initiate Replica Set

Log into the db via mongosh shell on any host

Initiate the replica set referencing 3 hosts

Check on the replica set

```
$ mongosh --host localhost:27000 admin
Enterprise admin>

> rs.initiate( {
    _id: "myRS",
    version: 1,
    members: [
        { _id: 0, host : "localhost:27000" },
        { _id: 1, host : "localhost:27001" },
        { _id: 2, host : "localhost:27002" }
     ] } )
{ok: 1, ...}

> rs.status()
{
  set: 'myRS',
  date: ISODate("2024-04-10T21:49:48.510Z"),
  myState: 2,
  term: Long("1"),
  syncSourceHost: 'localhost:27002',
  syncSourceId: 2,
  heartbeatIntervalMillis: Long("2000"),
  majorityVoteCount: 2,
  writeMajorityCount: 2,
...
}
```

When initializing the replica set, you might find the server you are logged in on becomes the secondary - identify the primary using rs.status(), remember all write operations go through the primary server.

//will log user into the admin database on the host mentioned
mongosh —host localhost:27000 admin OR mongosh mongodb://localhost:27000

//initialize command used for the electable members -each host would be referenced by their own hostnames, default port is 27017. This configuration is running on the same host under 3 different ports

```
rs.initiate( {
  _id: "myRS",
  version: 1,
  members: [
    { _id: 0, host : "localhost:27000" },
    { _id: 1, host : "localhost:27001" },
    { _id: 2, host : "localhost:27002" }
 ] } )
```

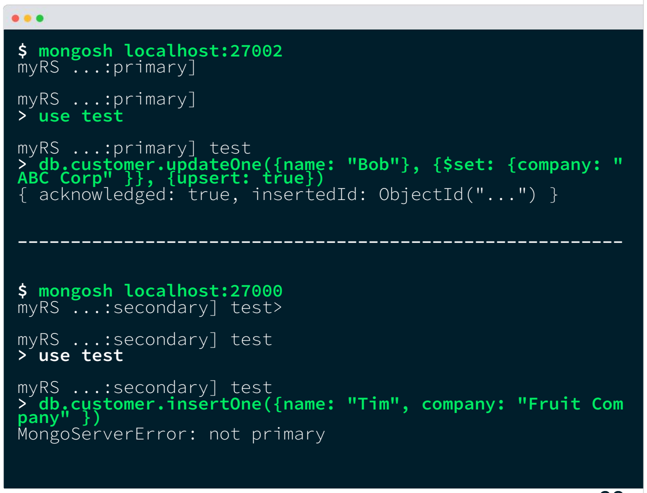
Oplog and Replication

On the primary server, create a record in a namespace

Switch tabs and log into a specific secondary host

Query for the inserted record in the correlating namespace

Try to write on a secondary



```
$ mongosh localhost:27002
myRS ...:primary]

myRS ...:primary]
> use test

myRS ...:primary] test
> db.customer.updateOne({name: "Bob"}, {$set: {company: "
ABC Corp" }}, {upsert: true})
{ acknowledged: true, insertedId: ObjectId("...") }

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

$ mongosh localhost:27000
myRS ...:secondary] test>

myRS ...:secondary] test
> use test

myRS ...:secondary] test
> db.customer.insertOne({name: "Tim", company: "Fruit Com
pany" })
MongoServerError: not primary
```

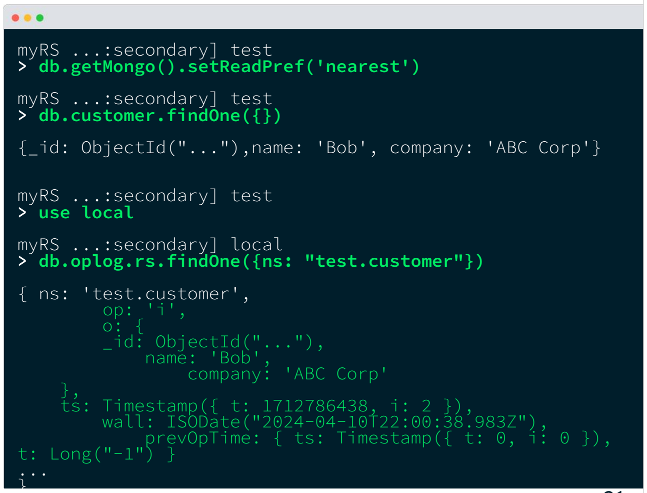
We can log into a single instance rather than the set
We normally don't do this except for maintenance tasks (like shutting it down)
On a Secondary we cannot do anything until we acknowledge it is a secondary. However, we cannot write to a secondary ever.

Viewing the oplog

Allow for read operations to happen directly from the secondary

Look into the oplog to understand how the record created in the primary is on the secondary

Oplog can be found in the **local.oplog.rs** namespace



```
myRS ...:secondary] test
> db.getMongo().setReadPref('nearest')

myRS ...:secondary] test
> db.customer.findOne({})

{_id: ObjectId("..."),name: 'Bob', company: 'ABC Corp'}


myRS ...:secondary] test
> use local

myRS ...:secondary] local
> db.oplog.rs.findOne({ns: "test.customer"})

{ ns: 'test.customer',
        op: 'i',
        o: {
        _id: ObjectId("..."),
            name: 'Bob',
                company: 'ABC Corp'
    },
    ts: Timestamp({ t: 1712786438, i: 2 }),
        wall: ISODate("2024-04-10T22:00:38.983Z"),
            prevOpTime: { ts: Timestamp({ t: 0, i: 0 }),
t: Long("-1") }
...
}
```

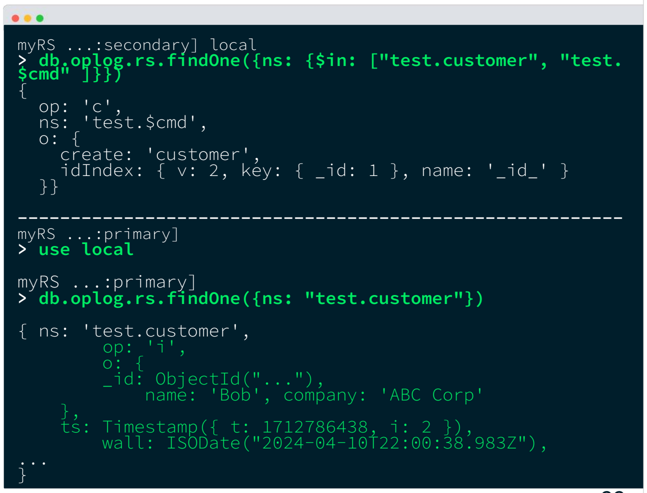
In the legacy mongo shell, a method called `rs.secondaryOk()` is available that provides us the acknowledgement so that data can be read from the secondary.

Viewing the oplog

The $cmd shows the internal database command that created the collection customer as a result of the previous insert.

Log in or switch tabs to the primary server.

Verify the record is identical to the one in the secondary.



```
myRS ...:secondary] local
> db.oplog.rs.findOne({ns: {$in: ["test.customer", "test.
$cmd" ]}})
{
  op: 'c',
  ns: 'test.$cmd',
  o: {
    create: 'customer',
    idIndex: { v: 2, key: { _id: 1 }, name: '_id_' }
  }}

----------------------------------------------------------
myRS ...:primary]
> use local

myRS ...:primary]
> db.oplog.rs.findOne({ns: "test.customer"})

{ ns: 'test.customer',
        op: 'i',
        o: {
        _id: ObjectId("..."),
            name: 'Bob', company: 'ABC Corp'
    },
    ts: Timestamp({ t: 1712786438, i: 2 }),
        wall: ISODate("2024-04-10T22:00:38.983Z"),
...
}
```

You can see the record on the oplog on the Primary - and its an idempotent change - due to the upsert being applied.
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 so we can query by specific namespaces or 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.
The OpLog may look different, as per the version of the MongoDB server you are using. This version is referencing mongoDB logs in version 7.0.5

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)

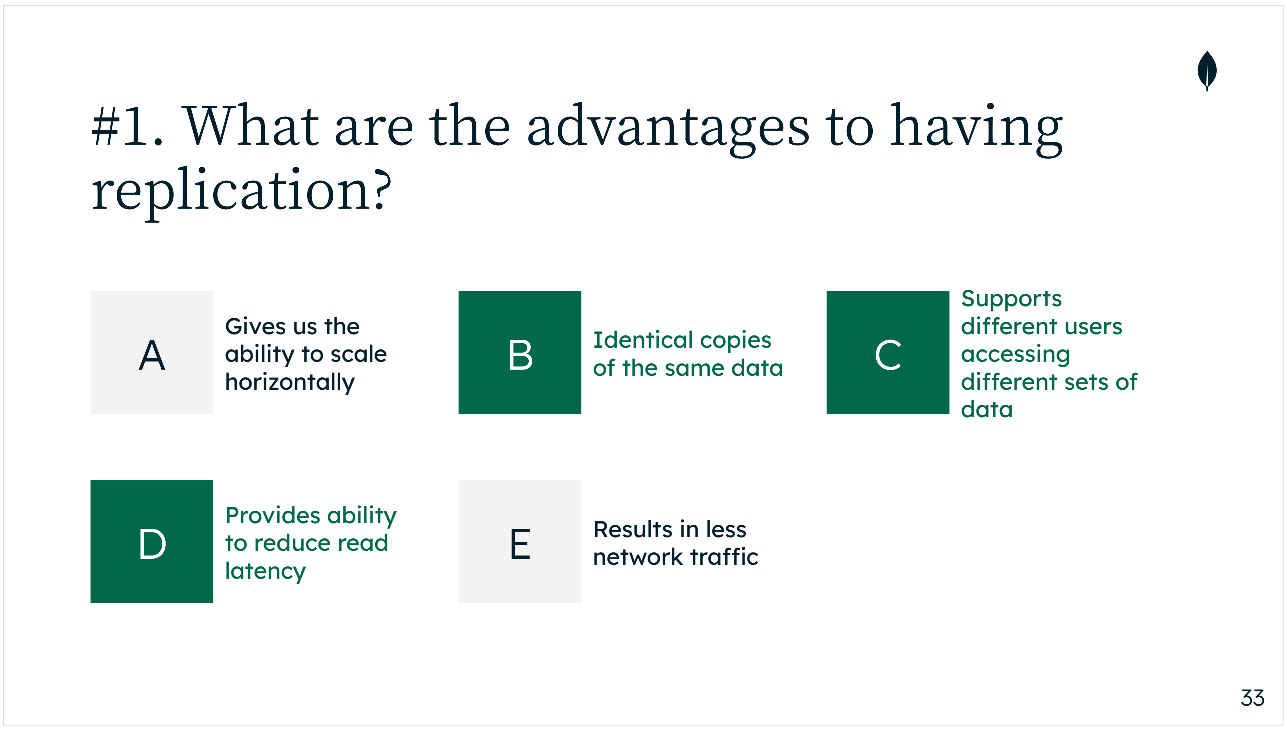
- Can auto restart on non-transient error
- Resumable on transient error
- Copy Oplog at the same time
- Overall more resilient
- Can use a secondary as source

Quiz Time!

#1. What are the advantages to having replication?

A	Gives us the ability to scale horizontally
B	Identical copies of the same data
C	Supports different users accessing different sets of data
D	Provides ability to reduce read latency
E	Results in less network traffic

Answer in the next slide.



#1. What are the advantages to having replication?

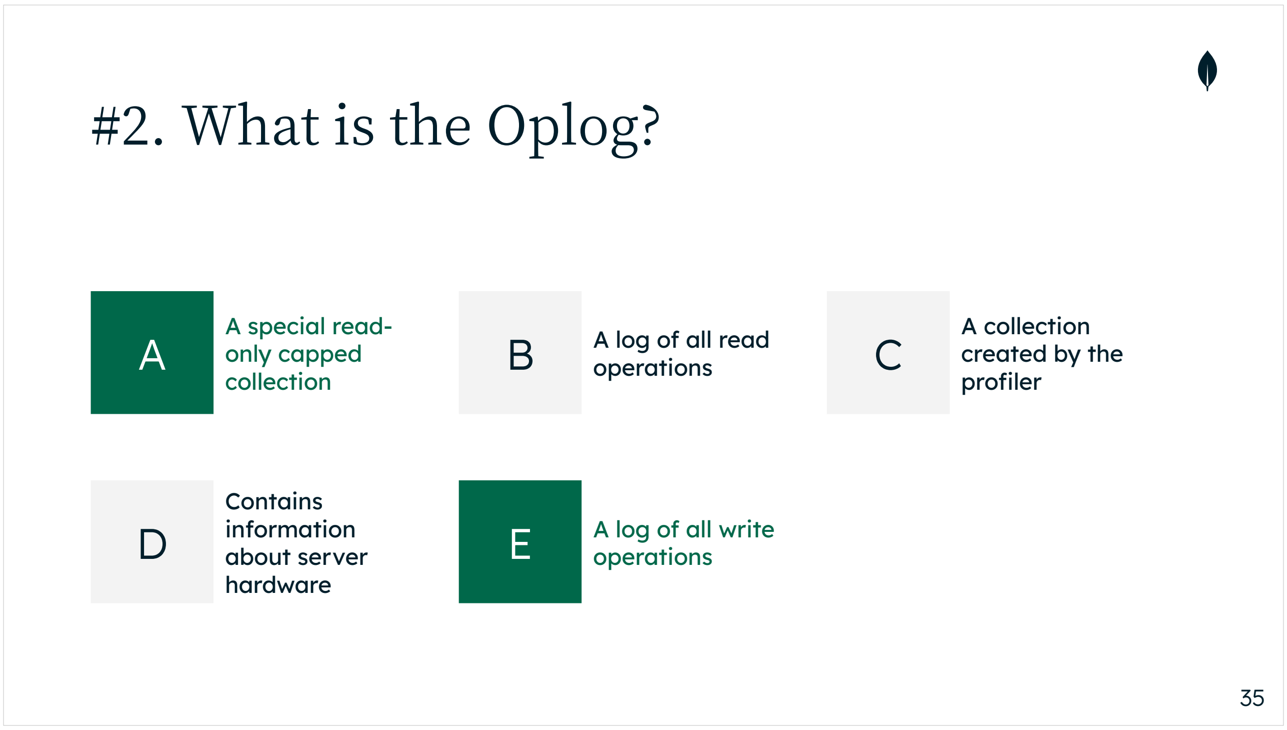
- A — Gives us the ability to scale horizontally
- [✓ B] — Identical copies of the same data
- [✓ C] — Supports different users accessing different sets of data
- [✓ D] — Provides ability to reduce read latency
- E — Results in less network traffic

Scaling is done via sharding, not replication and because there would be multiple nodes in a RS - it would involve additional network traffic.

#2. What is the Oplog?

A	A special read-only capped collection
B	A log of all read operations
C	A collection created by the profiler
D	Contains information about server hardware
E	A log of all write operations

Answer in the next slide.



#2. What is the Oplog?

[✓ A] A special read-only capped collection

B A log of all read operations

C A collection created by the profiler

D Contains information about server hardware

[✓ E] A log of all write operations

The oplog is an internal, read-only, capped collection. It contains all write operations (inserts, updates, deletes) made on the respective server. It is used by replica set nodes to stay in sync with each other. By storing the oplog with a backup, we can perform point-in-time restores.

Recap

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

Writes to a Primary are replicated to Secondaries

The oplog is a capped collection that is read by the secondaries to perform the same write operation