Lecture 10-11

Type	Lecture

Distributed Databases

Replication: Keeps copies of the same data on multiple nodes.

- **Purpose:** Availability, durability, backup no loss of data, fault tolerance.
- **Challenge:** Synchronization (to keep the copies in sync), consistency between replicas.

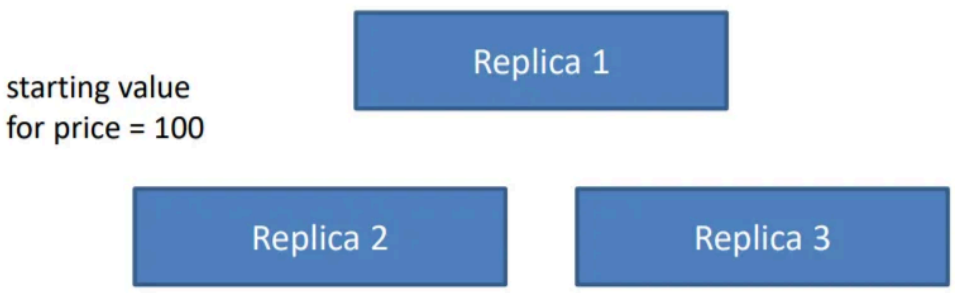
Partitioning (Sharding): Divides data across multiple nodes.

- **Purpose:** Scale horizontally (each node handles a part of the dataset, which makes it able to tolerate high volume of data).
- **Challenge:** Querying across shards; consistency (distribute workloads even over partitions).

Replication

```
--T1
db.price.updateMany({},
{$inc: {price + 10}})
```

```
--T2
db.price.updateMany({},
{$mul: {price * 1.2}})
```



We assume that write command1 and write command 2 run concurrently.
command1 writes to rep1 {$inc: {100 + 10}} → price on rep1: 110
command2 writes to rep2 {$mul: {100* 1.2}} → price on rep2: 120

Replica 3:

now replicas 1 and 2 need to replicate with replica 3, but since they have different values they are out of sync.

If there is no central authority (like a primary) to order these writes, each replica can apply different updates in different orders. This leads to inconsistency between replicas — this is known as replica divergence.

what value replica 3 will store depends on which update it receives first or at all, which is undefined in a decentralized, uncoordinated system.

Synchronization Issues in Replicated Databases: Riak Example

```
Bucket bucket = connection
    .createBucket(bucketName)
    .allow_mult(true / false)
    .n_Val(numberOfReplicationCopies, default 3)
    .last_write_wins(default false)
    .w(numberOfNodesToRespondToWrite)
    .r(numberOfNodesToRespondToRead)
    .execute();
```

allow_mult(true / false):

- When set to `true`, multiple versions (siblings) of a value can coexist.
- This happens when concurrent writes to the same key are accepted by different nodes.
- Why? In eventually consistent systems, different nodes may accept writes while disconnected and later reconcile.
- Result: Multiple versions exist → Application must manually resolve them.

This is powerful but shifts the burden of conflict resolution to the application.

Sibling Conflicts (When allow_mult = true):

If two clients write different values for the same key concurrently:

- Riak stores both versions as siblings.
- The next reader must handle conflicting versions.

n_Val – Number of Replicas:

Sets the number of replica copies for each key. Default is 3. So, every piece of data will exist on 3 different nodes.

w – Write Quorum:

Specifies the minimum number of nodes that must acknowledge a write for it to be considered successful. For example, `w=2` means the write must be accepted by at least 2 out of 3 replicas.

r – Read Quorum:

Specifies the minimum number of nodes that must respond to a read request. Similar to `w`, ensures you're likely getting up-to-date data.

last_write_wins (LWW):

When `true`, Riak automatically resolves conflicts by choosing the value with the latest timestamp. This avoids sibling values, but:

- **Can cause data loss**, because one write may **overwrite another valid concurrent write** just because of timing.
- **Not safe in many use-cases** (e.g., counters, banking balances).

LWW is a **trade-off**:

- Simpler handling → No siblings.
- But you risk losing updates that arrive slightly later.

Therefore, databases prioritizing data consistency (e.g., MongoDB, PostgreSQL) avoid LWW.

Application-Level Conflict Resolution

> "The application will need to decide which value is more correct depending on the use case."

This is not automatic in Riak. Developers must:

- Retrieve all siblings.
- Apply custom merge logic (e.g., choose most recent, merge fields, add values).
- Write back the resolved version.

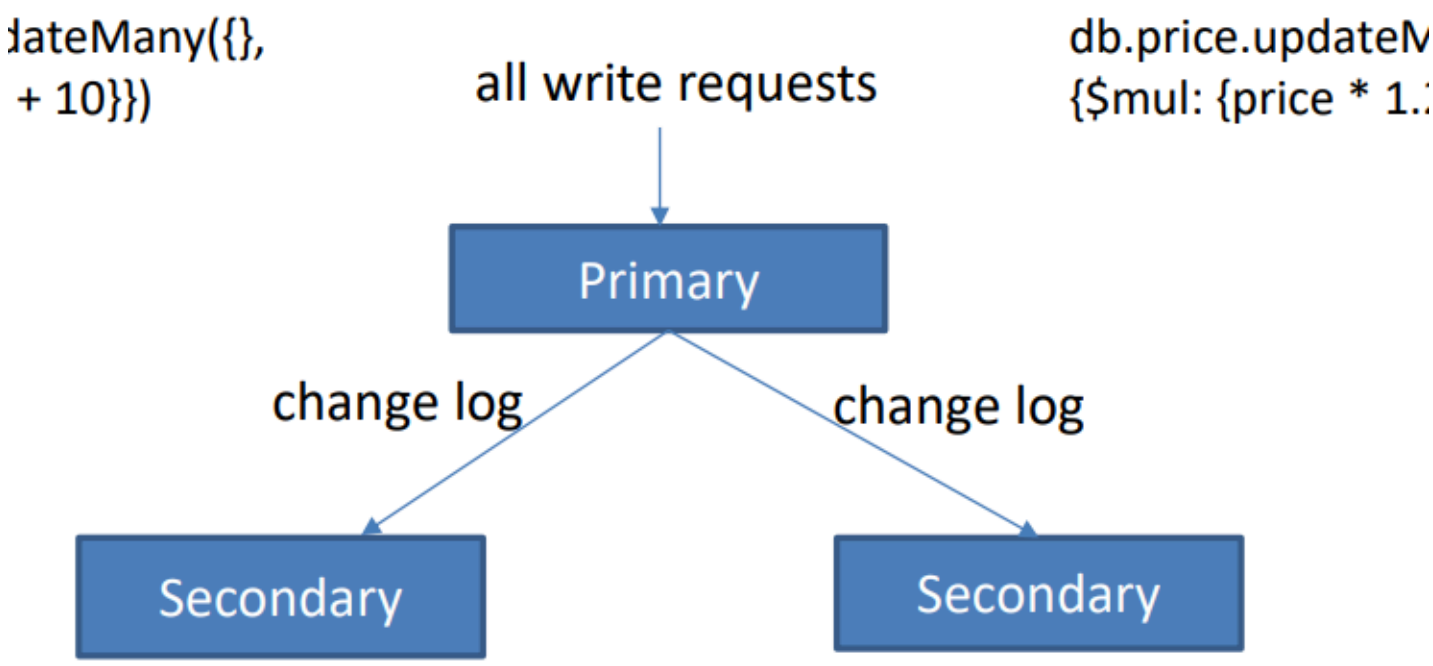
Summary:

Riak uses eventual consistency and allows siblings (multiple values for the same key).

Options:

- `allow_mult = true`: stores conflicting values (requires app-level resolution).
- `last_write_wins = true`: resolves conflict by timestamp (can lead to data loss).

Single Leader Replication



Single Leader Replication (also called **Primary-Secondary Replication**) is a replication model where:

- One node (the Primary) handles all write operations.
- Other nodes (Secondaries) receive updates from the Primary.
- The Primary logs each write and forwards it in order to the secondaries, which apply the same operations.

Process:

1. All Writes Go to the Primary

- Applications must **always write to the Primary** node.
- This centralization ensures that **writes are processed in a single, linear order**, avoiding race conditions and conflicts.

If multiple nodes accepted writes, they might conflict — like in Riak or Cassandra without special handling. Having a single writer prevents this.

2. Primary Sends a Change Log

- After writing data, the Primary creates a **replication log (also called oplog in MongoDB (this is the only replication method in relational databases))**.
- This log records each operation in **the exact order it was applied**.
- Secondaries **read the oplog** and apply changes to replicate the data.

If secondaries applied operations in different order, they could diverge. Ordered logs preserve data consistency across replicas.

3. Secondaries Apply Changes from the Log

- Secondaries are **read-only** in this model.
- They replicate changes **asynchronously**, meaning there can be **replication lag**.
- If Primary fails, one of the secondaries can be **promoted to Primary** through an **election process**.

Riak vs mongodb

Aspect	Riak (Eventual Consistency)	MongoDB/Postgres (Strong Consistency)
Write handling	Multiple versions (siblings)	Serialized via primary
Conflict resolution	Application responsibility	Handled by replication protocol
Concurrent writes	Stored as siblings	Rejected or queued
LWW option	Available but risky	Not default or recommended
Use case	High-availability, partition-tolerant apps (e.g., sensor data)	Apps needing strong correctness (e.g., banking)

MongoDB Replica Set

A Replica Set is MongoDB's implementation of Single Leader Replication. It is a set of Mongodb server instances that store the same data.

A replica set contains a primary node on which all write operations are performed. In addition, it contains one or more secondary nodes.

```
rs.initiate({
  _id: "rs1", --name of the replica set
  members: [ -- a list of MongoDB instances (nodes), each running on a different port.
    { _id: 0, host: "localhost:27018" },
    { _id: 1, host: "localhost:27019" },
    { _id: 2, host: "localhost:27020" }
  ]
})
```

default port: 27017 - leave that untouched .

Once the replica set is initiated, you can inspect its status with:

```
rs.status()
```

This provides:

- Role of each node (primary/secondary).
- Health of each node.
- Sync information and lag.
- Election status and member states.

Replicas are installed typically on separate servers using the same port, but one may run them on one system using different ports.

MongoDB Replica Set Initialization:

1. create data directories

   Each `mongod` instance (MongoDB server process) needs its own **data folder** to store database files.

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

2. start mongod instances (Replica Set Members)

   Run each instance on a **different port** and bind them to their **respective data directories**, enabling replication by assigning the same `--replSet` name.

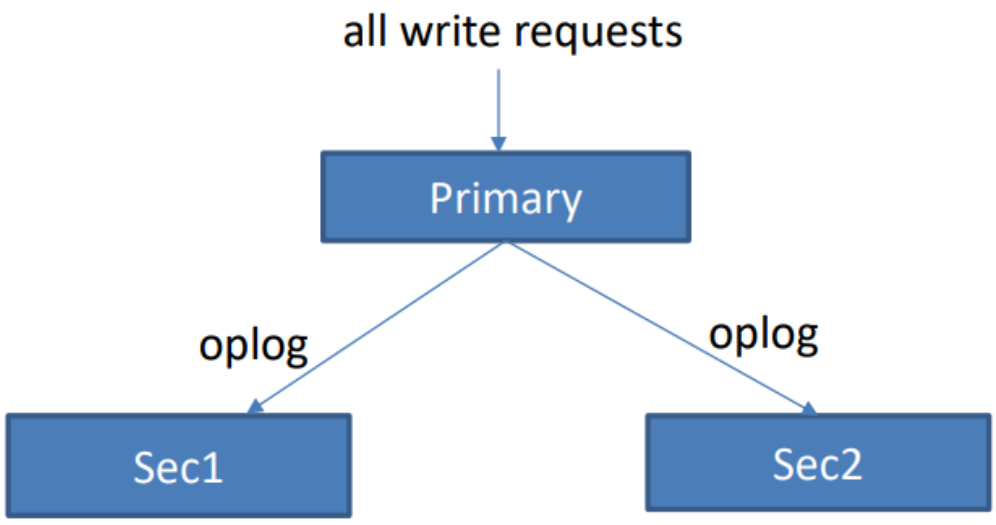
```
mongod --replSet "rs1" --port 27017 --dbpath /data/rs1 --bind_ip localhost
mongod --replSet "rs1" --port 27018 --dbpath /data/rs2 --bind_ip localhost
mongod --replSet "rs1" --port 27019 --dbpath /data/rs3 --bind_ip localhost
```

3. initiate the replica sets

   Connect to one instance (usually to the default one on 27017), then run:

```
rs.initiate({
  _id: "rs1",
  members: [
    { _id: 0, host: "localhost:27017" },
    { _id: 1, host: "localhost:27018" },
    { _id: 2, host: "localhost:27019" }
  ]
})
```

MongoDB Single Leader Replication



- Chained replication is possible: primary → secondary1 → secondary2.
- Writes only go to the primary, secondaries replicate via the oplog.
- Primaries can be denoted, secondaries can be promoted at any time, mongodb does this by itself.

Issue: *Chained replication may add replication lag.*

---

rs.status() vs rs.status().ok

rs.status()

example output:

```
{
  set: "rs1",
  date: ISODate("2025-05-29T21:02:53Z"),
  myState: 1,
  members: [
    {
      _id: 0,
      name: "localhost:27017",
      stateStr: "PRIMARY",
      health: 1,
      uptime: 6032,
      ...
```

```
    },
    ...
  ],
  ok: 1
}
```

Field	Description
`set`	Name of the replica set (e.g., `"rs1"`).
`date`	Timestamp of when the status was reported.
`myState`	State code of the current node (e.g., `1 = PRIMARY`, `2 = SECONDARY`).
`term`	Current election term (used in primary elections).
`heartbeatIntervalMillis`	Interval (in ms) between heartbeat messages.
`members`	List of all replica set members and their individual status.
`ok`	`1` if the command succeeded, `0` if it failed.

Inside the members array:

Field	Description
`_id`	ID of the replica set member.
`name`	Host and port of the node (e.g., `"localhost:27017"`).
`state`	Numerical state code (see below).
`stateStr`	Human-readable state (e.g., `"PRIMARY"`, `"SECONDARY"`).
`uptime`	Time (in seconds) this member has been up.
`optime`	Latest operation time (used for replication sync).
`optimeDate`	Human-readable version of `optime`.
`lastHeartbeat`	Last successful heartbeat received from this node.
`lastHeartbeatRecv`	Last heartbeat this node received from the other.
`pingMs`	Average ping latency to this node in milliseconds.
`syncSourceHost`	Hostname of the node this member is syncing from (only on secondaries).
`health`	`1` = healthy, `0` = unreachable.
`configVersion`	Version number of the replica set configuration.
`self`	`true` if this document represents the current node.

rs.status().ok

This accesses only the `ok` field from the result of `rs.status()`.

- If it returns `1`: the replica set is healthy and the command succeeded.
- If it returns `0`: there was a problem (e.g., the current node can't contact others or is not in a replica set).

MongoDB RS Synchronization

All Write Operations Are Logged in the Oplog:

- When you perform any write operation (insert, update, delete) on the primary, MongoDB records this change in a special internal collection:

  ```
  local.oplog.rs
  ```

- This collection is called the **oplog** (short for **operations log**).
- It is a **capped collection**, meaning it has a fixed size and automatically removes old entries when full.

The Oplog Contains Entries for Each Operation:

Field	Meaning
`ts`	Timestamp of the operation
`op`	Operation type (`i`=insert, `u`=update, `d`=delete, `n`=no-op)
`ns`	Namespace (e.g., `test.teacher`) — combination of database and collection
`o`	The actual operation content (e.g., the update applied)
`o2`	Optional field — criteria used to find the document (used in updates)

Secondaries Pull and Apply Oplog Entries

- Secondaries continuously read from the primary's oplog.
- They apply each operation in the same order to maintain consistency.
- This process is **asynchronous**, meaning there may be a slight delay (replication lag).

To inspect the oplog directly:

```
use local
db.oplog.rs.find({ ns: "test.teacher" }).sort({ $natural: -1 }).limit(5)
```

- `use local`: Switch to the internal database where the oplog is stored.
- `db.oplog.rs`: The capped collection holding oplog entries.
- `sort({ $natural: -1 })`: Shows the most recent operations first.
- `limit(5)`: Only show the last 5 operations.

Capped Collection = Collection with fixed size limit

A **capped collection** is a **special type of MongoDB collection** that has:

- A **fixed maximum size** (in bytes), and optionally
- A **maximum number of documents** it can hold.

Once this limit is reached, MongoDB **automatically overwrites the oldest documents** in the collection with new ones.

Feature	Description
Fixed Size	You define a byte-size limit (`size`) and optionally a max document count (`max`).
Insertion Order	Documents are stored in the **order inserted** and can be read that way.
Automatic Overwrite	When the collection is full, **oldest documents are removed** to make space.
No Document Deletion	You **cannot delete individual documents** — only via overwrite.
No Growing Documents	You cannot update a document in a way that **increases its size**.
Efficient Storage	No need for index maintenance or free space reuse logic.

Capped collections are ideal for **logs**, **metrics**, **audit trails**, and **replication** because:

You only care about **recent data,** You don't need to manually delete old records, You want **high write throughput**.

MongoDB's **oplog.rs** (used for replication in replica sets) is implemented as a capped collection:

- Ensures bounded disk usage.
- Automatically rotates old operations out.
- Allows secondaries to replicate in natural order of operations.

How to Create a Capped Collection:

```
db.createCollection("log", {
  capped: true,
```

```
  size: 5242880,   // 5 MB
  max: 5000        // optional: max 5000 documents
})
```

You can verify if a collection is capped with:

```
db.log.isCapped()  // returns true or false
```

MongoDB RS Synchronization

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

this is a dynamic (non-deterministic) update, because each time it runs, it produces a different value.

MongoDB replication works by copying **exact operations** from the primary's oplog to secondaries.

For this to work reliably:

- The operations must be idempotent — meaning the same result is achieved even if the operation is applied more than once.

If the operation includes a function like `Date.now()`, it evaluates differently on the primary and secondaries.

MongoDB avoids this issue

MongoDB does not store the JavaScript expression `Date.now()` in the oplog.

Instead, it evaluates `Date.now()` on the primary to get a timestamp (e.g., `ISODate("2025-05-29T21:20:45.123Z")`), and then it stores that literal value in the oplog.

Example oplog entry:

```
{
  ts: Timestamp(...),
  op: "u",
  ns: "test.teacher",
  o2: { t_name: "Dost" },
  o: { $set: { last_access: ISODate("2025-05-29T21:20:45.123Z") } }
}
```

So when secondaries apply this update, they use the same exact value, maintaining consistency.

Check:

```
use local
db.oplog.rs.find({ ns: "test.teacher" }).sort({ $natural: -1 }).limit(1)
```

`last_access` value is already resolved to a fixed date — not `Date.now()`.

MongoDB Replication OpLog

An operation is idempotent if applying it once or multiple times has the same effect.

- If `f(x) = f(f(x))`, then `f` is idempotent.
- Example: `f(x) = min(x, 10)` is idempotent: `min(min(x, 10), 10) = min(x, 10)`

MongoDB Write Acknowledgements

```
db.teacher.updateOne(
  { name: "Teufel" },
  { $set: { remark: "You will be successful when working with me." } }
)
```

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

acknowledged: true

- This means **the primary node has accepted the write**, and MongoDB has confirmed the command execution to the client.
- **However**: In a replica set, this only means the primary executed the command — **it does not mean secondaries have applied it yet**.

In a single-node MongoDB, a write is successful if "acknowledged: true" is returned.

BUT, In a replica set, this only confirms the write reached the primary, not whether it's been replicated. That's where write concern comes in.

MongoDB Write Concern

In MongoDB, **write concern** specifies **how many replica set members must confirm** a write before the client considers it successful.

It determines the **level of guarantee** for:

- **Durability**: How likely it is that a write will survive a failure.
- **Replication**: Whether and how much the data has spread to other nodes.

Write Concern	Meaning	Durability	Replication
`w: 0`	Don't wait for any acknowledgment	Not durable	No replication guarantee
`w: 1`	Acknowledge after primary writes	Not durable	Secondaries may lag
`w: 2`	Primary + 1 secondary	Some durability	Replicated to at least 2 nodes
`w: "majority"`	Majority of nodes acknowledge	Durable	Sufficient for failover
`w: n`	All n nodes must acknowledge	Very durable	Fully replicated

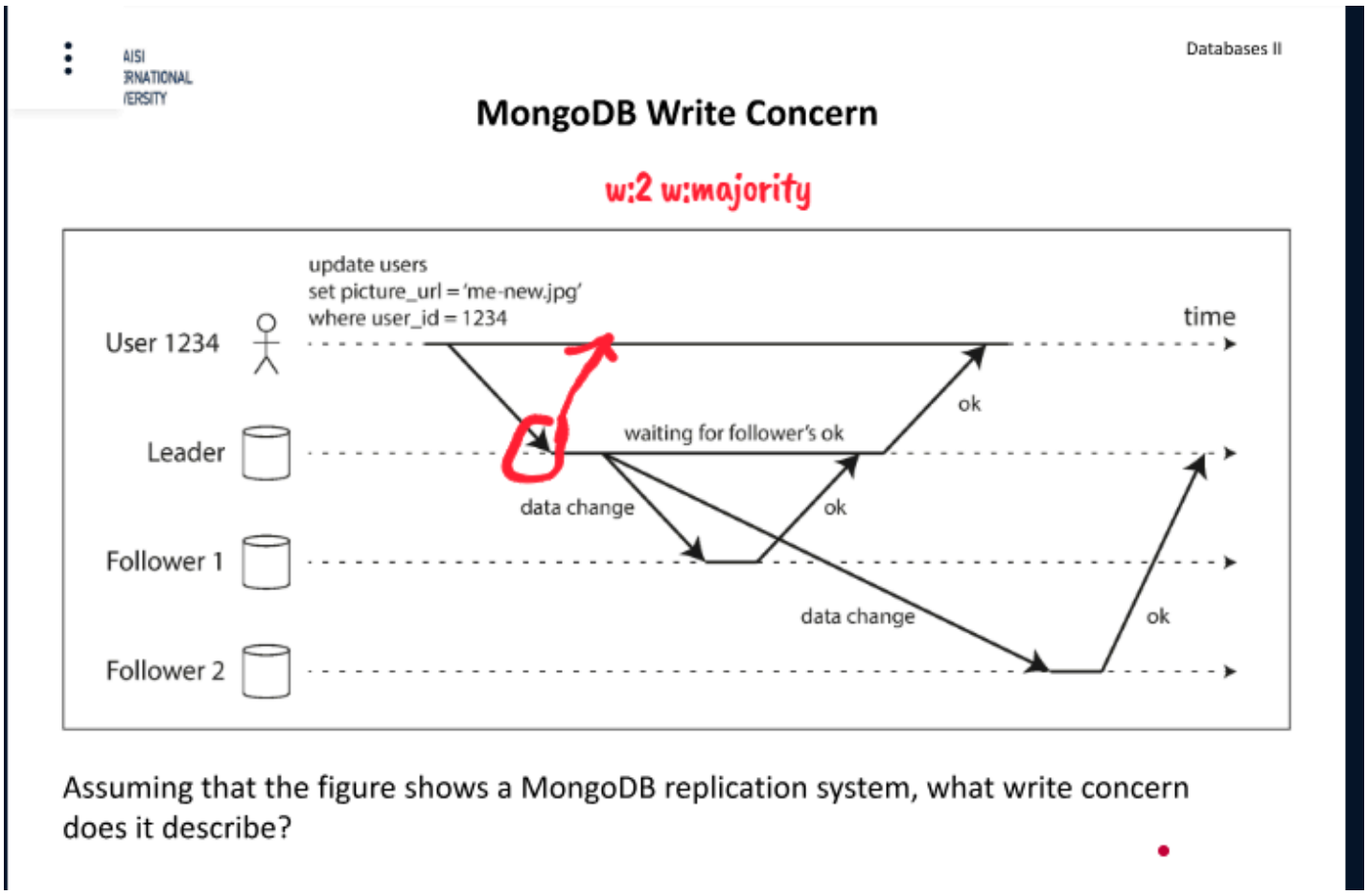
> "Rollback possible"
>
> → With `w: 0` or `w:1`, if the **primary fails before the operation is replicated**, the write may be **rolled back**.

> "Rollback not possible"
>
> → With `w: "majority"`, the write is **safe** — it's guaranteed to be replicated to a majority, so a new primary will have it.

Example:

```
db.collection.insertOne(
  { name: "Alice" },
```

```
  { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```



write should happen in 2 places.

Write concern can be set at

- query level
- session level (connection)
- database level

Example:

```
// First update
db.teacher.updateOne(
  { t_name: "Dost" },
  { $set: { t_counter: 5 } },
  { writeConcern: { w: 3, wtimeout: 3600 } }
)
```

stop one of the secondary nodes (`Ctrl + C` in its shell).

```
// Second update
db.teacher.updateOne(
  { t_name: "Free" },
  { $set: { t_counter: 15 } },
  { writeConcern: { w: 3, wtimeout: 3600 } }
)
```

MongoDB will only acknowledge the write after all 3 nodes confirm it (primary + both secondaries). This gives the strongest durability — ensures full replication before client sees success. Now, only 2 out of 3 nodes are available. The write will block until `wtimeout` (3.6 seconds). After that, MongoDB will return an error, saying it could not satisfy the write concern.

Strict write concerns (like w:3) are very safe — but can hurt availability. In production, it is often used `w: "majority"` which: Tolerates 1 node failure in a 3-node set. Still ensures that data is safe and won't be rolled back.

MongoDB Failover Strategy

The biggest risk in single-leader systems like MongoDB is that if the primary node fails, the system can no longer accept writes, and possibly becomes unavailable, unless there's an automatic failover mechanism in place.

The system needs to have a failover strategy A **failover strategy** ensures the system can:

1. **Detect failures** (e.g., primary becomes unresponsive).
2. **Automatically promote** a healthy secondary to become the new primary.
3. Resume normal operations **without manual intervention**.

Without a failover strategy:

- A human must detect the issue and intervene.
- Your application might fail during downtime.
- You risk **data loss or inconsistency** if writes are not coordinated properly during transition.

Native Postgres does not come with automatic failover. MongoDB comes with automatic failover.

Failure Detection

In a distributed system like MongoDB's replica set, nodes must:

- Monitor each other to detect if one goes offline, crashes, or becomes unreachable.
- This detection is critical to:
  - Trigger elections (for failover),
  - Avoid routing requests to dead nodes,
  - Maintain consistency.

Desired failure detection properties:

Property	Description
Completeness	All actually failed nodes are eventually recognized as failed by the rest of the system.
Efficiency	Failures are detected quickly — ideally within a few seconds.
Accuracy	The system should not falsely declare a healthy but slow node as failed (false positives should be minimal).

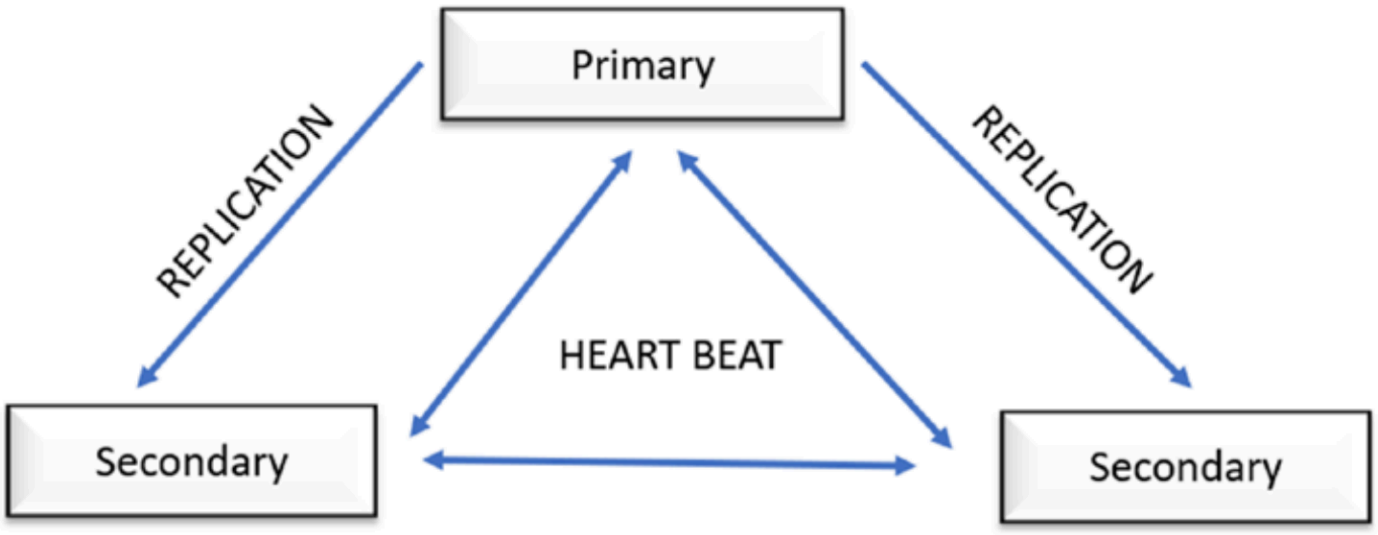
Faster detection = More sensitive to delays = More false positives.

Slower detection = Less responsive system = Higher availability risk.

Heartbeats (a.k.a Ping messages)

- Each MongoDB node sends a small message (heartbeat) to its peers at regular intervals (e.g., every 2 seconds).
- If a node doesn't respond within a configured timeout (e.g., 10 seconds), it's marked as unreachable or failed.

This is how MongoDB decides when to trigger a re-election or stop sending reads/writes to that node.



Replica set members send heartbeats (pings) to each other every two seconds. If a heartbeat does not return within 10 seconds (`heartbeatTimeout`), the other members mark the delinquent member as inaccessible.

Tuning the heartbeatTimeout is possible with the `settings.heartbeatTimeoutSecs` parameter.

MongoDB Failure Detection

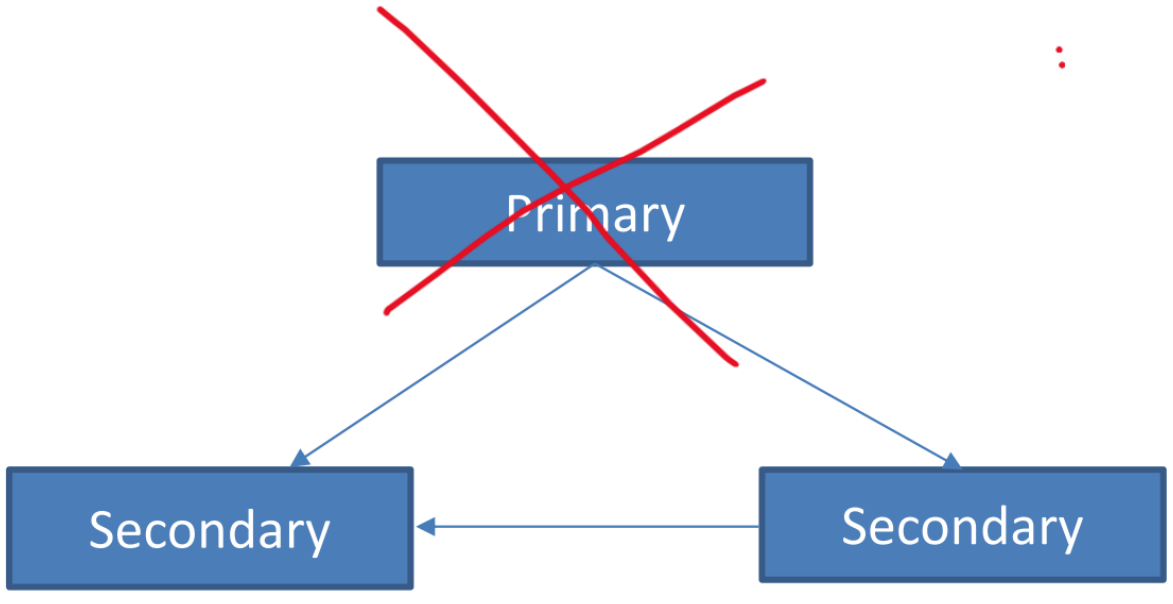
Increasing the heartbeatTimeout from 10 secs to 12 secs increases accuracy, When you increase `heartbeatTimeoutSecs`:

- You give more time for a node to respond to heartbeat pings.
- This reduces the chance of falsely declaring a slow but live node as dead.
- Therefore, it improves accuracy — because your failure detector is less likely to make a mistake.

Timeout Setting	Accuracy	Efficiency (Speed)
Short timeout (e.g., 5s)	❌ Lower accuracy (false positives)	✅ Higher efficiency (fast failover)
Long timeout (e.g., 12s)	✅ Higher accuracy (fewer false positives)	❌ Lower efficiency (slower detection)

MongoDB Automatic Failover

If the primary fails:



- Nodes use heartbeats to check on each other (every 2 seconds).
- If a node(primary) misses responses for `heartbeatTimeoutSecs` (default 10s), it's marked as unreachable.
- An eligible secondary calls for an election.

- The replica set runs an **election algorithm** (Raft-like).
- A majority of voting members must be available for the election to succeed.
- A secondary is promoted to **primary**.
- Clients automatically detect the new primary (via driver) and resume writes.

❗ **While the election is in progress:**

- The replica set is in **read-only mode**.
- No writes are accepted.
- This prevents split-brain (two or more nodes in a distributed system mistakenly believe they are the leader (primary) due to a network partition or failure in communication) and ensures data consistency.

MongoDB Election Process

```
heartbeatIntervalMillis: Long('2000'),
majorityVoteCount: 2,
writeMajorityCount: 2,
votingMembersCount: 3,
writableVotingMembersCount: 3
```

Parameter	Meaning
`heartbeatIntervalMillis: 2000`	Each member sends heartbeat messages every 2 seconds.
`majorityVoteCount: 2`	At least 2 out of 3 nodes must vote to elect a new primary.
`writeMajorityCount: 2`	A write is acknowledged when at least 2 nodes confirm it.
`votingMembersCount: 3`	Total number of members eligible to vote.
`writableVotingMembersCount: 3`	All 3 nodes are able to process writes when healthy.

- The existing primary is marked with a red ❌ → It has failed.
- The two remaining secondaries are still connected via heartbeats.

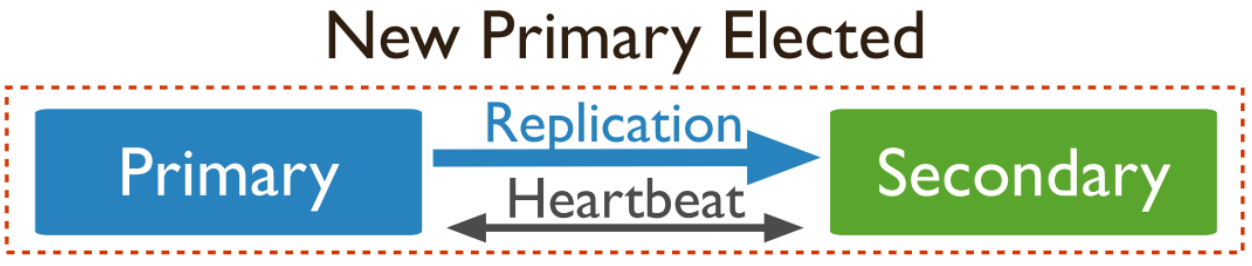
Since:

- There are 3 voting members total,
- And 2 are still online,

They form a majority (2 out of 3), which is sufficient to trigger an election.

- One of the secondaries steps up and initiates an **election**.
- The secondaries vote, and one of them becomes the new primary.

MongoDB's replica set election mechanism is inspired by Raft-like consensus — each member can propose itself and vote, but a majority must agree on the new leader.



- One of the secondaries is now promoted to Primary.
- The other remains a Secondary .
- Replication resumes (from primary to secondary).
- Heartbeat messages continue to ensure all members are alive and in sync.

rs.status() also gives information about elections

The replica set originally has 3 members:

- 1 Primary
- 2 Secondaries

We take down our primary. Two nodes are still running.

- The remaining 2 secondaries detect the failure (via missed heartbeats).
- Since 2 out of 3 nodes are still alive, they form a majority.
- One secondary initiates an election, and a new Primary is elected.

Replica set continues to operate normally.

Now we shut down the new Primary (which was previously a secondary).

- Only 1 node remains from the original 3.
- That node is a secondary, and it is now alone.
- It cannot elect itself to become primary, because a majority of voting members is required to hold an election (i.e., 2 out of 3).

No election occurs, no primary exists, replica set becomes read-only — no writes are accepted.

Split Brain Problem

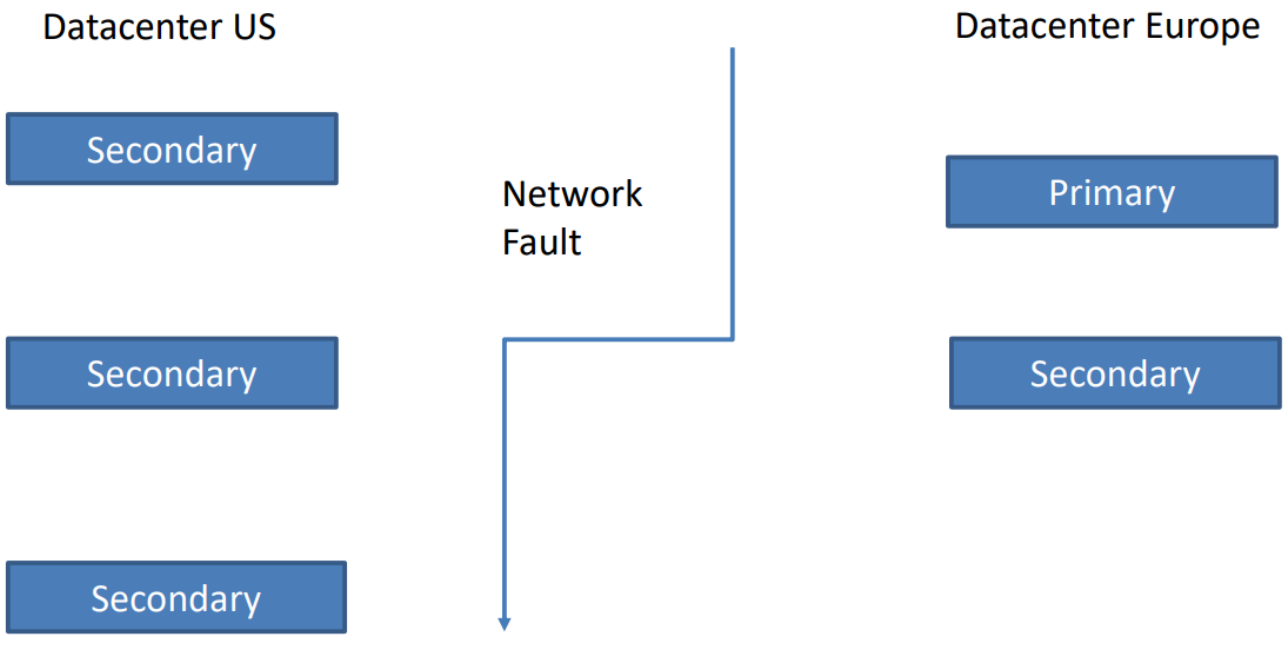
In distributed systems, a split brain occurs when:

- A network partition splits the cluster into two (or more) isolated groups.
- Each group believes the others are offline.
- Each group might try to act as the “true” primary, potentially allowing multiple primaries.

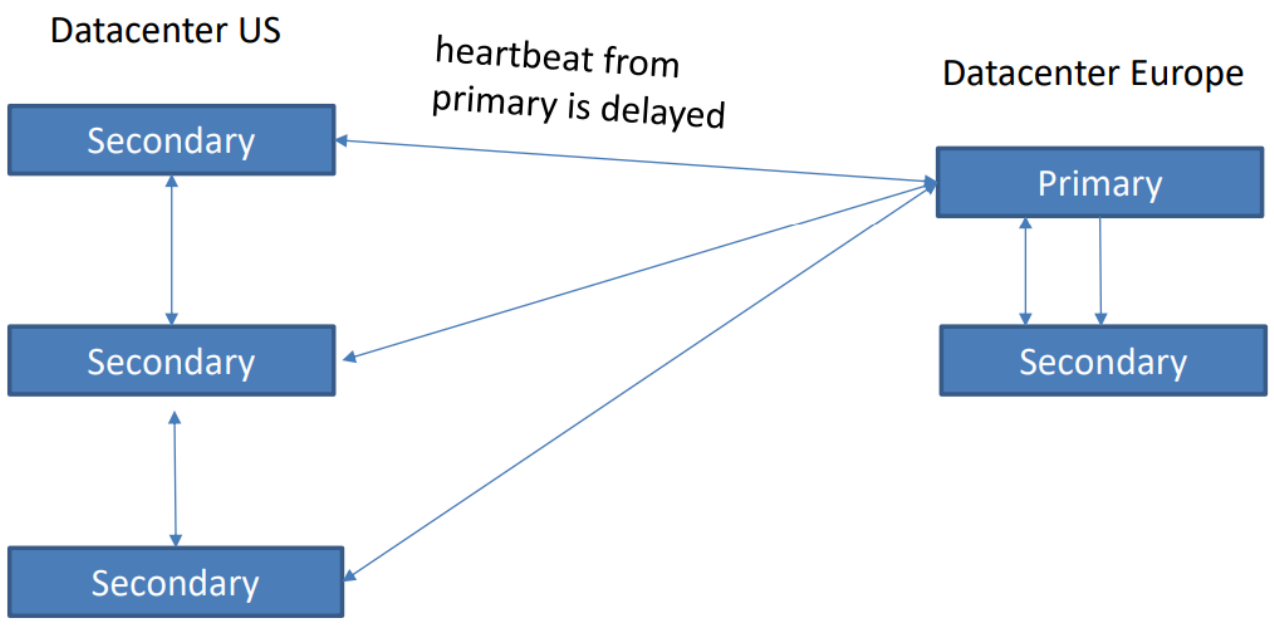
This is dangerous because it can lead to:

- Conflicting writes
- Data divergence (a situation where replicas of the same data no longer agree — i.e., their content becomes inconsistent over time.)
- Rollback scenarios
- Inconsistent read results

It typically occurs when Replicas accept writes independently without coordination (e.g., during a network partition) and those writes are not properly replicated or reconciled across all nodes.



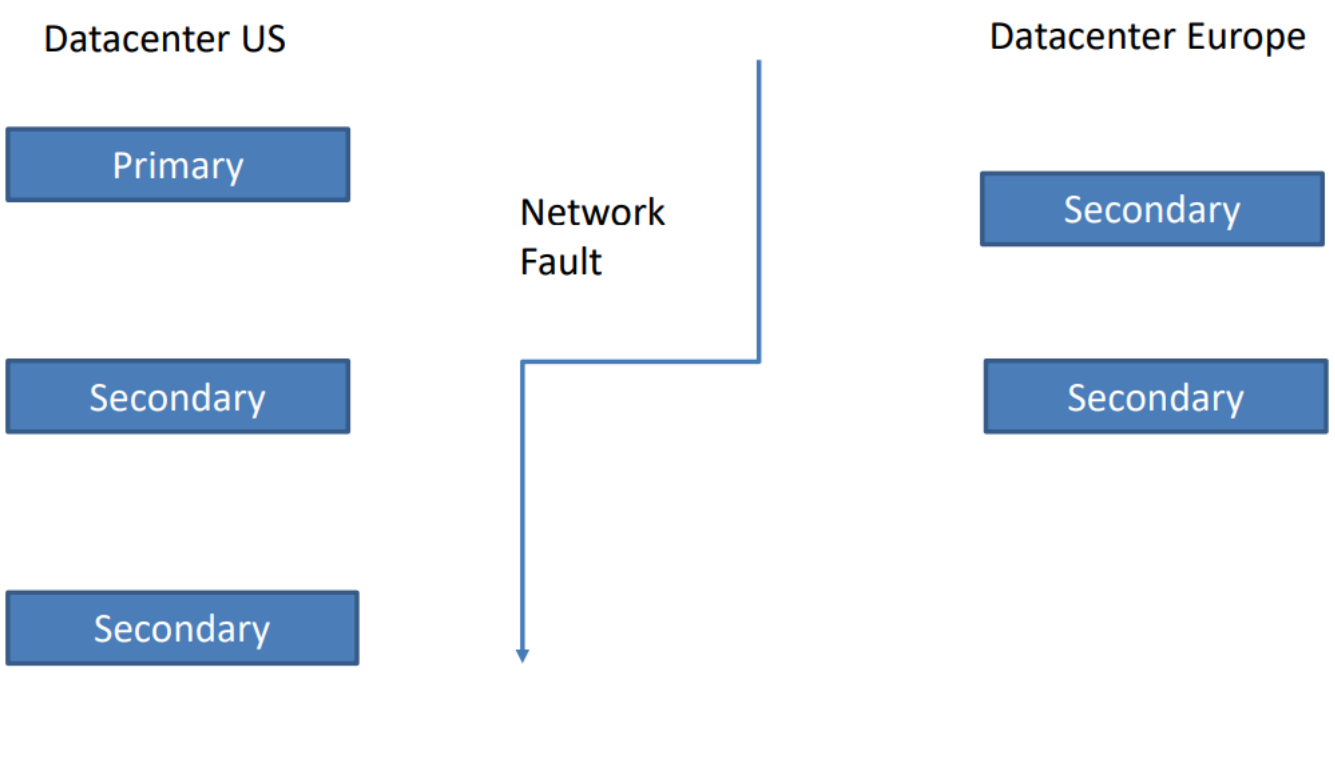
1. The primary is in Datacenter Europe, with secondaries in Datacenter US.
2. A network partition (network connection breaks) occurs.
3. Each side thinks the other is down.
4. A secondary in Datacenter US becomes a new (unauthorized) primary.
5. Datacenter Europe and US still accept writes.
6. once the connection renews the election will start again.
7. Datacenter US's primary will the election, since it will have the majority.
8. Datacenter Europe's primary will get demoted to secondary.



1. The primary is in Datacenter Europe, with secondaries in Datacenter US.
2. Datacenter US secondaries think that the primary is dead, because of the heartbeat.
3. Database US elects a new primary (has the majority).
4. Datacenter Europe's primary still thinks that it has the majority and it stays as the primary.
5. Datacenter US and Europe both accept writes.
    - Each side writes different changes to the same data.
    - These changes are not replicated across the partition.

- Once the network heals, MongoDB must reconcile differences, possibly using rollbacks — or data may be lost/conflicting.

Single Leader Replication



US behaves as before:

- US primary remains primary.
- Continues accepting writes.

Europe has less than a majority, it:

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

Replication Lag

> Replication lag is the delay between a write occurring on the primary and the same write being applied on a secondary. MongoDB uses asynchronous replication, so some lag is expected — but too much lag leads to problems.

In MongoDB, replication is asynchronous by default:

- The primary processes the write immediately.
- Secondaries pull new operations from the primary's oplog and apply them eventually.

Consequences of the replication lag:

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.

Causes of replication lag:

Cause	Explanation
High write load	Primary is processing many writes faster than secondaries can replicate.
Slow hardware (disks, CPU)	Secondaries cannot apply oplog entries fast enough.
Network latency or congestion	Slows down transfer of oplog entries between nodes.
Chained replication	If secondary A is syncing from secondary B (not directly from primary), it can lag even more.
Oplog too small	If lag is too high, and oplog entries expire before secondaries catch up, they must do a full resync.
Blocking operations	Index builds or long queries on secondaries can delay replication.

how MongoDB helps detect replication lag:

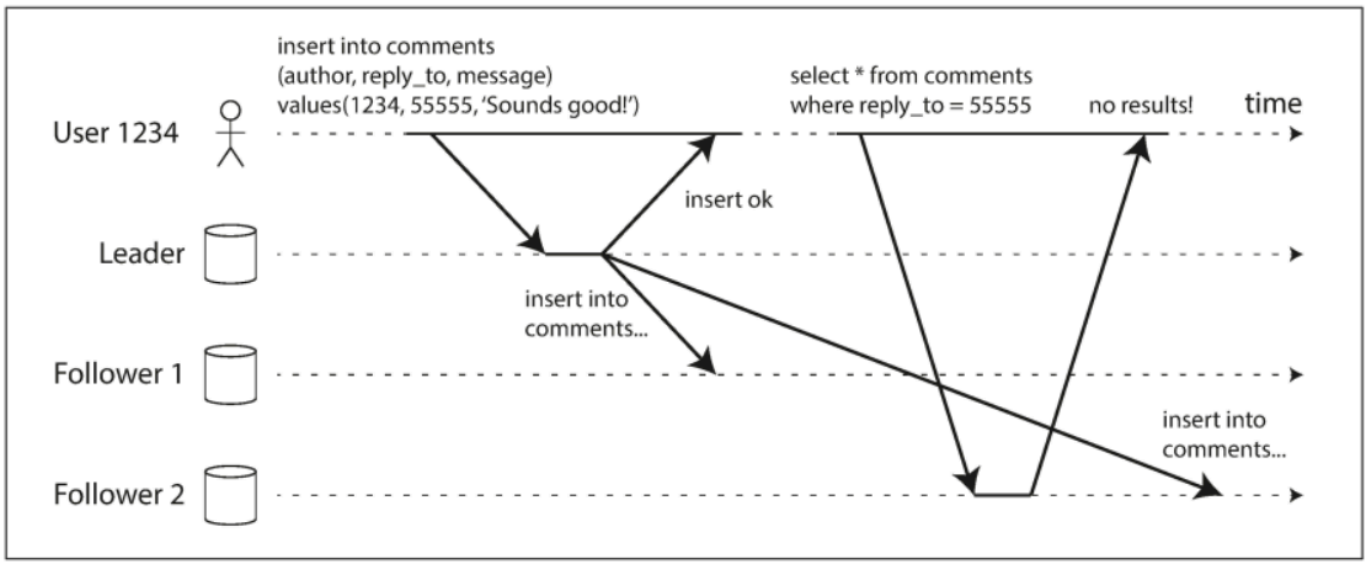
Command	Purpose
`rs.printSlaveReplicationInfo()`	Simple overview of replication lag per node (run this command on primary node)
`rs.status()`	Full health check including replication timestamps (Lag = Primary.optimeDate - Secondary.optimeDate)
`oplog.rs` timestamps	Advanced tracking of applied operations

Lag Time	Interpretation
0–2 seconds	✅ Normal and healthy
5–10 seconds	⚠️ Slight lag — monitor if sustained
> 30 seconds	❌ Problematic — may affect read consistency
> oplog window	❗ Critical — secondary may fall off the oplog and require a full resync (expensive)

How 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
Fast storage on secondaries	Faster oplog application
Disable chaining	Avoid indirect lag propagation
Optimize network	Reduce data transfer latency
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

Issues Resulting from Replication Lag



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.

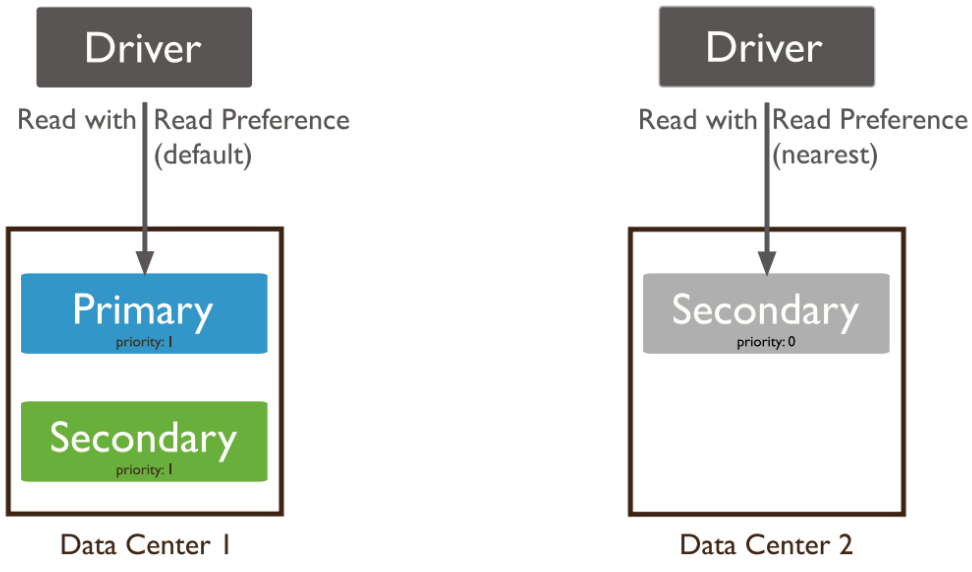
MongoDB Read Routing / Read Preference

Read Preference controls which replica set member (primary or secondary) a read request is routed to.

This lets you balance between:

- Consistency
- Availability
- Performance / Latency

Mode	Description	Staleness Risk
`primary` *(default)*	Always read from the primary (strong consistency)	None
`primaryPreferred`	Prefer primary, fallback to secondary if unavailable	Some
`secondary`	Always read from a secondary	Yes
`secondaryPreferred`	Prefer secondary, fallback to primary	Yes
`nearest`	Read from node with lowest ping time (regardless of role) (decreases latency)	Likely



In a replica set, only the primary is guaranteed to have the most up-to-date data.

Reading from secondaries:

- Can reduce load on the primary
- But may return stale or eventually consistent data

So if you have write heavy application, you have to make the primary default, so secondaries have more time to catch up, While if you have a read heavy application, then secondary is preferred for better performance.

MongoDB Read Concern

- `writeConcern` – controls durability of writes, controls how many nodes store the write.
- `readConcern` – controls consistency level of reads, controls from where and how safely you can read the data.

Write concern and read concern are independent.

Write concern determines how many replica set members must confirm a write before MongoDB considers it successful.

Write Concern	Description
`w:1`	Acknowledge only when written to primary
`w:"majority"`	Acknowledge when written to majority of nodes
`w:0`	No acknowledgment at all (fastest, least safe)
`w:n`	Acknowledge when written to **n** nodes

Read concern determines what level of consistency is required when reading data. `readConcern` allows you to control the consistency of reads from replica sets and sharded clusters.

Read Concern	Description
`local` **(default)**	Returns data from the node's current state (fast, may be stale, even dirty read) (If the read does not need to be absolutely up-to-date)
`majority`	Returns data that has been acknowledged by a majority of the replica set (If the read needs to be up-to-date, no dirty reads, stale reads possible)
`linearizable`	Returns the most recent committed data — only from the primary, only for reads of single documents
`available` ***(sharded only)***	Returns whatever is available on any shard/replica, regardless of consistency

Through the effective use of write concerns and read concerns, you can adjust the level of consistency and availability guarantees as appropriate:

- **Performance** (e.g., fast reads via `local`)
- **Consistency** (e.g., strict reads via `majority` or `linearizable`)
- **Availability** (e.g., fallback reads via `available` or `nearest`)

ReadConcern can be set on:

- **Query level** – set per individual query
- **Session level** – applies to all queries in a session
- **Database level** – global default for all queries on a database

---

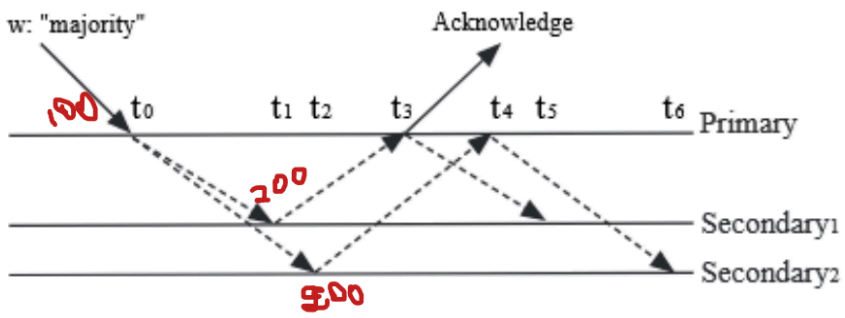
Read concern as query option:

```
db.teacher.insertOne({
  "t_id": 5,
  "t_name": "Angel",
  "t_mail": "angel@galopp.xx",
```

```
  "t_postalcode": 5600,
  "t_dob": new Date(2000-10-20),
  "t_gender": "f",
  "t_education": "HighSchool",
  "t_counter": 0
})

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

- We inserted `"Angel"`
- Then immediately tried to read `"Angel"` with `"majority"`
- The read may return the document or not, depending on whether it has been replicated to a majority.



t0 – t6: time points

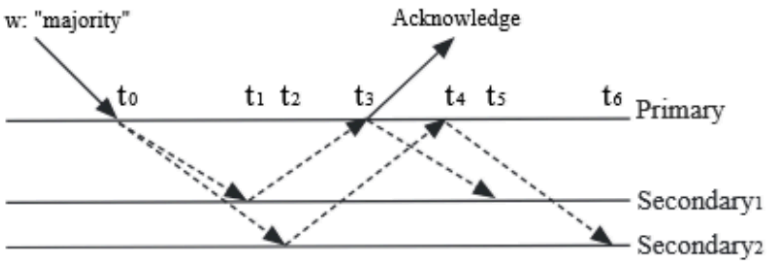
data object x:
before the write at t0: 100
after the write t0: 200

Which value do the following reads return with `readConcern("local")`?

With `readConcern: "local"`:

- Reads return the **most recent local state** of the node,
- Without waiting for:
  - Majority acknowledgment
  - Oplog replication
  - Write concern to be satisfied

#	Read Description	Value	Reason
1	Read on **Primary after time t0**	`200`	Primary applied the write at `t0`. With `readConcern: "local"`, it returns immediately.
2	Read on **Secondary 1 before t1**	`100`	Secondary 1 has not yet received the write at this time — it still sees old value.
3	Read on **Secondary 1 after t1**	`200`	Secondary 1 has now applied the write locally — `readConcern: "local"` allows reading it.
4	Read on **Secondary 2 before t2**	`100`	Write hasn't reached this node yet — it still sees old value.
5	Read on **Secondary 2 after t2**	`200`	Secondary 2 has applied the write locally — it can now return the new value.



t0 – t6: time points

data object x:
before the write at t0: 100
after the write t0: 200

Which value do the following reads with `readConcern("majority")` return:

A read with `readConcern: "majority"` only returns data that has been replicated to a majority of replica set members and acknowledged.

#	Read	Value	Reason
1	Read on **Primary before** `t3`	`100`	Majority hasn't acknowledged the write yet.
2	Read on **Primary after** `t3`	`200`	By `t3`, Secondary1 has applied the write, and at `t5`, the write is fully acknowledged.
3	Read on **Secondary1 before** `t5`	`100`	Even though this node has applied the write, it cannot return `200` until the majority has acknowledged it.
4	Read on **Secondary1 after** `t5`	`200`	After `t5`, the write is acknowledged by the majority. This read can safely return the new value.
5	Read on **Secondary2 before** `t6`	`100`	This node hasn't yet applied the write.
6	Read on **Secondary2 after** `t6`	`200`	After `t6`, it has applied the write and the write is majority-committed → safe to read with `"majority"`.