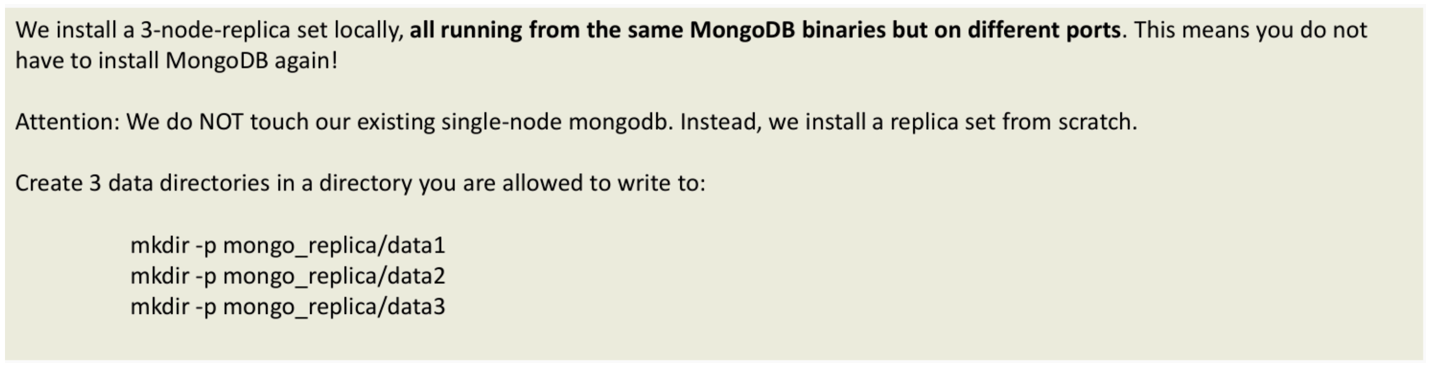
# A12

| Type | Homework |
|---|---|

## MongoDB Replica Set Installation



We install a 3-node-replica set locally, **all running from the same MongoDB binaries but on different ports**. This means you do not have to install MongoDB again!

Attention: We do NOT touch our existing single-node mongodb. Instead, we install a replica set from scratch.

Create 3 data directories in a directory you are allowed to write to:

```
mkdir -p mongo_replica/data1
mkdir -p mongo_replica/data2
mkdir -p mongo_replica/data3
```

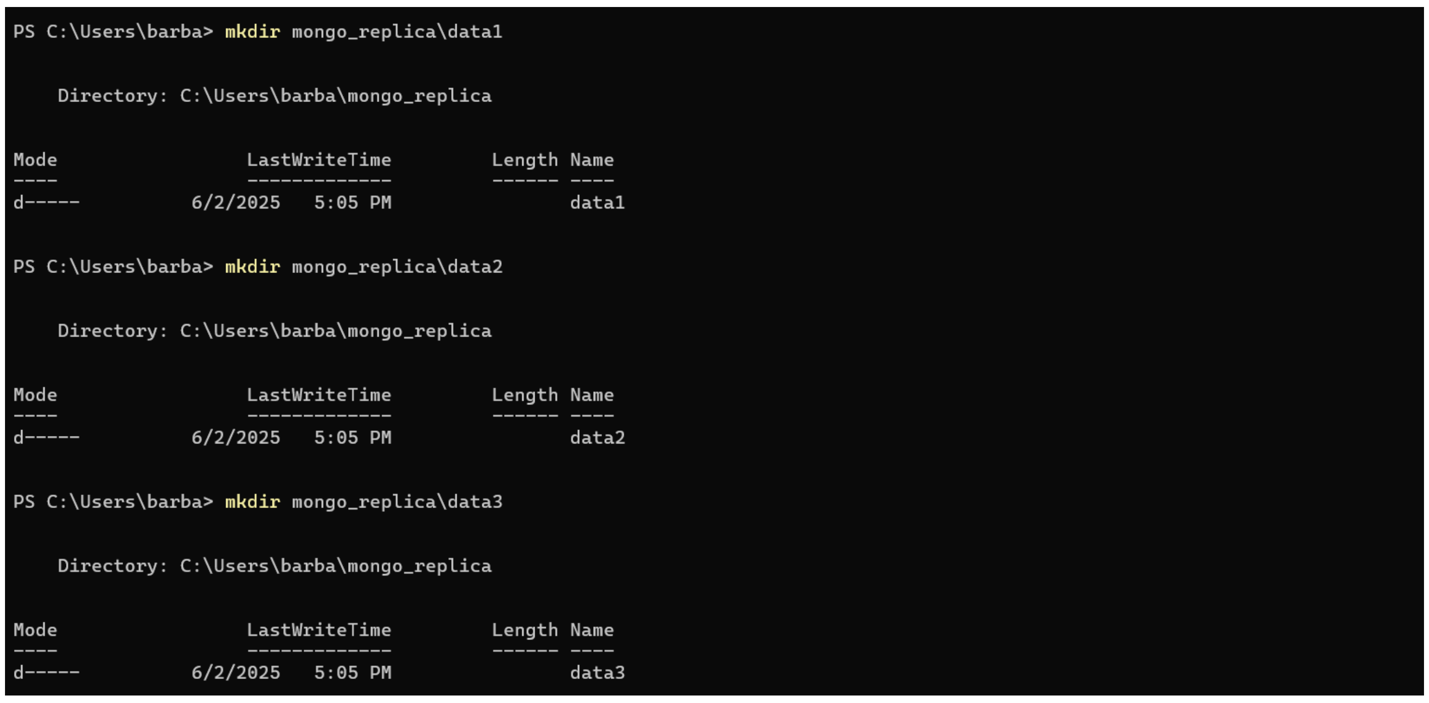
❗ TO DELETE THIS REPLICAS

```
rmdir /s /q mongo_replica\data1
rmdir /s /q mongo_replica\data2
rmdir /s /q mongo_replica\data3
```

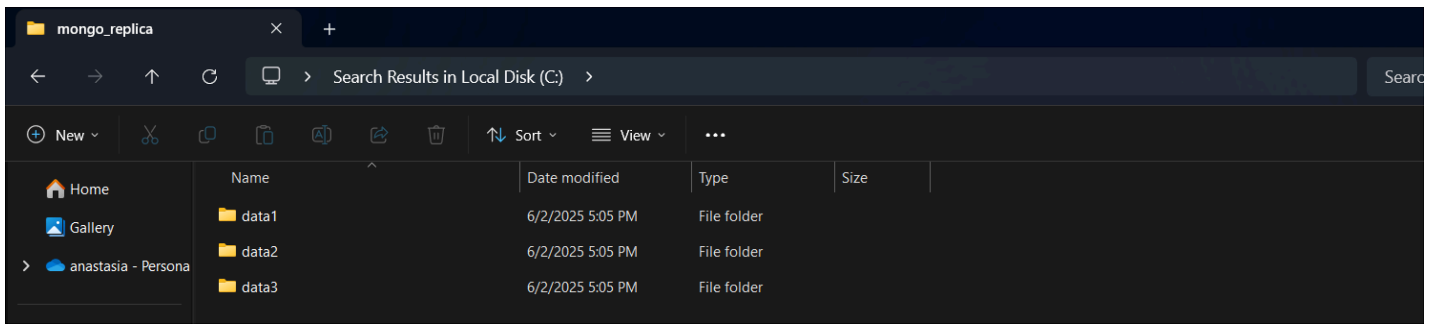
### Create data directories in cmd:

```
mkdir mongo_replica\data1
mkdir mongo_replica\data2
mkdir mongo_replica\data3
```

Result:

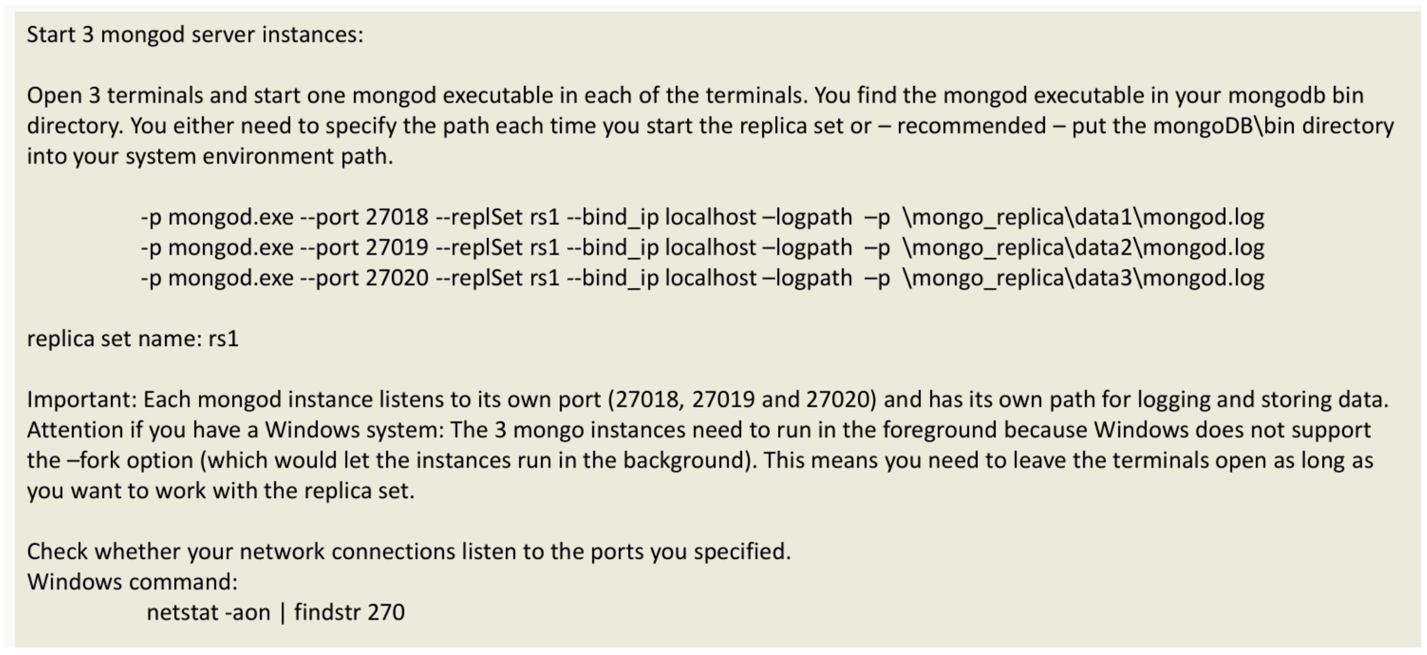


```
PS C:\Users\barba> mkdir mongo_replica\data1

    Directory: C:\Users\barba\mongo_replica

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          6/2/2025   5:05 PM                data1

PS C:\Users\barba> mkdir mongo_replica\data2

    Directory: C:\Users\barba\mongo_replica

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          6/2/2025   5:05 PM                data2

PS C:\Users\barba> mkdir mongo_replica\data3

    Directory: C:\Users\barba\mongo_replica

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          6/2/2025   5:05 PM                data3
```



| Name | Date modified | Type | Size |
|---|---|---|---|
| data1 | 6/2/2025 5:05 PM | File folder | |
| data2 | 6/2/2025 5:05 PM | File folder | |
| data3 | 6/2/2025 5:05 PM | File folder | |

## MongoDB Replica Set



Start 3 mongod server instances:

Open 3 terminals and start one mongod executable in each of the terminals. You find the mongod executable in your mongodb bin directory. You either need to specify the path each time you start the replica set or – recommended – put the mongoDB\bin directory into your system environment path.

```
-p mongod.exe --port 27018 --replSet rs1 --bind_ip localhost –logpath  –p  \mongo_replica\data1\mongod.log
-p mongod.exe --port 27019 --replSet rs1 --bind_ip localhost –logpath  –p  \mongo_replica\data2\mongod.log
-p mongod.exe --port 27020 --replSet rs1 --bind_ip localhost –logpath  –p  \mongo_replica\data3\mongod.log
```

replica set name: rs1

Important: Each mongod instance listens to its own port (27018, 27019 and 27020) and has its own path for logging and storing data. Attention if you have a Windows system: The 3 mongo instances need to run in the foreground because Windows does not support the –fork option (which would let the instances run in the background). This means you need to leave the terminals open as long as you want to work with the replica set.

Check whether your network connections listen to the ports you specified.
Windows command:

```
netstat -aon | findstr 270
```

Start 3 MongoDB instances with different ports

CMD 1:

```
mongod.exe --port 27018 --replSet rs1 --bind_ip localhost --logpath mongo_replica/data1/mongod.log --dbpath mongo_r
```

CMD 2:

```
mongod.exe --port 27019 --replSet rs1 --bind_ip localhost --logpath mongo_replica/data2/mongod.log --dbpath mongo_r
```

CMD 3:

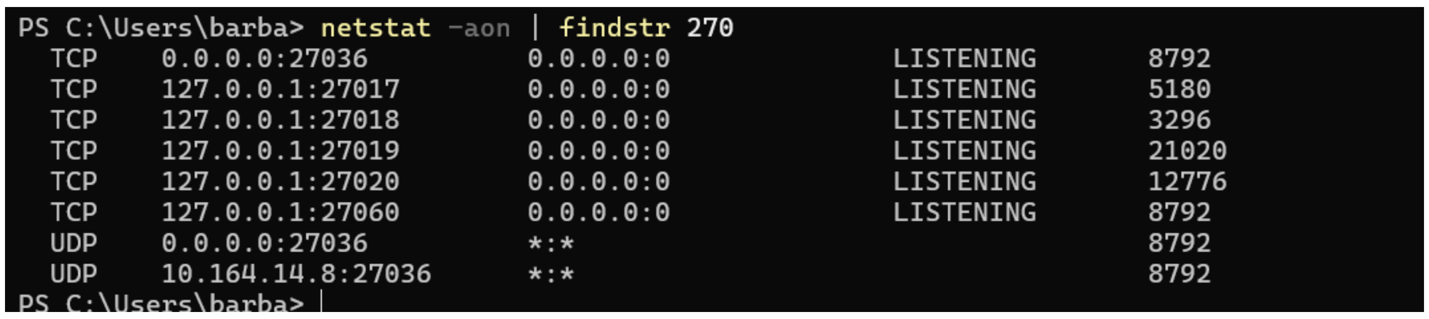
```
mongod.exe --port 27020 --replSet rs1 --bind_ip localhost --logpath mongo_replica/data3/mongod.log --dbpath mongo_
```

all three windows should be left open.

Check that all mongod processes are listening on the expected ports:

```
netstat -aon | findstr 270
```

Result:



```
PS C:\Users\barba> netstat -aon | findstr 270
  TCP    0.0.0.0:27036          0.0.0.0:0              LISTENING       8792
  TCP    127.0.0.1:27017        0.0.0.0:0              LISTENING       5180
  TCP    127.0.0.1:27018        0.0.0.0:0              LISTENING       3296
  TCP    127.0.0.1:27019        0.0.0.0:0              LISTENING       21020
  TCP    127.0.0.1:27020        0.0.0.0:0              LISTENING       12776
  TCP    127.0.0.1:27060        0.0.0.0:0              LISTENING       8792
  UDP    0.0.0.0:27036          *:*                                    8792
  UDP    10.164.14.8:27036      *:*                                    8792
PS C:\Users\barba> |
```

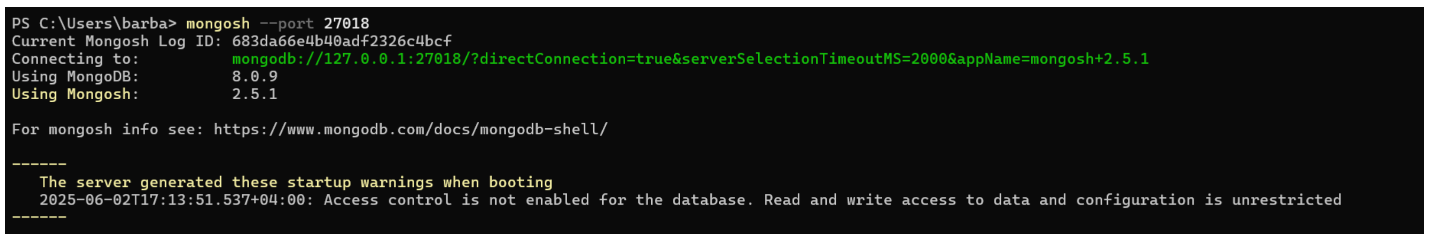
## MongoDB Replica Set

Connect to the server instances via mongo shell.

CMD 4:

```
mongosh --port 27018
```

You automatically connect to the default database test that comes with MongoDB. You can use the test database.

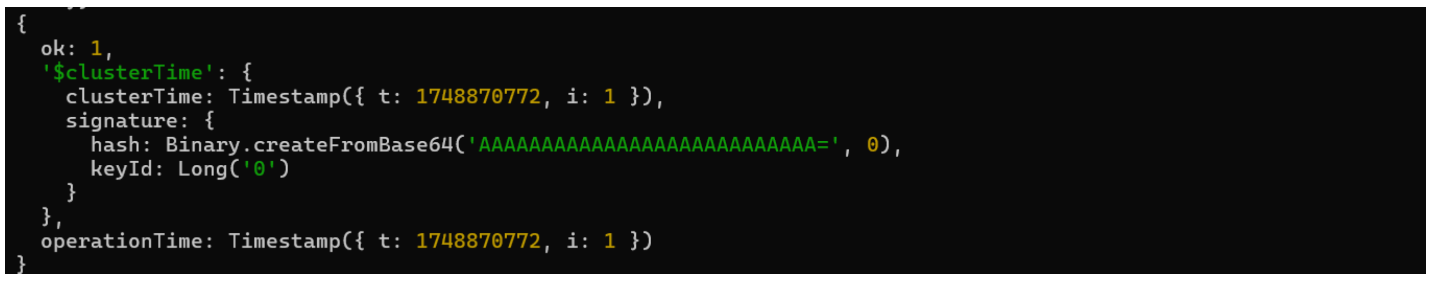


```
PS C:\Users\barba> mongosh --port 27018
Current Mongosh Log ID: 683da66e4b40adf2326c4bcf
Connecting to:          mongodb://127.0.0.1:27018/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.1
Using MongoDB:          8.0.9
Using Mongosh:          2.5.1

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-06-02T17:13:51.537+04:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
------
```

Then in the shell (CMD 4) initiate the replica set:

```
rs.initiate({
 _id: "rs1",
 members: [
   { _id: 0, host: "localhost:27018" },
   { _id: 1, host: "localhost:27019" },
   { _id: 2, host: "localhost:27020" }
 ]
})
```

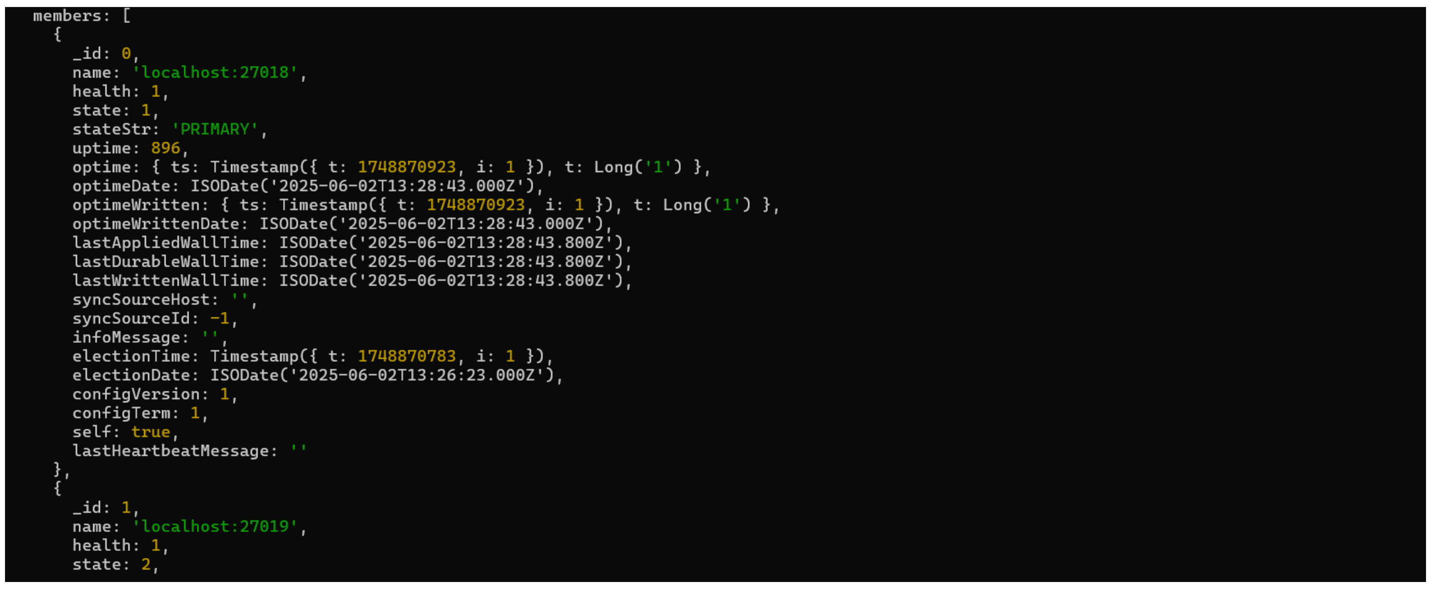
Result:



```
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1748870772, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1748870772, i: 1 })
}
```

You immediately see that you work on the primary.
Verify that your replica set is running:

```
rs.status()
```

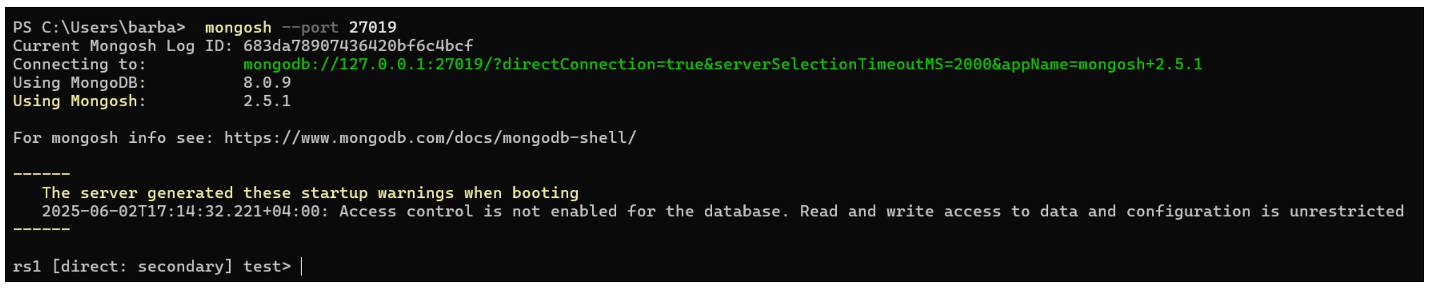


```
  members: [
    {
      _id: 0,
      name: 'localhost:27018',
      health: 1,
      state: 1,
      stateStr: 'PRIMARY',
      uptime: 896,
      optime: { ts: Timestamp({ t: 1748870923, i: 1 }), t: Long('1') },
      optimeDate: ISODate('2025-06-02T13:28:43.000Z'),
      optimeWritten: { ts: Timestamp({ t: 1748870923, i: 1 }), t: Long('1') },
      optimeWrittenDate: ISODate('2025-06-02T13:28:43.000Z'),
      lastAppliedWallTime: ISODate('2025-06-02T13:28:43.800Z'),
      lastDurableWallTime: ISODate('2025-06-02T13:28:43.800Z'),
      lastWrittenWallTime: ISODate('2025-06-02T13:28:43.800Z'),
      syncSourceHost: '',
      syncSourceId: -1,
      infoMessage: '',
      electionTime: Timestamp({ t: 1748870783, i: 1 }),
      electionDate: ISODate('2025-06-02T13:26:23.000Z'),
      configVersion: 1,
      configTerm: 1,
      self: true,
      lastHeartbeatMessage: ''
    },
    {
      _id: 1,
      name: 'localhost:27019',
      health: 1,
      state: 2,
```

**<u>Open two more terminals and start a mongo shell in each of them connecting to the other ports:</u>**

CMD 5:

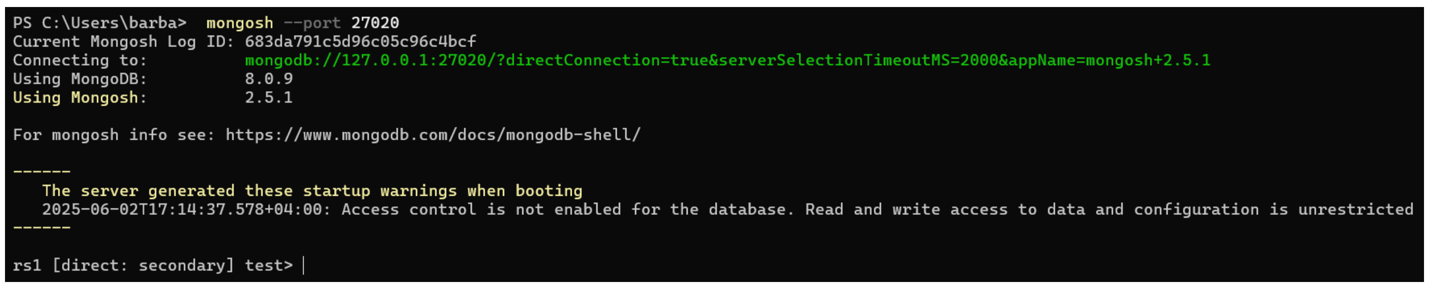
```
mongosh --port 27019
```



```
PS C:\Users\barba>  mongosh --port 27019
Current Mongosh Log ID: 683da78907436420bf6c4bcf
Connecting to:          mongodb://127.0.0.1:27019/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.1
Using MongoDB:          8.0.9
Using Mongosh:          2.5.1

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-06-02T17:14:32.221+04:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
------

rs1 [direct: secondary] test> |
```

CMD 6:

```
mongosh --port 27020
```



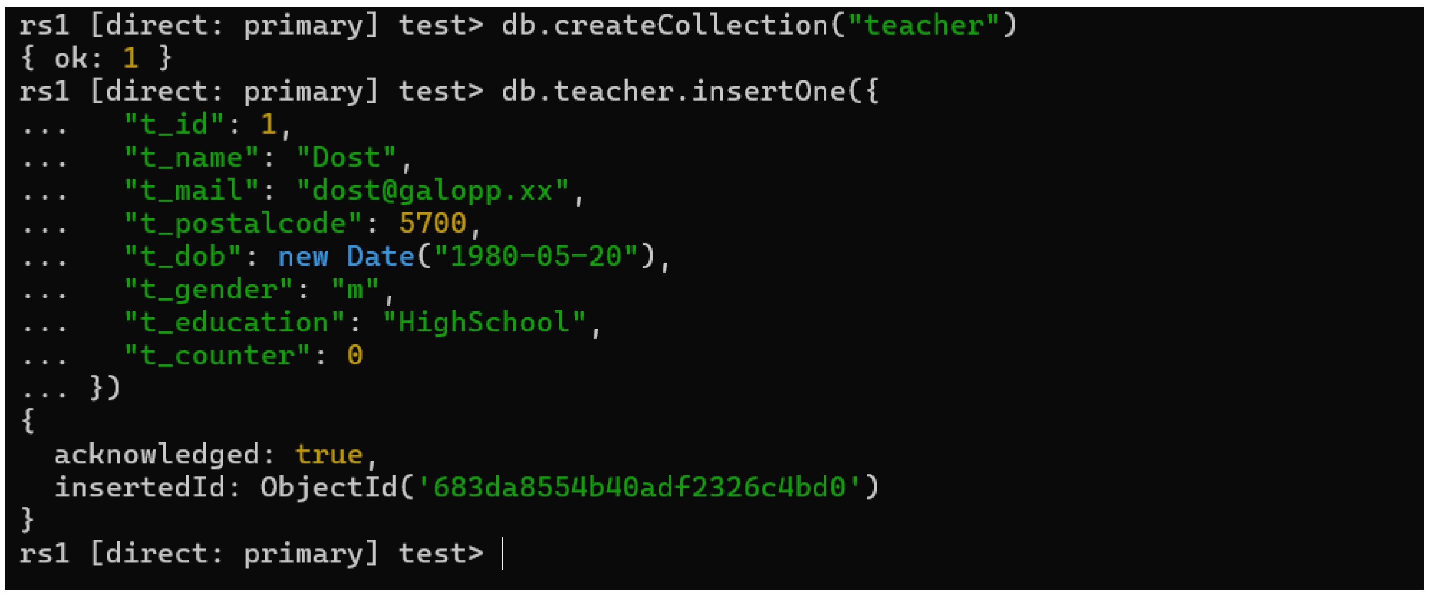
```
PS C:\Users\barba>  mongosh --port 27020
Current Mongosh Log ID: 683da791c5d96c05c96c4bcf
Connecting to:          mongodb://127.0.0.1:27020/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+2.5.1
Using MongoDB:          8.0.9
Using Mongosh:          2.5.1

For mongosh info see: https://www.mongodb.com/docs/mongodb-shell/

------
   The server generated these startup warnings when booting
   2025-06-02T17:14:37.578+04:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
------

rs1 [direct: secondary] test>
```

You immediately see that you are working on secondaries.

**On the primary (CMD 4), create a teacher collection and insert a teacher document.**

```
db.createCollection("teacher")

db.teacher.insertOne({
  "t_id": 1,
  "t_name": "Dost",
  "t_mail": "dost@galopp.xx",
  "t_postalcode": 5700,
  "t_dob": new Date("1980-05-20"),
  "t_gender": "m",
  "t_education": "HighSchool",
  "t_counter": 0
})
```

Result:



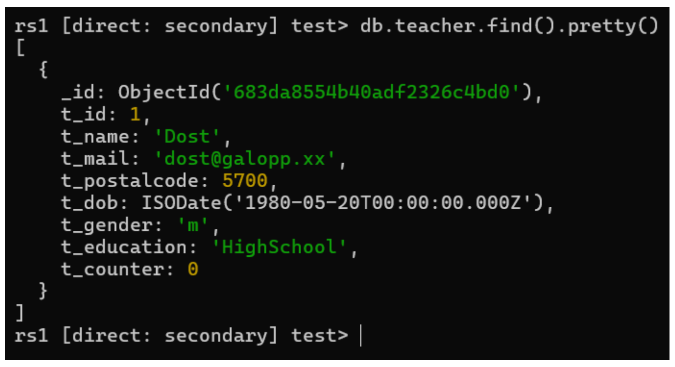
```
rs1 [direct: primary] test> db.createCollection("teacher")
{ ok: 1 }
rs1 [direct: primary] test> db.teacher.insertOne({
...    "t_id": 1,
...    "t_name": "Dost",
...    "t_mail": "dost@galopp.xx",
...    "t_postalcode": 5700,
...    "t_dob": new Date("1980-05-20"),
...    "t_gender": "m",
...    "t_education": "HighSchool",
...    "t_counter": 0
... })
{
  acknowledged: true,
  insertedId: ObjectId('683da8554b40adf2326c4bd0')
}
rs1 [direct: primary] test>
```

**On the secondary(CMD 5 and CMD 6), check if the document was replicated.**

```
db.teacher.find().pretty()
```

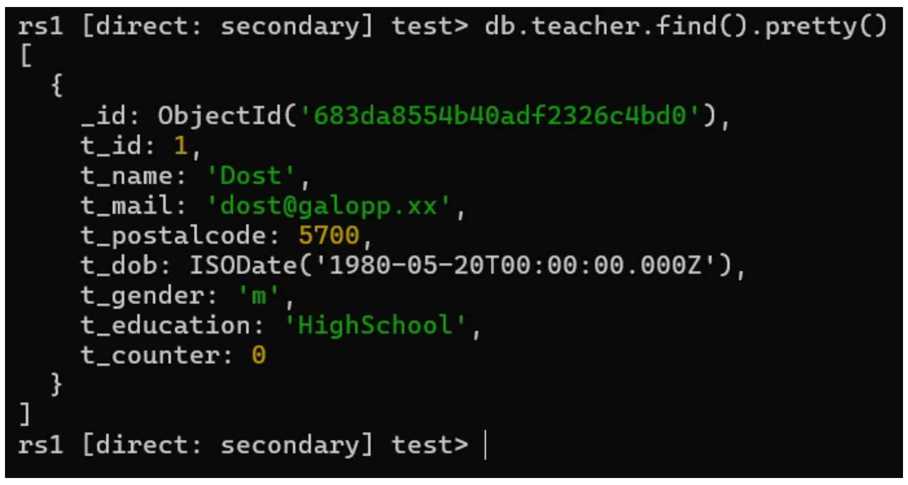
Result:

CMD 5:



```
rs1 [direct: secondary] test> db.teacher.find().pretty()
[
  {
    _id: ObjectId('683da8554b40adf2326c4bd0'),
    t_id: 1,
    t_name: 'Dost',
    t_mail: 'dost@galopp.xx',
    t_postalcode: 5700,
    t_dob: ISODate('1980-05-20T00:00:00.000Z'),
    t_gender: 'm',
    t_education: 'HighSchool',
    t_counter: 0
  }
]
rs1 [direct: secondary] test>
```

CMD 6:



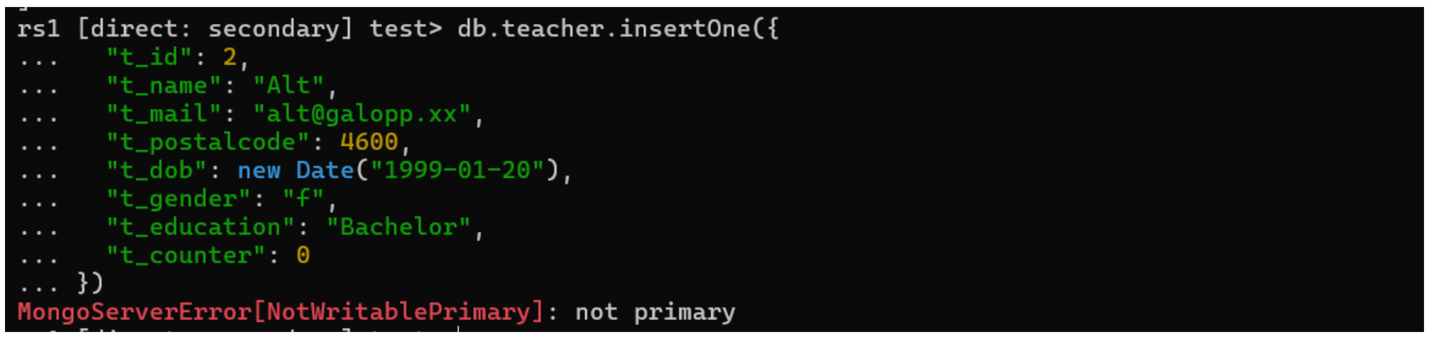
```
rs1 [direct: secondary] test> db.teacher.find().pretty()
[
  {
    _id: ObjectId('683da8554b40adf2326c4bd0'),
    t_id: 1,
    t_name: 'Dost',
    t_mail: 'dost@galopp.xx',
    t_postalcode: 5700,
    t_dob: ISODate('1980-05-20T00:00:00.000Z'),
    t_gender: 'm',
    t_education: 'HighSchool',
    t_counter: 0
  }
]
rs1 [direct: secondary] test>
```

It was replicated on both secondaries.

<u>**On the secondary (CMD 5 or CMD 6), try to insert another document.**</u>

```
db.teacher.insertOne({
  "t_id": 2,
  "t_name": "Alt",
  "t_mail": "alt@galopp.xx",
  "t_postalcode": 4600,
  "t_dob": new Date("1999-01-20"),
  "t_gender": "f",
  "t_education": "Bachelor",
  "t_counter": 0
})
```

Result:



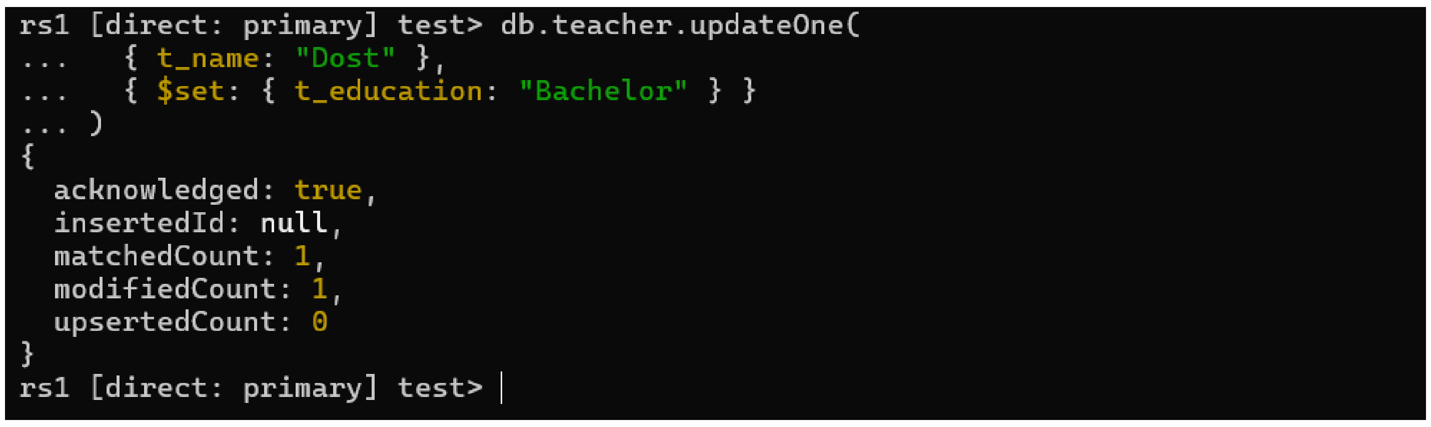
```
rs1 [direct: secondary] test> db.teacher.insertOne({
...     "t_id": 2,
...     "t_name": "Alt",
...     "t_mail": "alt@galopp.xx",
...     "t_postalcode": 4600,
...     "t_dob": new Date("1999-01-20"),
...     "t_gender": "f",
...     "t_education": "Bachelor",
...     "t_counter": 0
... })
MongoServerError[NotWritablePrimary]: not primary
```

This insert operation failed, because by default writes are not allowed on secondaries.

<u>**On the primary (CMD 4), we update the document:**</u>

```
db.teacher.updateOne(
  { t_name: "Dost" },
    { $set: { t_education: "Bachelor" } }
)
```

Result:



```
rs1 [direct: primary] test> db.teacher.updateOne(
...     { t_name: "Dost" },
...     { $set: { t_education: "Bachelor" } }
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
rs1 [direct: primary] test> |
```

---

## MongoDB Write Concern

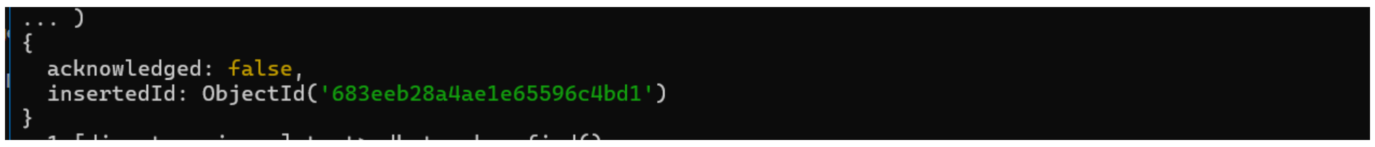
1. <u>**Insert a document with write concern set to 0, What does Mongodb return? What is the result of the write operation?**</u>

   (we can not insert in the secondaries, only primary)

   ```
   db.teacher.insertOne({
     "t_id": 13,
     "t_name": "Noah",
     "t_mail": "noah@galopp.xx",
     "t_postalcode": 5000,
     "t_dob": new Date("1999-01-20"),
   ```

```
    "t_gender": "m",
    "t_education": "Bachelor",
    "t_counter": 0
  },
     { writeConcern: { w: 0}}
  )
```

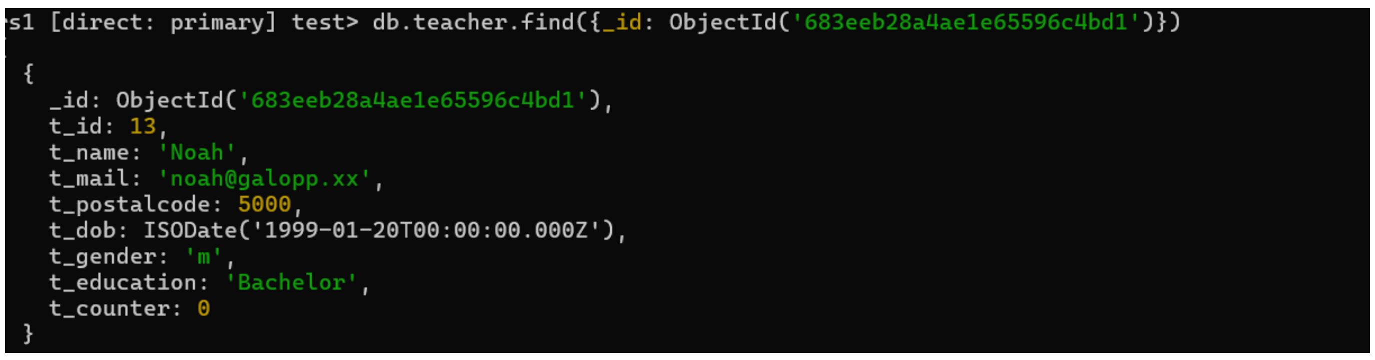
Result:



```
... )
{
  acknowledged: false,
  insertedId: ObjectId('683eeb28a4ae1e65596c4bd1')
}
```

w:0 → Don't wait for any acknowledgment

`acknowledged: false` , meaning that we have no confirmation of success or failure but the write was likely executed, we call `db.teacher.find({_id: ObjectId('683eeb28a4ae1e65596c4bd1')})`



```
s1 [direct: primary] test> db.teacher.find({_id: ObjectId('683eeb28a4ae1e65596c4bd1')})

 {
   _id: ObjectId('683eeb28a4ae1e65596c4bd1'),
   t_id: 13,
   t_name: 'Noah',
   t_mail: 'noah@galopp.xx',
   t_postalcode: 5000,
   t_dob: ISODate('1999-01-20T00:00:00.000Z'),
   t_gender: 'm',
   t_education: 'Bachelor',
   t_counter: 0
 }
```

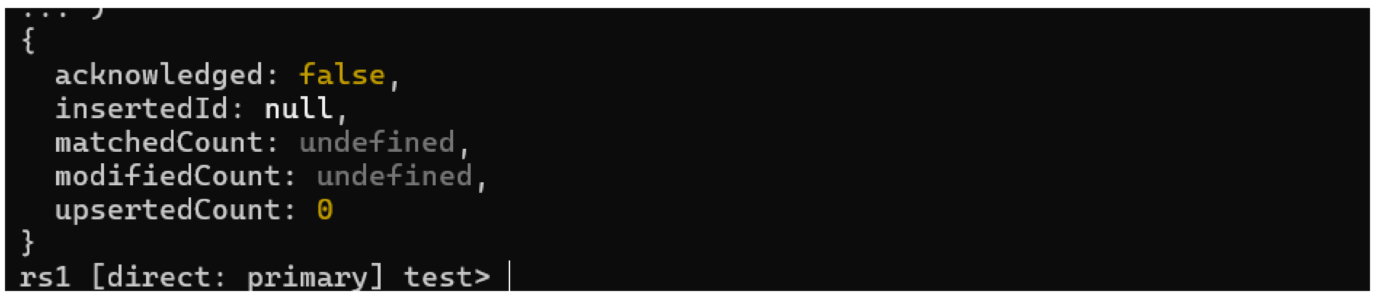
so even though it gave us no confirmation the write operation was executed (replicated on secondaries as well).

2. **Update the document with write concern set to 0. What does Mongodb return? Is the write operation executed?**

(can not update on secondaries)

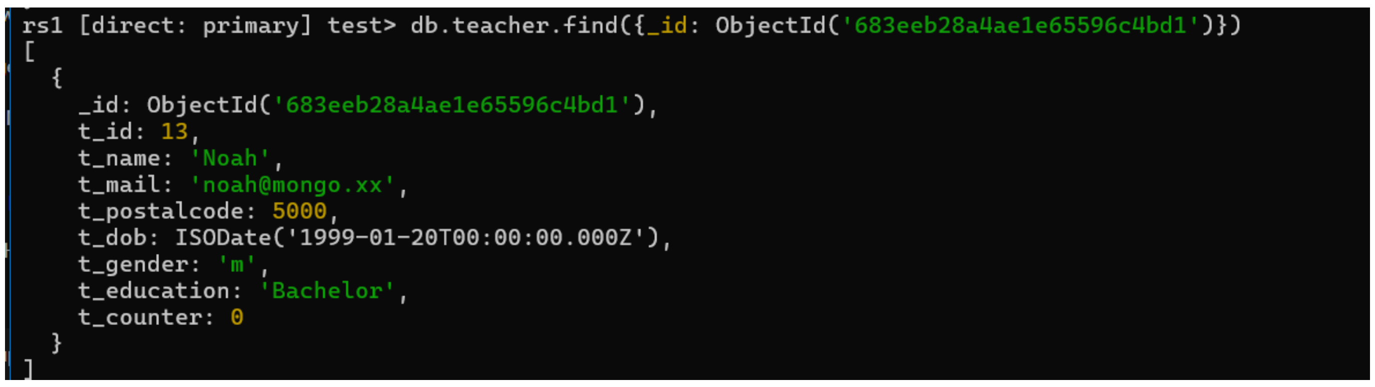
```
db.teacher.updateOne(
  { t_id: 13 },
  { $set: { t_mail: "noah@mongo.xx" } },
  { writeConcern: { w: 0 } }
)
```

Result:



```
... )
{
  acknowledged: false,
  insertedId: null,
  matchedCount: undefined,
  modifiedCount: undefined,
  upsertedCount: 0
}
rs1 [direct: primary] test>
```

once again the `aknowledged: false` , so no confirmation, but the write was succesful, we check it by calling `db.teacher.find({_id: ObjectId('683eeb28a4ae1e65596c4bd1')})` .



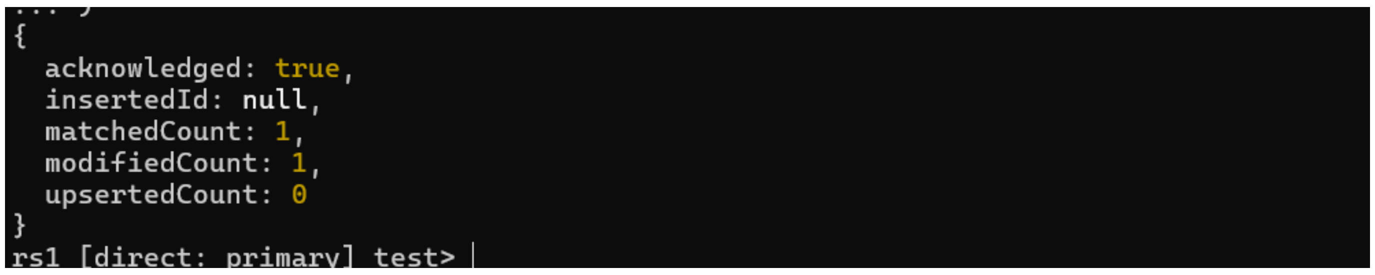
```
rs1 [direct: primary] test> db.teacher.find({_id: ObjectId('683eeb28a4ae1e65596c4bd1')})
[
  {
    _id: ObjectId('683eeb28a4ae1e65596c4bd1'),
    t_id: 13,
    t_name: 'Noah',
    t_mail: 'noah@mongo.xx',
    t_postalcode: 5000,
    t_dob: ISODate('1999-01-20T00:00:00.000Z'),
    t_gender: 'm',
    t_education: 'Bachelor',
    t_counter: 0
  }
]
```

the write was replicated on secondaries as well.

3. **Run a different update on the document with write concern set to 1.**

```
db.teacher.updateOne(
  { t_id: 13 },
  { $set: { t_mail: "noah@mongoDB.xx" } },
  { writeConcern: { w: 1 } }
)
```

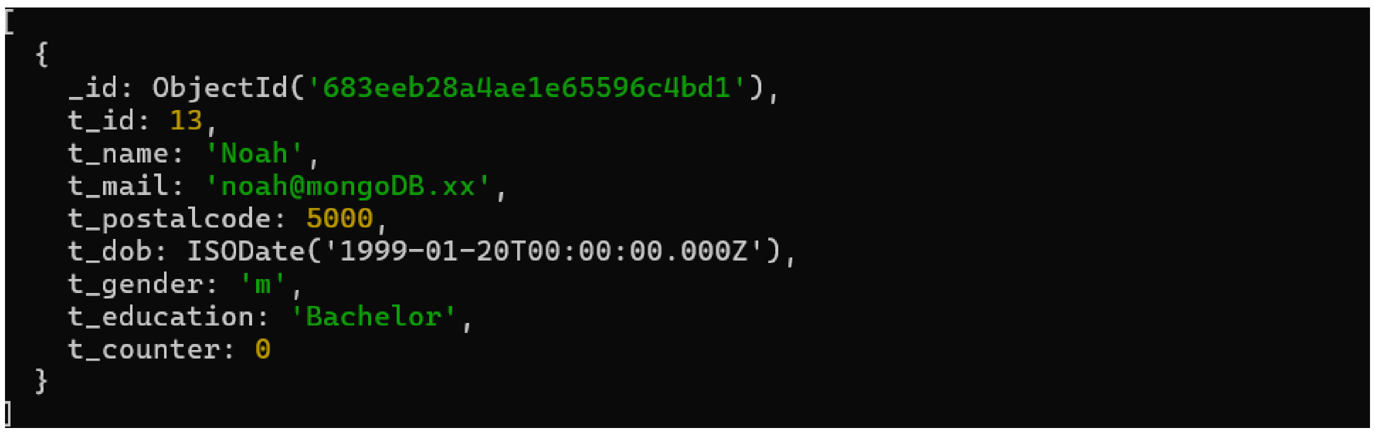
Result:



```
... }
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
rs1 [direct: primary] test>
```

w:1 → Acknowledge only when written to primary

this time we get back the acknowledged: true, and by calling `db.teacher.find({_id: ObjectId('683eeb28a4ae1e65596c4bd1')})` we see that the update was successful



```
[
  {
    _id: ObjectId('683eeb28a4ae1e65596c4bd1'),
    t_id: 13,
    t_name: 'Noah',
    t_mail: 'noah@mongoDB.xx',
    t_postalcode: 5000,
    t_dob: ISODate('1999-01-20T00:00:00.000Z'),
    t_gender: 'm',
    t_education: 'Bachelor',
    t_counter: 0
  }
]
```

the update was replicated on the secondaries as well.

IF we wan to delete the teacher(just a note):

```
db.teacher.deleteOne({ _id: ObjectId("683eeb28a4ae1e65596c4bd1") })
```

4. **What is your recommendation in regard to using w:0.**

Avoid `w:0` in most applications.

- MongoDB returns **immediately**, without waiting for the write to be processed.
- we get no confirmation that the operation even reached the database.
- If MongoDB crashes, restarts, or rejects the write due to schema/validation issues, we will never know.

Use it only when:

- Performance is more critical than durability.
- You can tolerate possible data loss.
- You handle success/failure some other way.

Use at least `w:1` (default) for data consistency and reliability.

- MongoDB waits for acknowledgment from the primary node.
- Ensures the primary received and committed the write to memory and journal (if journaling is enabled).
- Client is notified only after success.
- If it fails, the client knows and can retry.

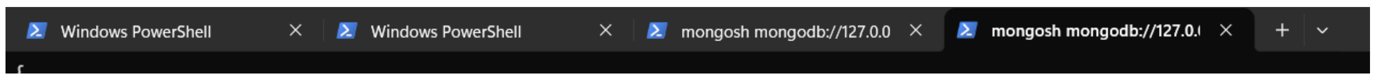
This makes `w:1` more consistent because:

- The client knows the data is actually stored on the primary.
- Errors (e.g., crash, disk full, failover) are detected by the client.

---

1. **Take a secondary (CMD 3 and CMD 6) down (Windows: CTRL + C).**

   we are left with only `port: 27018` and `port: 27019.`

2. **Insert a document with write concern set to 3 into your collection in db test in replica set. What does Mongodb return? What is the result of the write operation?**

   ```
   db.teacher.insertOne({
     "t_id": 14,
     "t_name": "snow",
     "t_mail": "snow@galopp.xx",
     "t_postalcode": 5000,
     "t_dob": new Date("1999-01-20"),
     "t_gender": "f",
     "t_education": "Bachelor",
     "t_counter": 0
   },
       { writeConcern: { w: 3, wtimeOut: 3000}}
   )
   ```

   Mongodb does not return anything it is stuck indefinitely (or if we pass on `wtimeOut`, until the `wtimeOut: 3000` runs out), because it is waiting for something that cannot happen:

   - `w:3` means MongoDB will wait for 3 nodes to acknowledge the write.
   - With only 2 nodes (1 primary and 1 secondary) online, it waits indefinitely for the 3rd node, which never responds.

3. **What is your recommendation in regard to using w:3 in a 3-node-replica set?**

   Using `w:3` in a 3-node replica set means MongoDB will wait for all three nodes (1 primary + 2 secondaries) to acknowledge the write before confirming success.
   we should use
   `w: 3`:

   - When maximum data durability is critical (e.g. financial transactions, auditing logs).
   - When you cannot afford to lose a write even if the primary crashes immediately after.
   - When your application can tolerate higher latency in exchange for stronger consistency.

   Cons:

   - Availability Risk: If even one node is down or slow, the write fails with a `WriteConcernError`, even though it may have succeeded on the primary.
   - Performance Cost: Slower writes due to network/replication delay across all secondaries.
   - Fragility: During routine maintenance or network hiccups, the application may experience write failures.

   Avoid using `w:3` unless the system strictly requires full replica acknowledgment. Use `w:majority` instead for a balance of durability, availability, and performance.

---

## MongoDB Oplog

Does MongoDB use a statement-based replication?

1. **Read Kleppmann, chapter 5, Implementation of Replication Logs**

   ⭐Ok⭐

2. **Explain what – according to Kleppmann – a statement-based replication log is.**

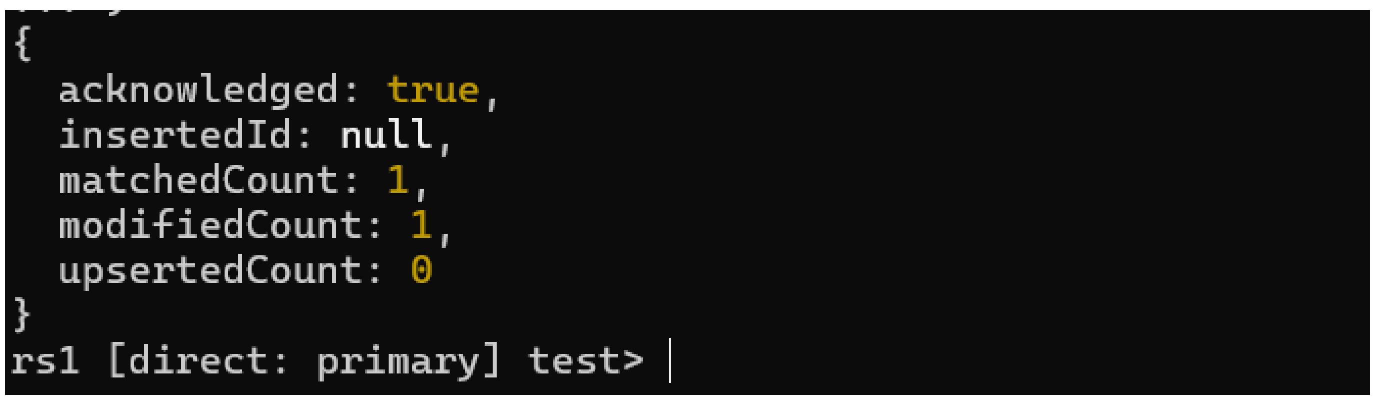
   According to Kleppmann a statement-based replication logs record every insert, update and delete statements, which are forwarded to followers and each follower parses and executed that SQL statement as if it had been received from a client.

3. **Run this update command on one of your document:**

   `db.teacher.updateOne({ t_name: "................" }, { $set: {last_access: Date.now()}})`

```
db.teacher.updateOne(
  { t_id: 13 },
  { $set: { t_mail: "noah@kleppmann.xx" } },
  { writeConcern: { w: 1 } }
)
```

   Result:



```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
rs1 [direct: primary] test>
```

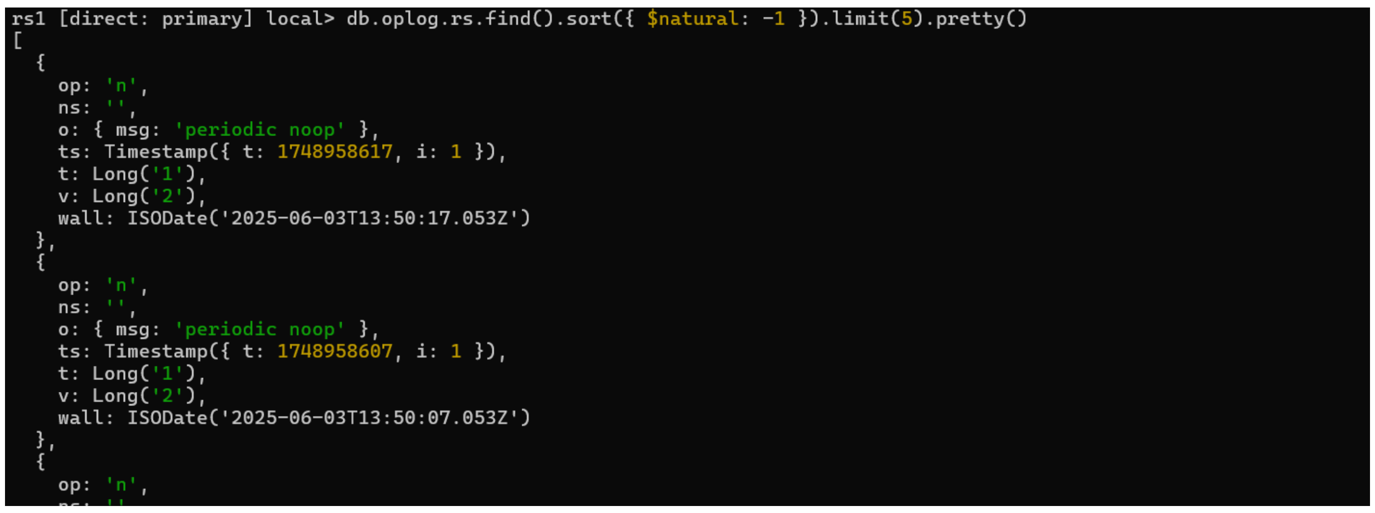
4. **Switch to db local, open the oplog and check what MongoDB stores in the oplog for inserts and updates.**

```
use local
#inspect the recent 5 operations.
db.oplog.rs.find().sort({ $natural: -1 }).limit(5).pretty()
```

   Or specify commands (*i=insert, u=update, d=delete, c=command*):

```
#Exclude no-op entries and show real operations
db.oplog.rs.find({op: { $in: ["i", "u", "d", "c"] }}).sort({ $natural: -1 }).limit(5).pretty()
```

   Result:



```
rs1 [direct: primary] local> db.oplog.rs.find().sort({ $natural: -1 }).limit(5).pretty()
[
  {
    op: 'n',
    ns: '',
    o: { msg: 'periodic noop' },
    ts: Timestamp({ t: 1748958617, i: 1 }),
    t: Long('1'),
    v: Long('2'),
    wall: ISODate('2025-06-03T13:50:17.053Z')
  },
  {
    op: 'n',
    ns: '',
    o: { msg: 'periodic noop' },
    ts: Timestamp({ t: 1748958607, i: 1 }),
    t: Long('1'),
    v: Long('2'),
    wall: ISODate('2025-06-03T13:50:07.053Z')
  },
  {
    op: 'n',
```

5. **Does MogoDB use statement-based replication?**

   No, MongoDB uses operation-based (logical) replication, not statement-based.

   The replication log (oplog) contains the exact changes made to documents, not the original statements or JS expressions.

   This approach:

   - Ensures determinism across replicas,

- Avoids problems like `NOW()` or `Math.random()` differing on each node,
- Allows safe, consistent replication.

## Replication Lag

Asynchronous replication may lead to replication lag.

- **What problems does replication lag cause?**

| Effect | Description |
|---|---|
| **Stale Reads** | If you read from a secondary that hasn't caught up, you may see old data. |
| **Inconsistent Data Views** | A client might read different versions of data depending on which node it contacts. |
| **Testing and Monitoring Issues** | Reads from secondaries might fail to reflect recent writes → misleading test results. |

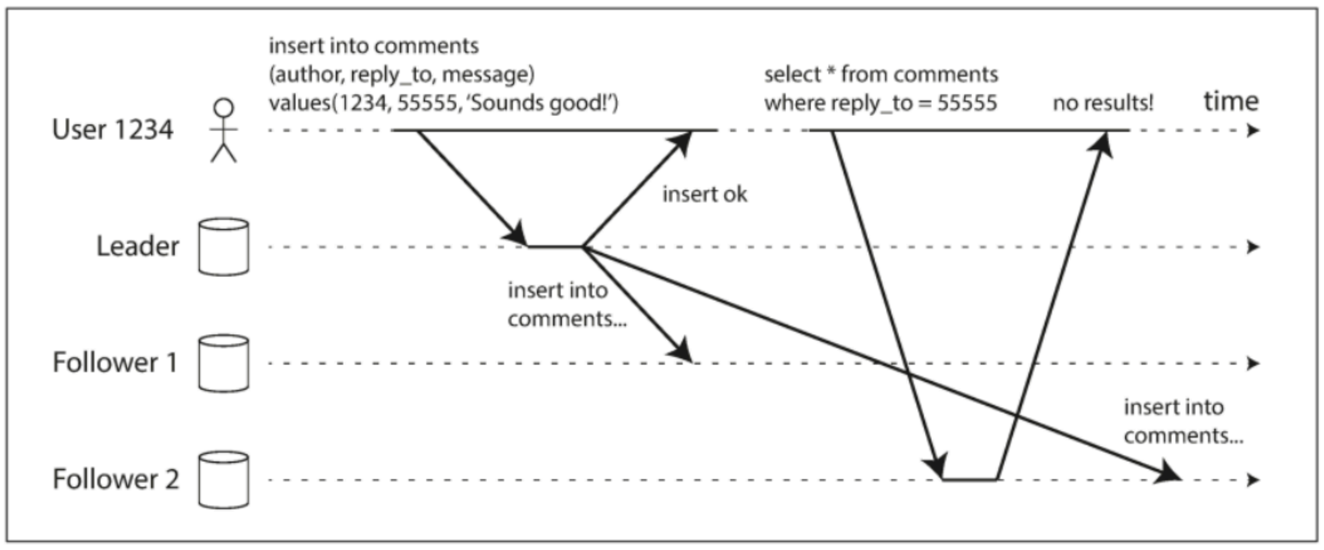
The more secondaries lag behind, the more likely that read operations return stale data.

- **What measures can you take to decrease replication lag?**

| Strategy | Impact |
|---|---|
| Change read preference | `readPreference: "primary"`, Guaranteed up-to-date data, No risk of stale reads, Best choice when consistency matters |
| Disable chaining | Avoid indirect lag propagation |
| Increase oplog size | Prevent resyncs due to lag |
| Avoid blocking ops | Let secondaries apply changes promptly |
| Control write rate | Let replication keep up |
| Use flow control | Automatic lag-based write regulation |

*Attention: Only list measures that you can take – not measures out of your control (like better network bandwidth or such)*

## Issues Resulting from Replication Lag



**What write concern is used here?**

`w:1` since the user only waits for the leaders acknowledgment (OK).

**Describe the replication lag problem displayed in the picture.**

1. User 1234 inserted a comment in the primary
2. the primary sends the write operation to follower 1 and 2
3. follower 1 receives quickly, while follower 2 lags behind
4. User 1234 tries to read the comment from follower 2, but since follower 2 lagged behind it has not received write operation yet

5. User 1234 gets no results.

## MongoDB Nodes Failure Handling

Take down 2 of the 3 nodes.

- **Why is the last running node not promoted to primary?**

  because we have a 3-node replica set, a secondary node needs at least 2 votes to be promoted to a primary node, because we took down other two nodes it only can not have 2 votes.

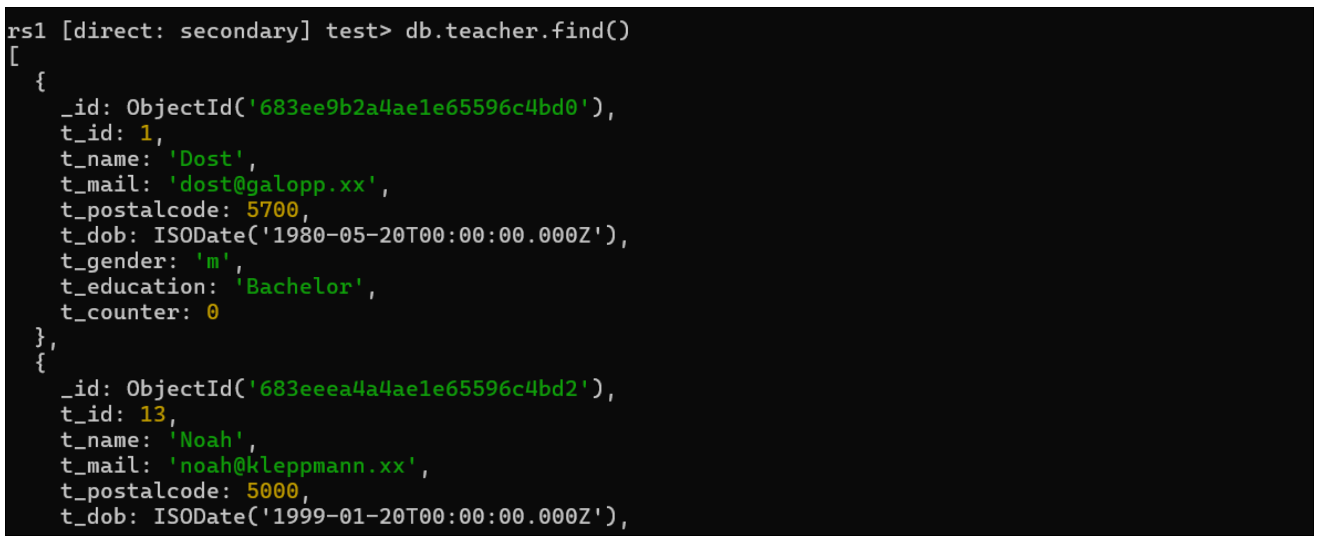
- **Has the system come to a complete halt or does it still accept and execute requests?**

  Since it is a secondary node it cannot accept writes, although it can still accept reads.

  **Show examples on your running node:**

  1. Read

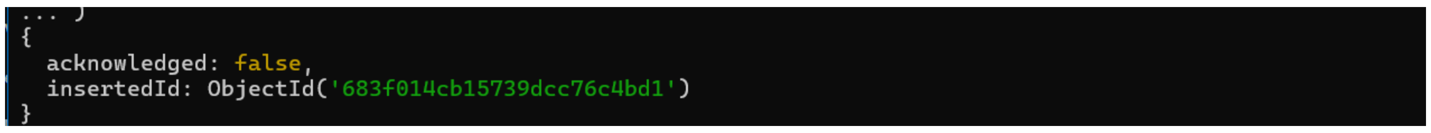
  ```
  db.teacher.find();
  ```



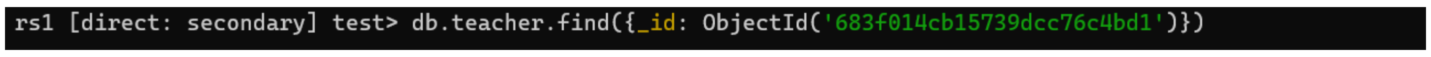
```
rs1 [direct: secondary] test> db.teacher.find()
[
  {
    _id: ObjectId('683ee9b2a4ae1e65596c4bd0'),
    t_id: 1,
    t_name: 'Dost',
    t_mail: 'dost@galopp.xx',
    t_postalcode: 5700,
    t_dob: ISODate('1980-05-20T00:00:00.000Z'),
    t_gender: 'm',
    t_education: 'Bachelor',
    t_counter: 0
  },
  {
    _id: ObjectId('683eeea4a4ae1e65596c4bd2'),
    t_id: 13,
    t_name: 'Noah',
    t_mail: 'noah@kleppmann.xx',
    t_postalcode: 5000,
    t_dob: ISODate('1999-01-20T00:00:00.000Z'),
```

  2. Write

  ```
  db.teacher.insertOne({
    "t_id": 14,
    "t_name": "snow",
    "t_mail": "snow@galopp.xx",
    "t_postalcode": 5000,
    "t_dob": new Date("1999-01-20"),
    "t_gender": "f",
    "t_education": "Bachelor",
    "t_counter": 0
  },
     { writeConcern: { w: 0 }}
  )
  ```

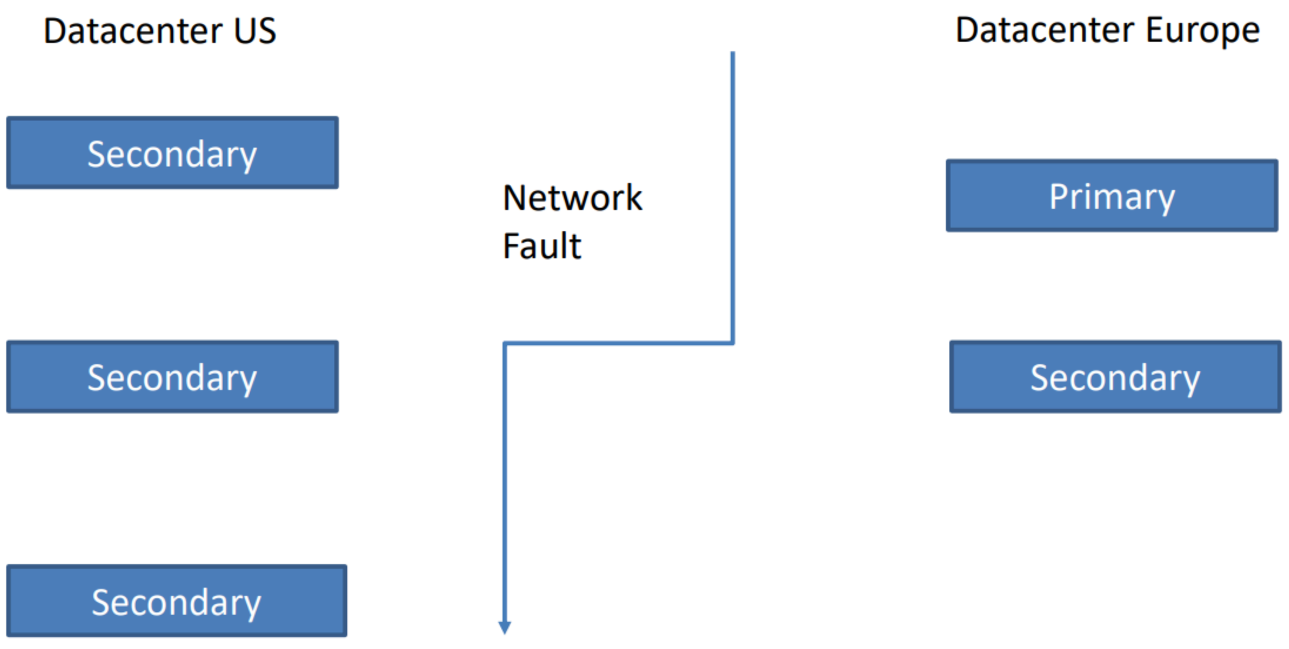


```
... )
{
  acknowledged: false,
  insertedId: ObjectId('683f014cb15739dcc76c4bd1')
}
```



```
rs1 [direct: secondary] test> db.teacher.find({_id: ObjectId('683f014cb15739dcc76c4bd1')})
```

## Split Brain Problem



**<u>What happens when the network connection breaks?</u>**

Both Datacenter US and Europe think that the other is down, US can not see the primary node, so it thinks that the primary node has shut down.

**<u>How do the nodes in Europe react?</u>**

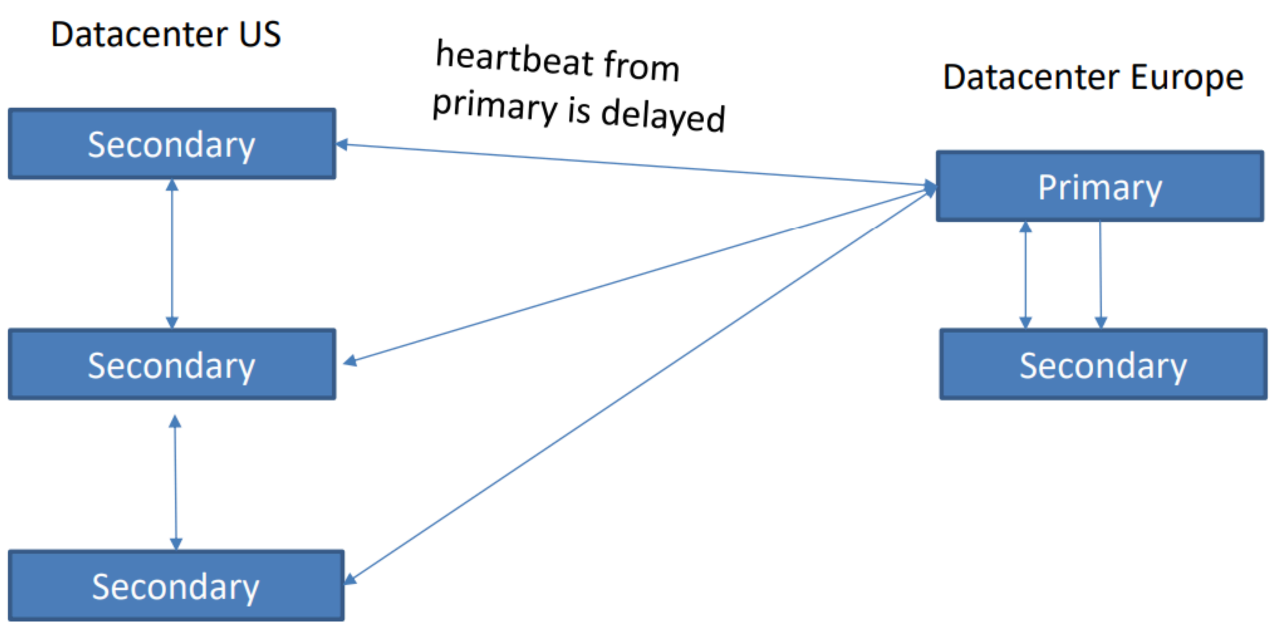
nodes in Europe continue accepting both reads and writes, since they have a primary (but can not replicate to the US secondaries)

**<u>How do the nodes in the US react?</u>**

since they think that the primary has shut down they begin an election for a new primary.

**<u>What happens with the write operations on the Europe primary?</u>**

since US choose a new primary node, if the connection is restored to normal, Europe's primary node is demoted to the secondary and its writes are discarded.



**<u>The US secondaries think that the primary is down because the primary did not respond within the `heartbeatTimeout`. But primary is up and heartbeat simply delayed. What happens?</u>**

Datacenter US secondaries think that the primary is down, because the primary in the Europe did not respond in the time provided by `heartbeatTimeout`.

**<u>How do the nodes in Europe react?</u>**

nodes in Europe continue accepting both reads and writes, since they have a primary (but can not replicate to the US secondaries)

**<u>How do the nodes in the US react?</u>**

since they think that the primary has shut down they begin an election for a new primary.

**Is data loss possible?**

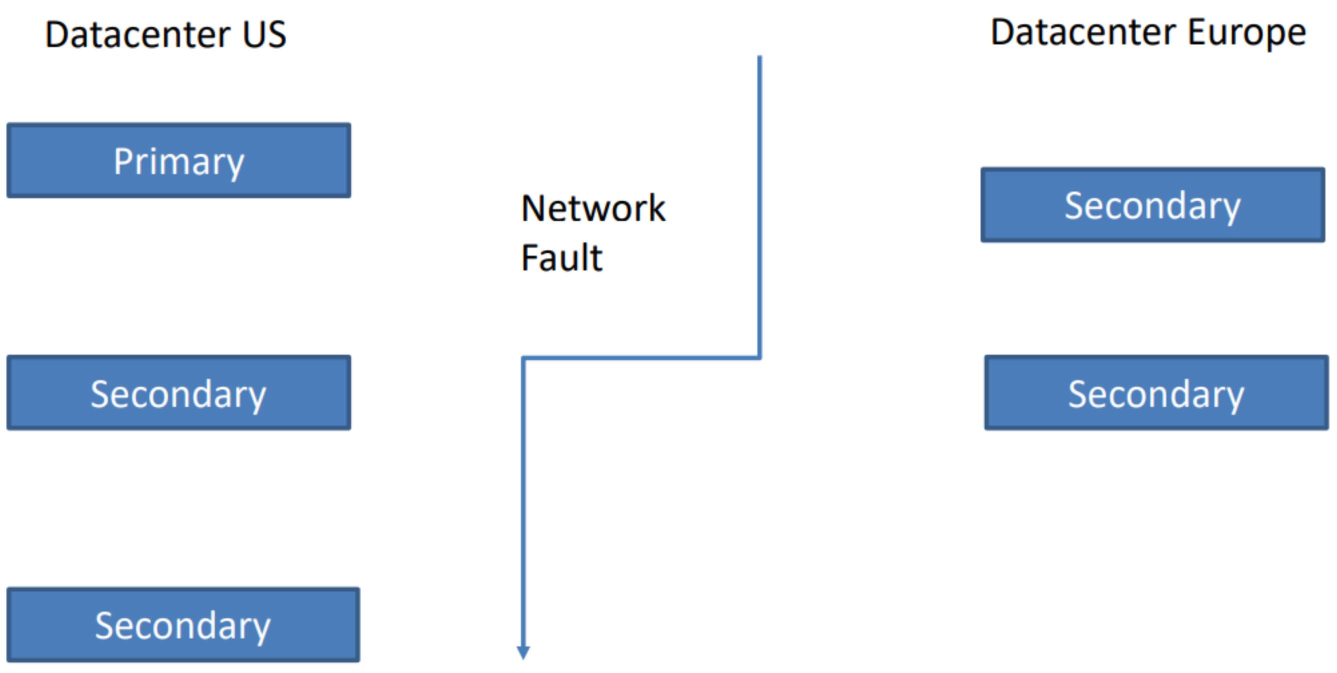
yes, since when the new primary (US) is elected, new writes of the previous primary (Europe) are discarded.

**If heartbeats are delayed often, what measure can you take to prevent premature elections?**

We can increase `heartbeatTimeout` or connect to a better network.

---

## Single Leader Replication



**What happens in this scenario when the network connection breaks?**

Both datacenters US and Europe think that the other is down, they can't access each others nodes.

**How do the nodes in Europe react?**

Nodes in Europe:

- Cannot hold elections.
- Cannot write.
- All nodes stay secondaries.
  - Replica set becomes read-only in Europe.

**How do the nodes in the US react?**

US primary remains primary and continues accepting writes.

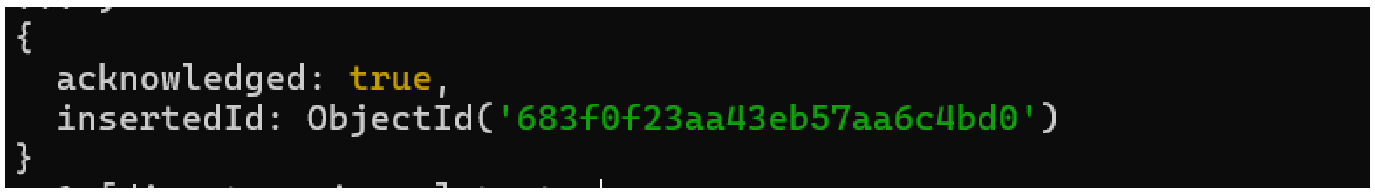
---

## MongoDB

- You have one secondary running.
- Start another mongod instance and connect via shell
- **Insert a document with w: majority.**

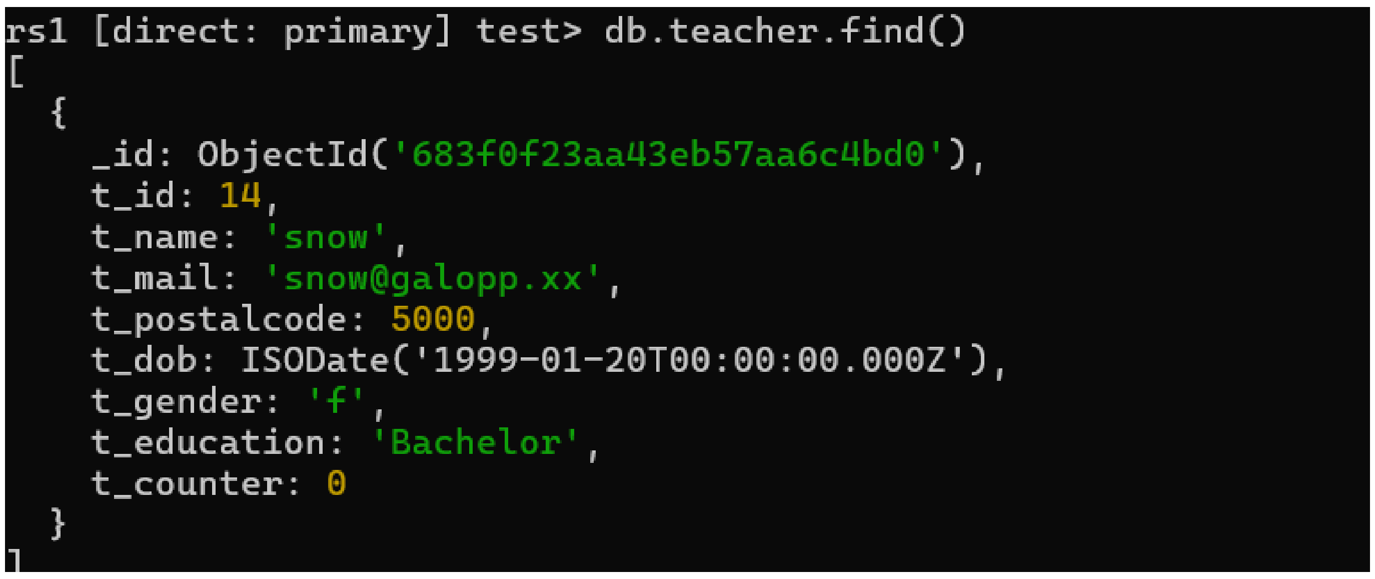
```
db.teacher.insertOne({
  "t_id": 14,
  "t_name": "snow",
  "t_mail": "snow@galopp.xx",
  "t_postalcode": 5000,
  "t_dob": new Date("1999-01-20"),
  "t_gender": "f",
  "t_education": "Bachelor",
  "t_counter": 0
},
  { writeConcern: { w: "majority" }}
)
```

Result:



```
{
  acknowledged: true,
  insertedId: ObjectId('683f0f23aa43eb57aa6c4bd0')
}
```

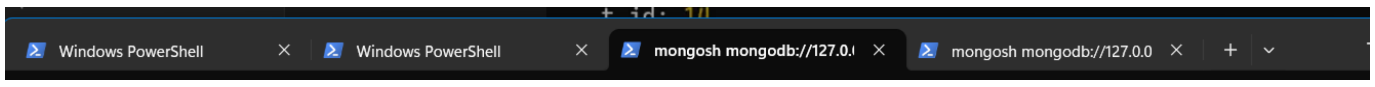
```
db.teacher.find()
```



```
rs1 [direct: primary] test> db.teacher.find()
[
  {
    _id: ObjectId('683f0f23aa43eb57aa6c4bd0'),
    t_id: 14,
    t_name: 'snow',
    t_mail: 'snow@galopp.xx',
    t_postalcode: 5000,
    t_dob: ISODate('1999-01-20T00:00:00.000Z'),
    t_gender: 'f',
    t_education: 'Bachelor',
    t_counter: 0
  }
]
```

- **Stop the node again so that only 1 secondary still runs.**

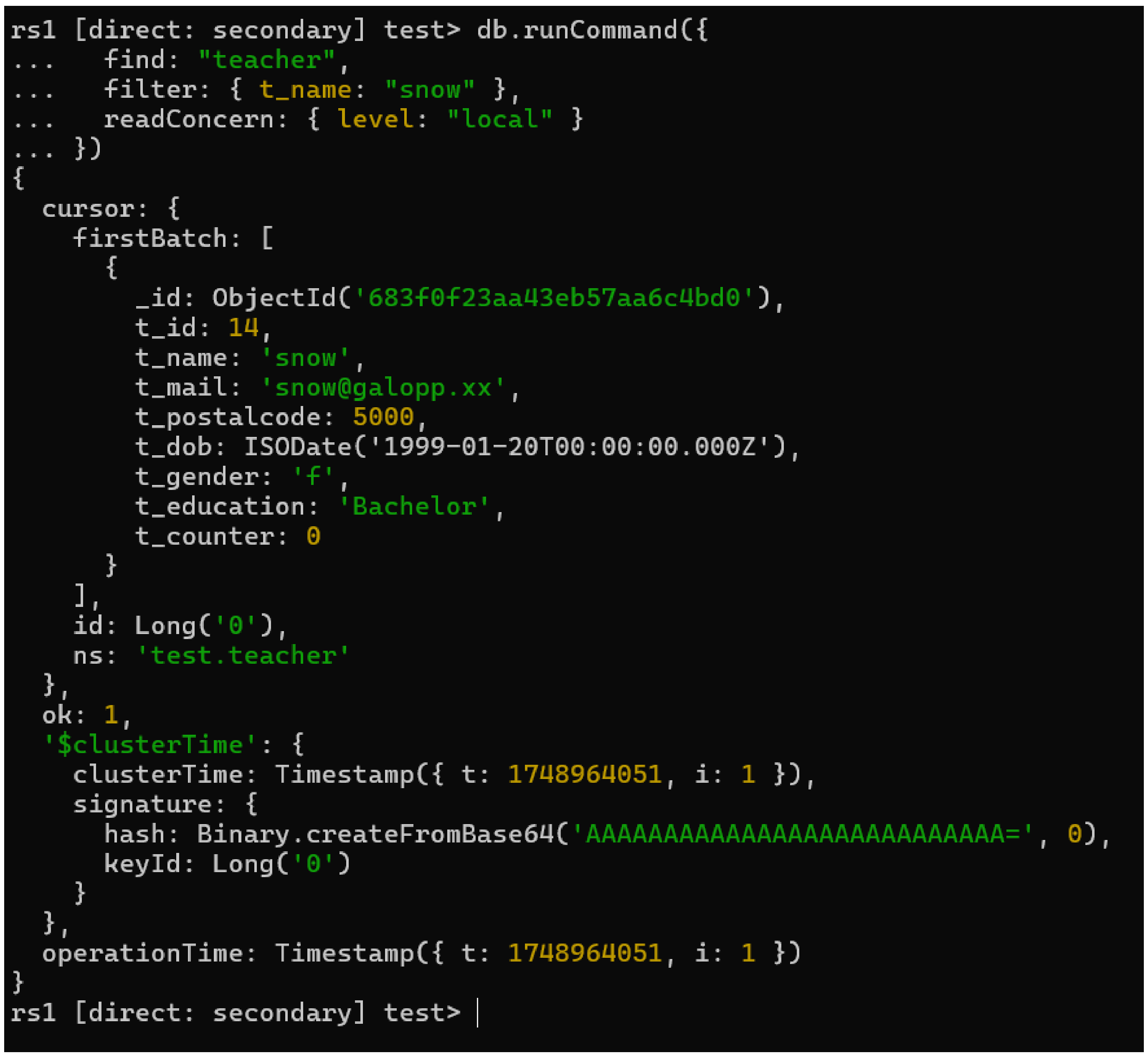
  Killed one secondary (1 secondary and 1 primary left):



- **Read the latest inserted document with 2 different readConcerns:**

```
db.runCommand({
  find: "teacher",
  filter: { t_name: "snow" },
  readConcern: { level: "local" }
})
```

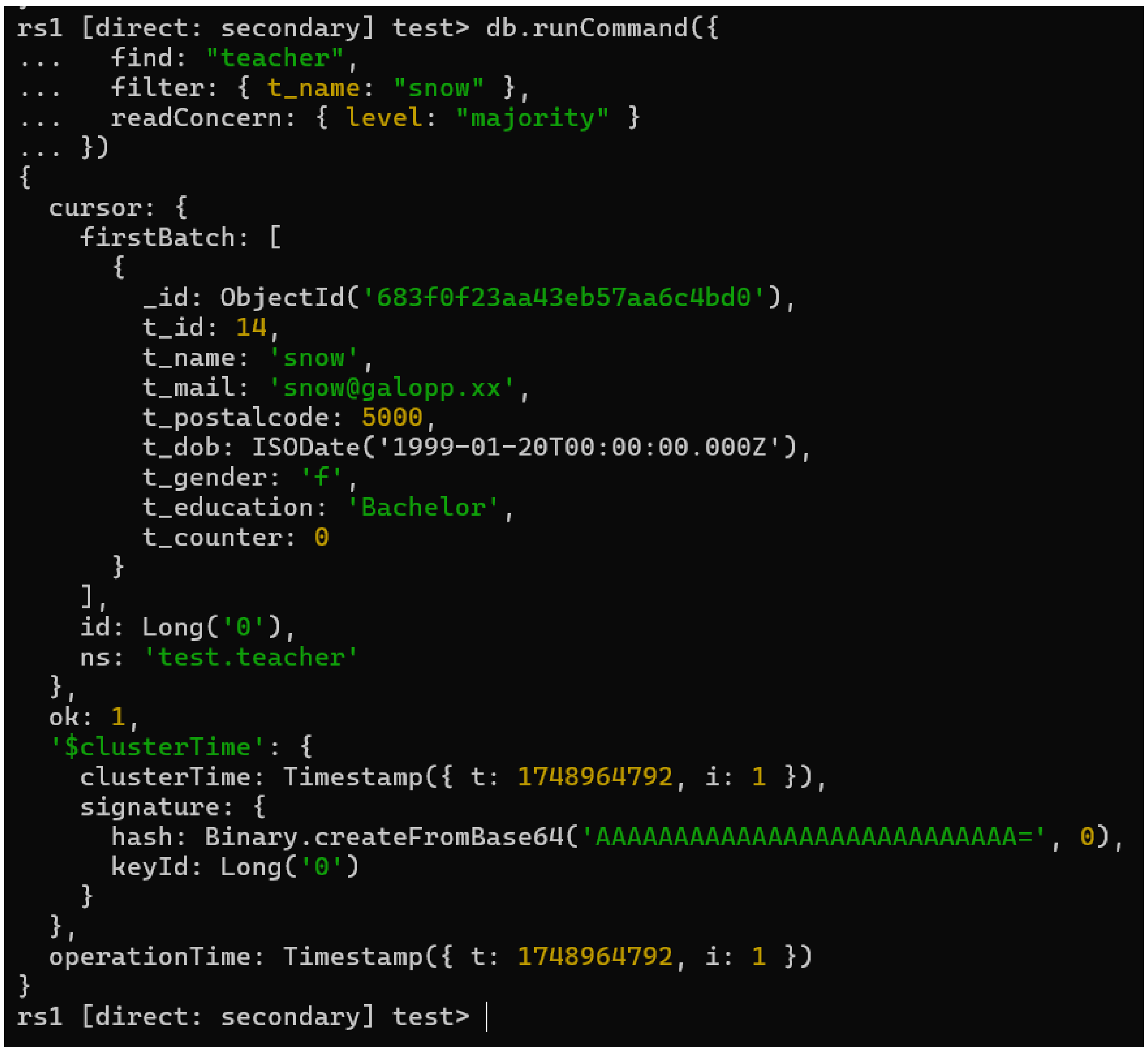
Result:



```
rs1 [direct: secondary] test> db.runCommand({
...   find: "teacher",
...   filter: { t_name: "snow" },
...   readConcern: { level: "local" }
... })
{
  cursor: {
    firstBatch: [
      {
        _id: ObjectId('683f0f23aa43eb57aa6c4bd0'),
        t_id: 14,
        t_name: 'snow',
        t_mail: 'snow@galopp.xx',
        t_postalcode: 5000,
        t_dob: ISODate('1999-01-20T00:00:00.000Z'),
        t_gender: 'f',
        t_education: 'Bachelor',
        t_counter: 0
      }
    ],
    id: Long('0'),
    ns: 'test.teacher'
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1748964051, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1748964051, i: 1 })
}
rs1 [direct: secondary] test>
```

```
db.runCommand({
  find: "teacher",
  filter: { t_name: "snow" },
  readConcern: { level: "majority" }
})
```

Result:



```
rs1 [direct: secondary] test> db.runCommand({
...   find: "teacher",
...   filter: { t_name: "snow" },
...   readConcern: { level: "majority" }
... })
{
  cursor: {
    firstBatch: [
      {
        _id: ObjectId('683f0f23aa43eb57aa6c4bd0'),
        t_id: 14,
        t_name: 'snow',
        t_mail: 'snow@galopp.xx',
        t_postalcode: 5000,
        t_dob: ISODate('1999-01-20T00:00:00.000Z'),
        t_gender: 'f',
        t_education: 'Bachelor',
        t_counter: 0
      }
    ],
    id: Long('0'),
    ns: 'test.teacher'
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1748964792, i: 1 }),
    signature: {
      hash: Binary.createFromBase64('AAAAAAAAAAAAAAAAAAAAAAAAAAA=', 0),
      keyId: Long('0')
    }
  },
  operationTime: Timestamp({ t: 1748964792, i: 1 })
}
rs1 [direct: secondary] test>
```

- **Do both commands run?**

  Both commands succeeded

  `readConcern: "local"` succeeded because:

  - It reads whatever is available on that node — no quorum (minimum number of nodes in a replica set that must agree) or coordination needed.
  - It can return stale or unreplicated data, but it works even if the node is isolated.

  `readConcern: "majority"` succeeded because:

  - The document ( `snow` ) was originally inserted with `writeConcern: { w: "majority" }` :
  - That means the majority of nodes (2 of 3) acknowledged the write before it was considered successful.
  - The secondary i am reading from was one of those nodes or has since caught up with the majority's agreed state.
  - So the secondary can serve that data with `readConcern: "majority"` .

---

## What is the difference between write concern, read concern and read preference?

| Feature | Affects | Controls | Key Options |
|---|---|---|---|
| **Write Concern** | Writes | How many nodes must acknowledge | `w: 0` , `w: 1` , `w: "majority"` |
| **Read Concern** | Reads | What level of data consistency you require | `"local"` , `"majority"` , `"linearizable"` |
| **Read Preference** | Reads | **Which** node to read from | `"primary"` , `"secondary"` , `"nearest"` , etc. |