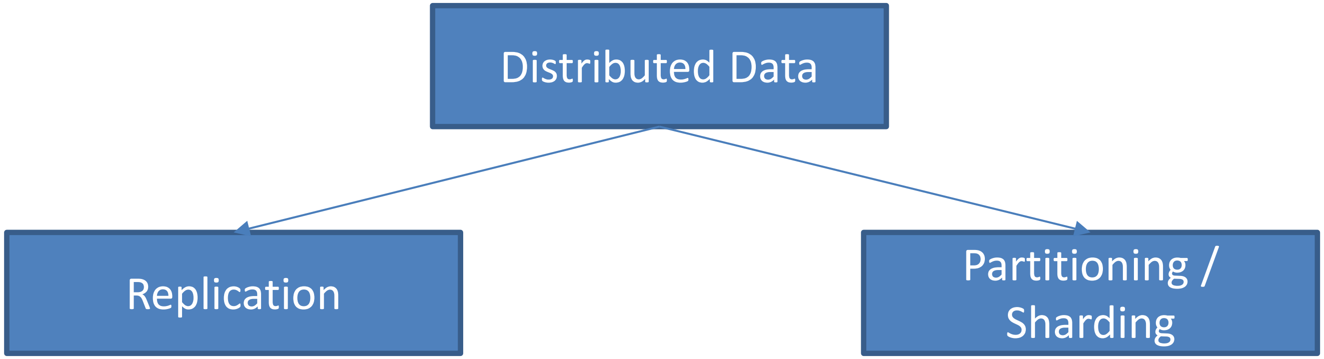
9
Distributed Databases
Replication

Reading: [Kl], chapter 5; [Ha] chapter 8
Seven Databases in seven weeks, chapter 4, 3

Distributed Databases



Distributed Data

Replication

Partitioning / Sharding

Copies of the data are kept on mutliple nodes.

Purpose? increase avalibility,durablity, backup, no data loss

Main challenge? to keep copies in sync

Data is divided onto multiple nodes.

Purpose? tolerate high workload, high volume of data

Main challenge? to distribute workloads evenly over partitions

```
t1
begin;
update price
set price = price + 10
commit;
```

```
t2
begin;
update price
set price = price * 1.2
commit;
```

starting value
for price = 100

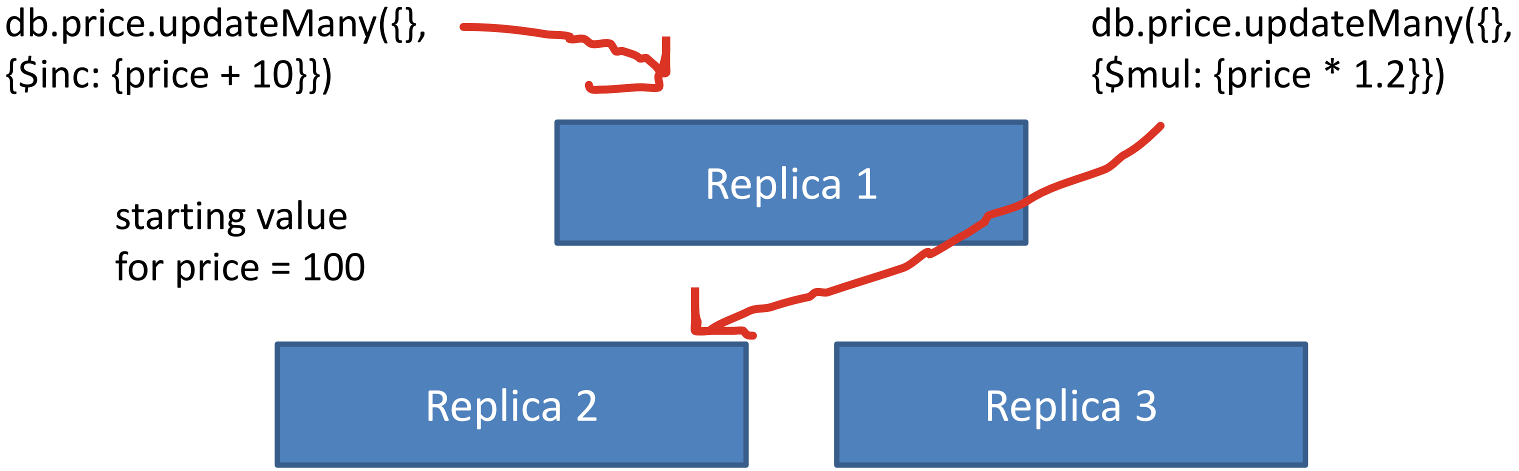
Single Node Server

Possible outcomes for value price:

1. If t2 starts after t1 commits: 132
2. If t1 starts after t2 commits: 130
3. If they run parallel, either 2PL or MVCC will make sure that result is consistent.

The result value is **either 132 or 130**!

Replication



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

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

starting value
for price = 100

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

What price value is replicated to rep3?
How do rep1 and rep2 replicate with each other?
The 2 write commands get the replica set out of sync.

so our goal is violated

so in replica 1 we get value 110 and in replica 2 we get 120. Now replica 1 and 2 need to replicate with replica 3 because our main goal is to keep copies in sync. But replica 3 gets either 110 or 120 depending on which replica replicates first.

Synchronization Issues in Replicated Databases
Riak Example

```
Creating a bucket (Riak version
1.x):
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(): allows multiple possible values for an object. A single key can store multiple sibling values, caused by concurrent writes on the same or on different nodes.

→ by definition, sibling values conflict with each other

→ The application will need to develop a strategy for conflict resolution, i.e. the application will need to decide which value is more correct depending on the use case.

allow_mult(true) is to resolve such a conflict.

LWW: Riak keeps only the value of the last write and disregards values of previous (concurrent) writes. "The problem with LWW is that it will necessarily drop some writes in the case of concurrent updates in the name of preventing sibling creation." https://docs.riak.com/riak/kv/latest/developing/usage/conflict-resolution/index.html

allow_mult(false) means that last write wins. In system they decide themselves which one is the last one. No siblings are created and Riak overwrites previous versions with the last one it accepts.

Databases that focus on data consistency typically choose a different approach to replication.

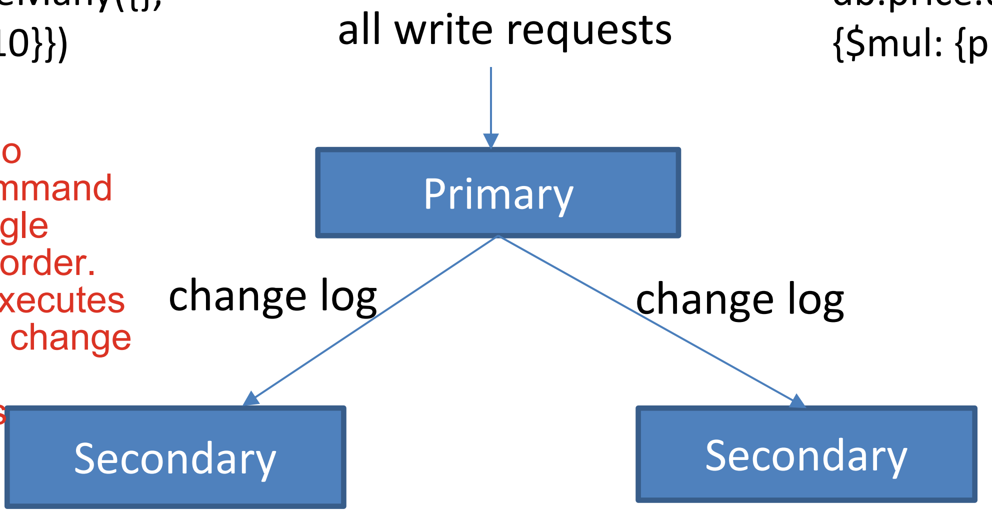
Single Leader Replication

(all write requests are handled by one server)

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

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

in the previous model we could not establish order, so we could not say which command happened first, but with single leader replication we have order. And in this order Primary executes all the protocols and sends change log which in mongodb is called oplog to secondaries and they apply changes from Primary.



- Writes must be addressed to the primary.
- Primary processes all writes and sends a change log – in the order writes were processed – to secondaries. Secondaries apply the changes – in the order they received them in the change log from the primary.
- MongoDB change log: oplog
- Oracle, Postgres, MySQL, MongoDB, CouchBase, Neo4j
- in certain configurations: Cassandra, Redis

these two concurrent write requests both go to Primary which then can apply all the measures and protocols that CC (Concurrency Control) could apply such as 2PL, MVCC... And then Secondaries apply changes that already happened on Primary.

goal is to keep copies in sync by which we get consistency.

MongoDB Replica Set

- In MongoDB, replication is implemented using a Replica Set, rs object. A replica set is a set of Mongod 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.

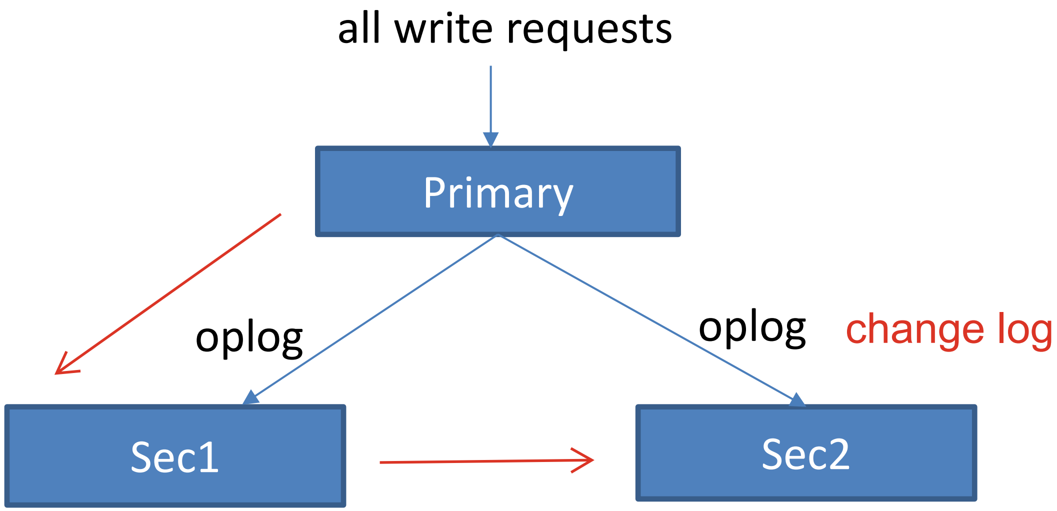
default port: 27017 which we leave untouched

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

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

1. create data directories
2. start mongod instances
3. initiate replica set and we only need to initiate once.

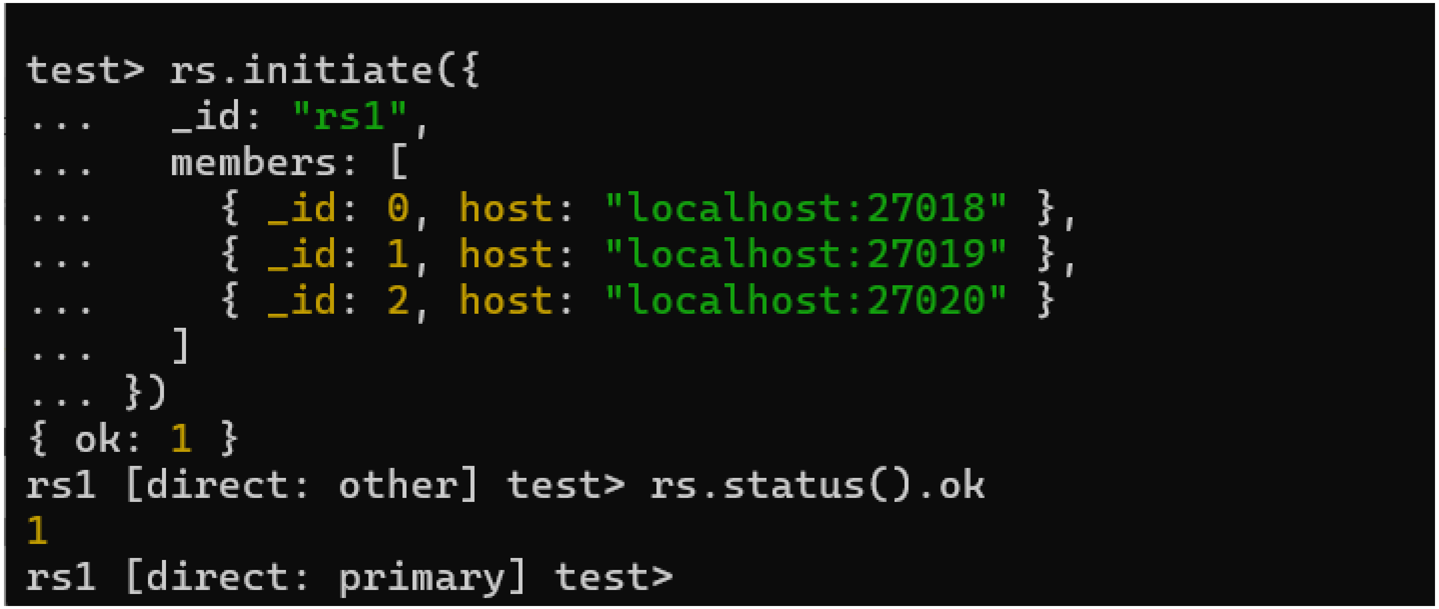
MongoDB Single Leader Replication



secondary can be promoted to primary and primary
demoted to secondary

Chained replication is also possible: primary → sec1 -→ sec2

In mongoDB it is also possible to have change replication meaning that primary sends oplog to Sec1 and Sec1 sends to Sec2, in which case arrow from primary to Sec2 will no longer be needed.



```
test> rs.initiate({
...    _id: "rs1",
...    members: [
...      { _id: 0, host: "localhost:27018" },
...      { _id: 1, host: "localhost:27019" },
...      { _id: 2, host: "localhost:27020" }
...    ]
... })
{ ok: 1 }
rs1 [direct: other] test> rs.status().ok
1
rs1 [direct: primary] test>
```

Return the status of the current mongod instance:

rs.status().ok

Return comprehensive information on the replica set, all nodes and their status:

~~rs.status().ok~~ rs.status()

We open three mongodb shells and connect to the three different mongodb instances running on different ports of the replica set. We automatically connect to db test. The prompt shows us whether we are connected to the primary or to a secondary.

MongoDB Replica Set

- On the primary, we 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})
```

this will be inserted

- On the secondary, we check if the document was replicated.

we can see all teachers even the one we added on primary

- On the secondary, we try to insert another document.

```
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})
```

we get error: MongoServerError[NotWritablePrimary]: not primary

- On the primary, we update the document:

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

it is updated on primary successfully and it is replicated on secondary as well.

Write operations only on the primary, read operations on primary or secondaries

MongoDB RS Synchronization

- All write operations are recorded in a collection called **opLog** (OperationsLog).
- The opLog is a capped collection. It is stored on db local.
- Secondaries apply the opLog **asynchronously** to their data.

capped collection has a fixed size and when it's full it starts to override oldest one.

Querying the oplog:

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

namespace

most recent upfront

Fields of the opLog are:

ts:	timestamp of the oplog entry
op:	operation type (i/u/d - insert, update, delete; n – no-op, for synchronizing)
ns:	namespace (database & collection)
o:	operation applied

in previous slide we did insert and update and therefore with sorting and limiting we should get those as well. with update we will see the difference about what was changed but with inserted we just see what was inserted.

Capped Collection = Collection with fixed size limit

Capped collections are collections with a fixed, predetermined size limit. The maximum size is defined in bytes and / or the maximum number of documents allowed. If the collection has reached the predefined size / prefixed number of documents, the oldest documents are simply overwritten.

Advantages:

- Capped collections are sorted according to the insertion time.
- Automatic removal / overwriting of old documents (can be an advantage depending on the use case)
- Very suitable if documents (contents) are stored only for a certain period of time and then should be deleted automatically.

Capped collections cannot be distributed to shards.

Application scenario: log files – specifically: oplog

Commands:

```
db.createCollection("log", { capped : true, size : 5242880, max : 5000 })    #creates a capped collection

db.<collection name>.isCapped()                                               #checks if a collection is capped.
```

MongoDB RS Synchronization

1. What issue do you see with the following update command in respect to replication?
2. Check in the oplog how mongodb handles this. Explain.

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

Date.now() is nondeterministic and if we apply this to primary and then secondaries copies won't be in sync anymore.

MongoDB Replication OpLog

https://www.mongodb.com/docs/manual/core/replica-set-oplog/

"Each operation in the oplog is idempotent. That is, oplog operations produce the same results whether applied once or multiple times to the target dataset."

idempotent: $f(x) = f\ (\ f(x)\ )$

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
}
```

Single-node MongoDB database: A write is successful when "acknowledged: true" is returned to the client.
When is a write considered to be successful in a MongoDB replica set?

MongoDB Replication

When is a write considered to be successful and acknowledged in a MongoDB replica set?

A) The write operation is completed on the primary node.

B) The write operation is completed on the primary node and on at least one secondary node.

C) The write operation is completed on the majority of nodes.

D) The write operation is completed on all nodes.

usually it's C because we want write operations to be completed on majority of nodes so that nothing goes wrong, however in mongoDB we have different options. We can define ourselves when write is acknowledged to be successful, you can see more details on next slide.

MongoDB Write Concern

The Write Concern parameter defines what an acknowledgment (acknowledged: true) implies. The following parameters are possible:

w:0 //Write operation is not acknowledged. (Rollback possible!)

w:1 // Write operation is acknowledged to the client when the operation on the primary node has been successfully completed. (Rollback possible!)

w:2 // Write operation is acknowledged to the client when the operation has been completed successfully on the the primary and one secondary node. (Rollback still possible?)

w:majority // default: Write operation is acknowledged to the client when the operation is completed successfully on the majority of the nodes. (Rollback not possible)

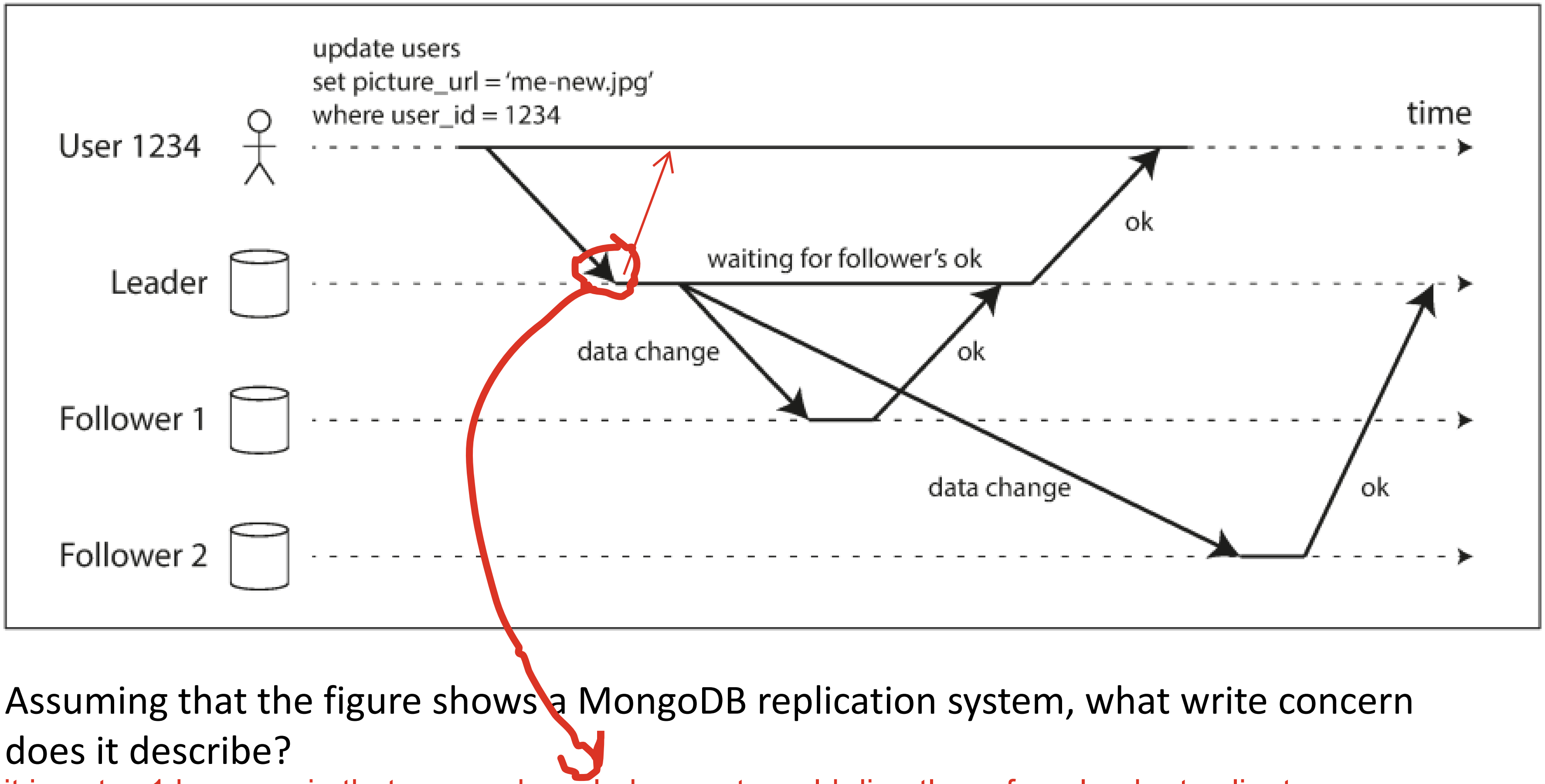
w:# of nodes // In a replica set with 3 nodes write concern w:3 is acknowledged to the client when the operation is completed successfully on all nodes.

Write concern allows the application to prioritize performance or durability.

so we still have asynchronous replication but we can force that system waits until all the nodes are replicated, basically forcing system to be sync which effects on performance and system is not meant to be for that.

MongoDB Write Concern

ok is acknowledgement



Assuming that the figure shows a MongoDB replication system, what write concern does it describe?

it is not w:1 because in that case acknowledgement would directly go from leader to client.

so answer is either w:2 or w:majority

[Kl], p.154

MongoDB Write Concern

Write concern can be set at

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

Example:

We update a teacher document with w:3

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

We stop the mongod instance on one of the secondaries (ctrl – c).

We update a different teacher document with w:3

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

Result? MongoWriteConcernError[WriteConcernFailed]: waiting for replication timed out

we had w:3 so all nodes and write is acknowledged to the client when the operation is completed successfully on all nodes but one of the nodes went down and therefore write was not successful

MongoDB Replica Set

What are the main reasons for replicating a database?

A) Increase Availability ✓

B) Scaling (distribute work load) ✗ this is for partitioning/sharding

C) Increase Durability (no loss of data) ✓

MongoDB Failover Strategy

Looking at a single-leader replicated database like MongoDB: What is the biggest risk in regard to availability / durability?

if primary goes down we can't do write operations anymore.

The system needs to have a failover strategy:

- It needs to be able to detect a failure that would stall availability (failure detection).
- It needs to autmatically switch to redundant (replicated) components to provide availability (automatic failover). repair

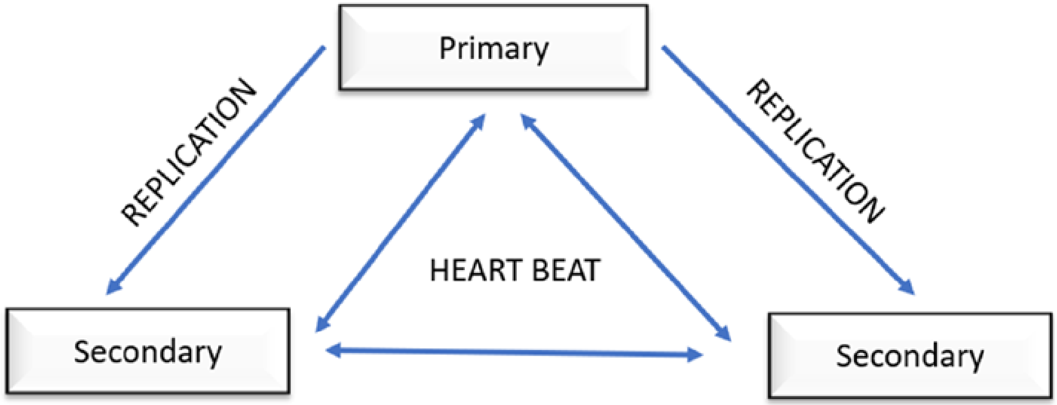
Native Postgres does not come with automatic failover. MongoDB comes with automatic failover.

Failure Detection

Desired failure detection properties:

1. Completeness:
   All running nodes eventually notice that a member fails.
2. Efficiency:
   The algorithm can detect the failure fast (within a couple of seconds).
3. Accuracy
   The algorithm is not accurate if it states a delayed (but running) node as dead or a failed node as delayed only.

Heartbeats or Pings are the common method to implement failure detection: They are very simple and possess strong completeness property. Usually, the database system has a parameter to set / tune the relation between efficiency and accuracy.



Failure Detection

MongoDB: https://www.mongodb.com/docs/manual/core/replica-set-elections/

"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

heartbeat time out is basically how long do nodes wait each other until they declare that other node is dead. 10 secs is default.

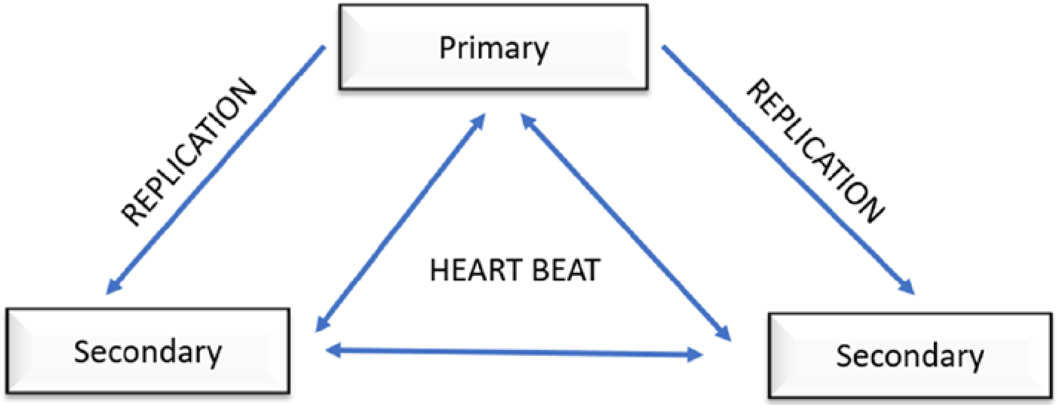
Increasing the heartbeatTimeout from 10 secs to 12 secs:

A) decreases accuracy

long heartbeat is good for accuracy and short is good for efficency.

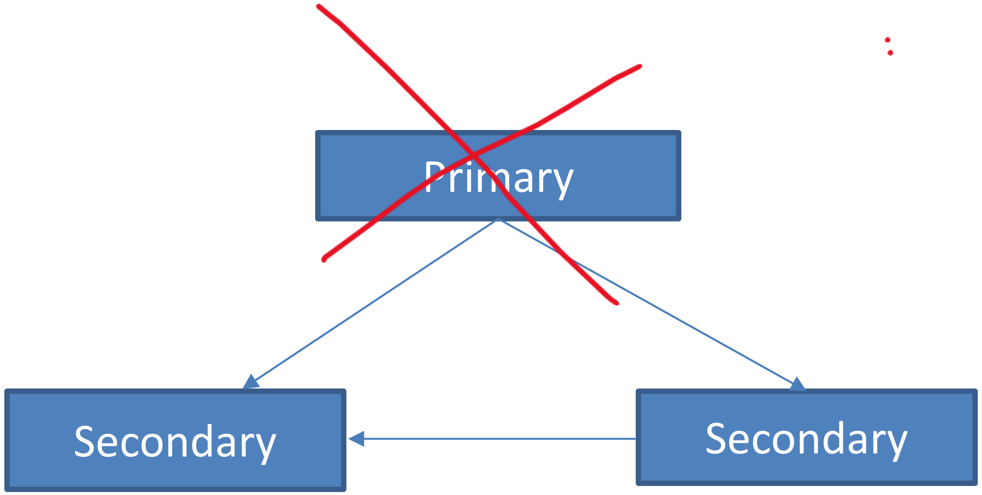
B) increases accuracy ✓

because by waiting longer we reduce false positives meaning we're less likely to incorrectly assume a node is down.



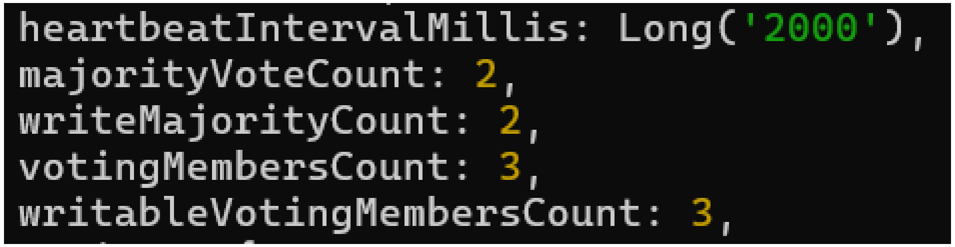
MongoDB Automatic Failover

If the primary fails ...

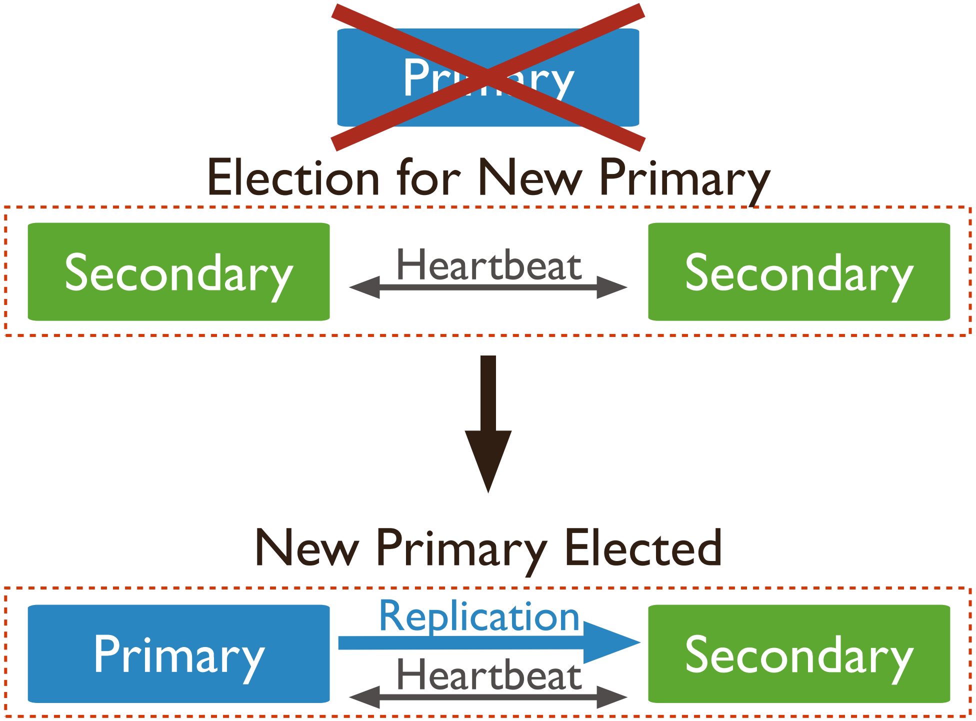


- When a primary does not communicate with the other members anymore, an eligible secondary calls for an election.
- The cluster can hold an election if the majority of voting nodes is available
- As soon as a new primary is elected, normal operations are resumed.
- No write processing during election.

MongoDB Election Process



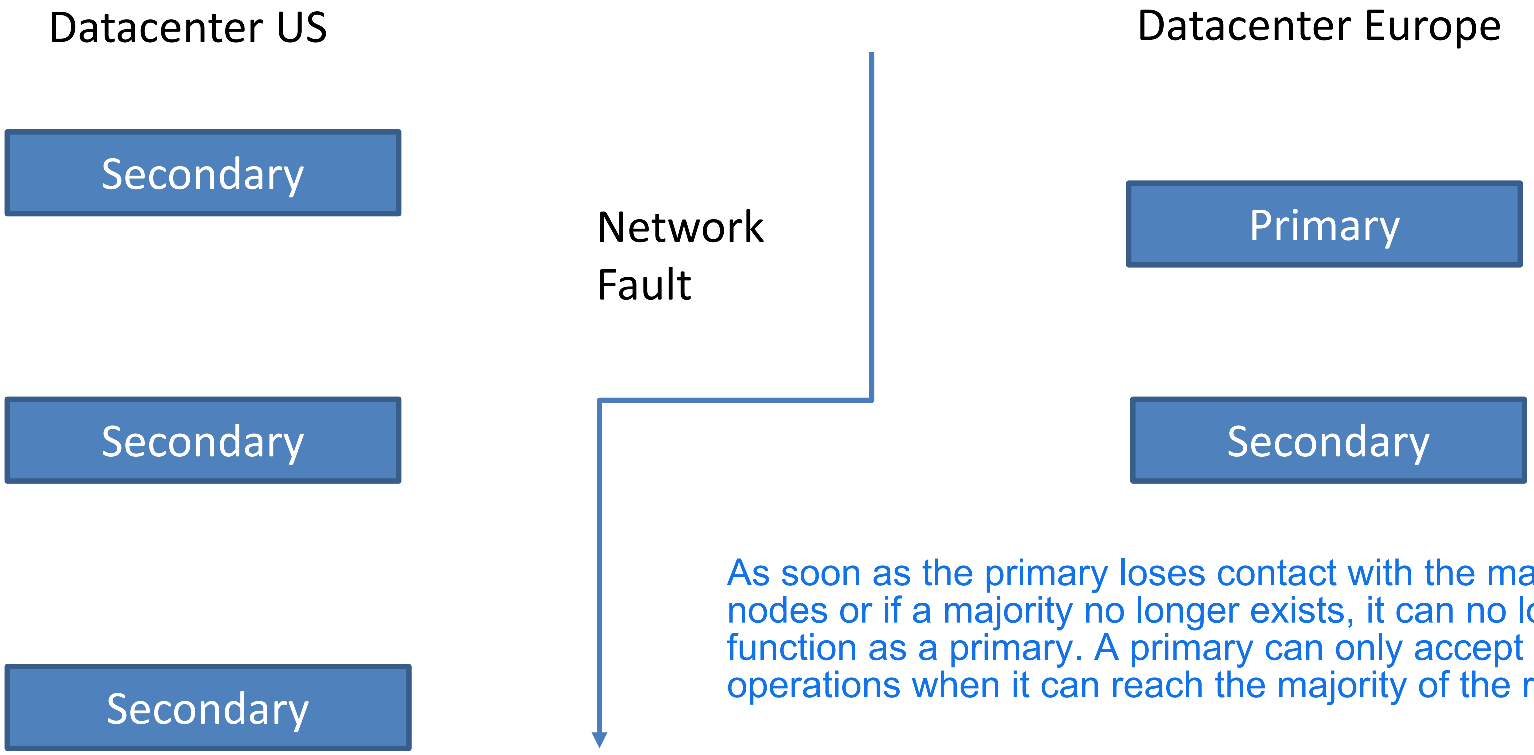
```
heartbeatIntervalMillis: Long('2000'),
majorityVoteCount: 2,
writeMajorityCount: 2,
votingMembersCount: 3,
writableVotingMembersCount: 3,
```



MongoDB Election Process

- We take down our primary. Two nodes are still running.
- Result? immediate election that gives us new primary
- Now we take down the new primary.
  One node is still running!
- Result? There is only one node running so an election is not necessary. This remaining node will stay in the secondary state because, without a majority of nodes MongoDB cannot maintain a replica set. This means that if we have fewer than the majority of nodes replication and write operations will no longer be possible.
- Explain.

Split Brain Problem

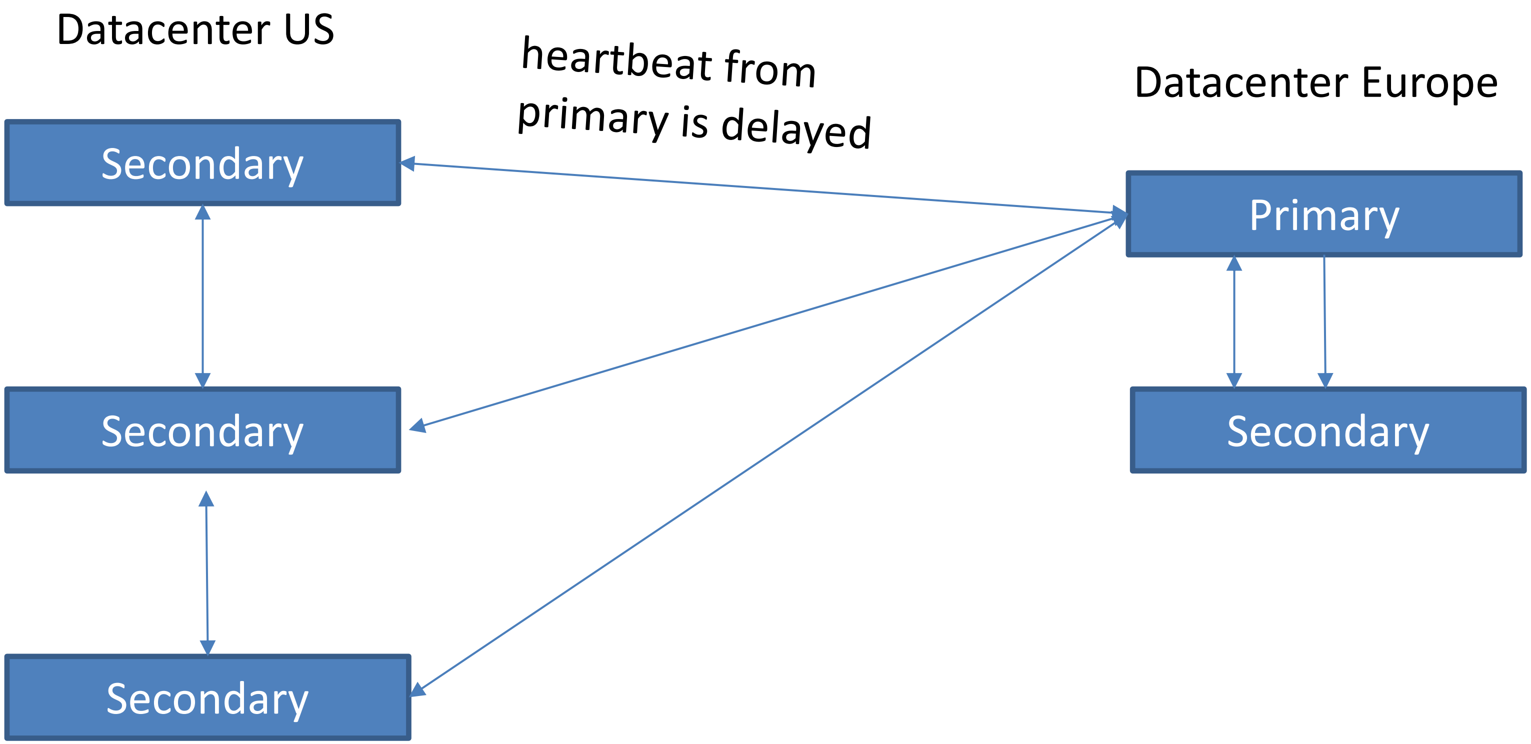


As soon as the primary loses contact with the majority of nodes or if a majority no longer exists, it can no longer function as a primary. A primary can only accept write operations when it can reach the majority of the replica set.

What happens when the network connection breaks? How do the nodes in Europe react? How do the nodes in the US react?

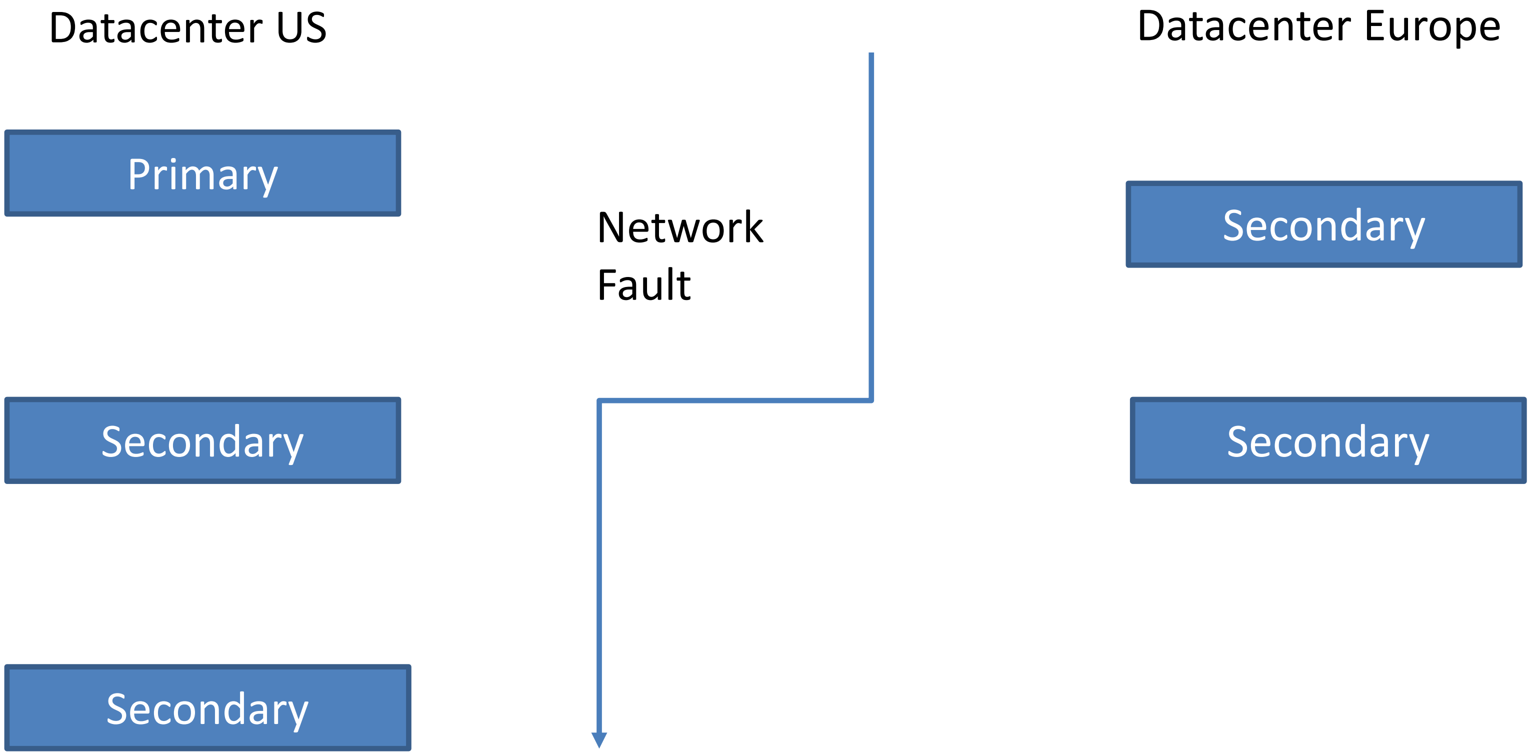
In this case when the network connection breaks, even though the primary and secondary nodes in Europe remain fully functional, the primary will demote itself to a secondary as soon as it detects that it can no longer reach a majority which in a 5-node replica set is 3 nodes. This is because a primary must be able to see a majority in order to accept write operations.

Split Brain Problem



The US secondaries think that the primary is down because the primary did not repond within the heartbeatTimeout. But primary is up and heartbeat simply delayed. What happens? How do the nodes in Europe react? How do the nodes in the US react?

Single Leader Replication



What happens in this scenario when the network connection breaks? How do the nodes in Europe react? How do the nodes in the US react?

Replication Lag

Asynchronous replication

→ delay between the write operation on the primary and its application on the secondaries.

→ secondaries do not apply the oplog at the same time so one secondary can be faster than another

→ replication lag. Replication lag is the delay between when a write happens on the primary and when that write is applied on a secondary.

As a result of the replication lag,

- read requests to secondaries may see stale data.

If you read from a secondary you might get old data because it hasn't caught up to the latest writes on the primary.

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

What could be common reasons for secondaries to fall behind?

network delay

One should not have different hardware: secondaries with slower CPUs, less memory or slower disks may not keep up with the write load resulting in replication lag.

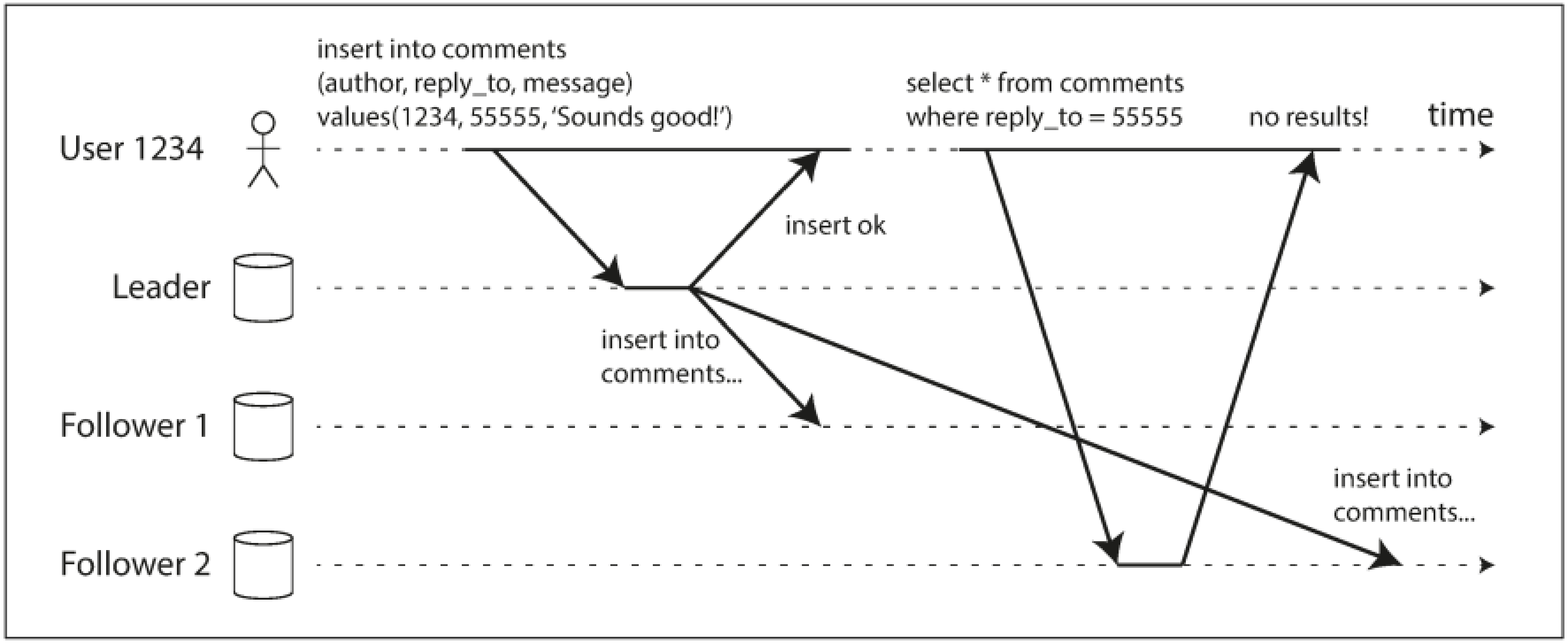
misconseption: primary works a lot, secondaries only reads
---> see often overloaded. They also have to apply each write.

secondaries have to do writes just as much as primary because they have to replicate and apply every single write operation just like the primary

since secondaries have to do all the writes and in addition reads also they may fall behind which could also be reason for replication lag.

Issues Resulting from Replication Lag

If you send your writes to the primary and your reads to the secondaries, but the secondaries can't keep up—because they also have to apply the writes—then you end up with a situation like this.



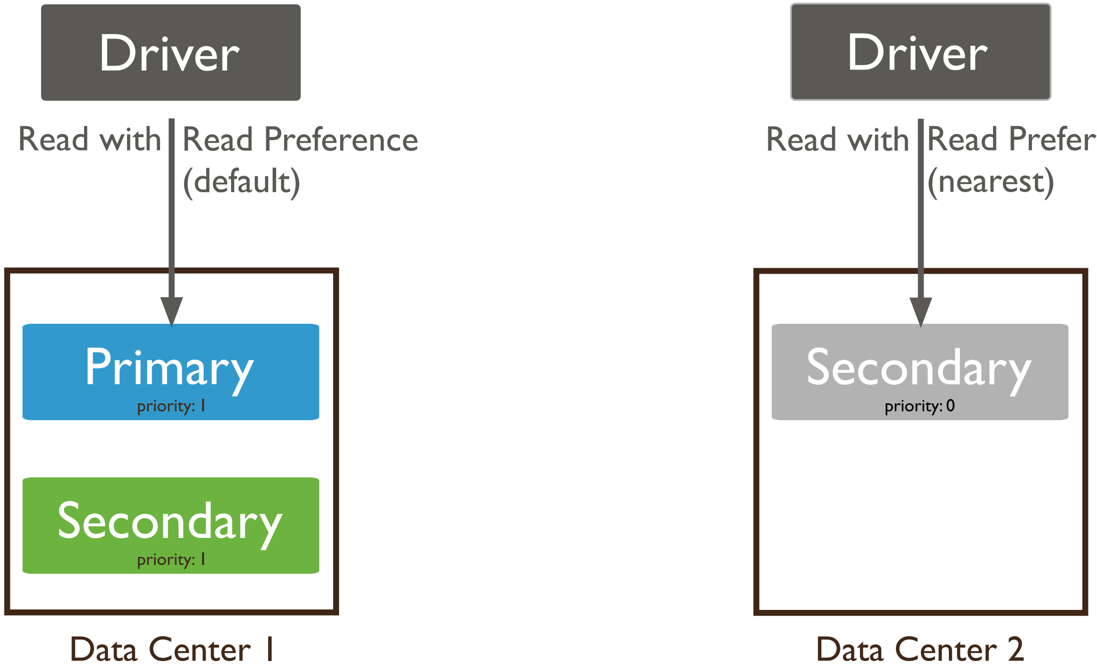
user1234 doesn't see his own writes

[Kl], p.163

Replication Lag

How to decrease replication lag?

MongoDB Read Routing / Read Preference

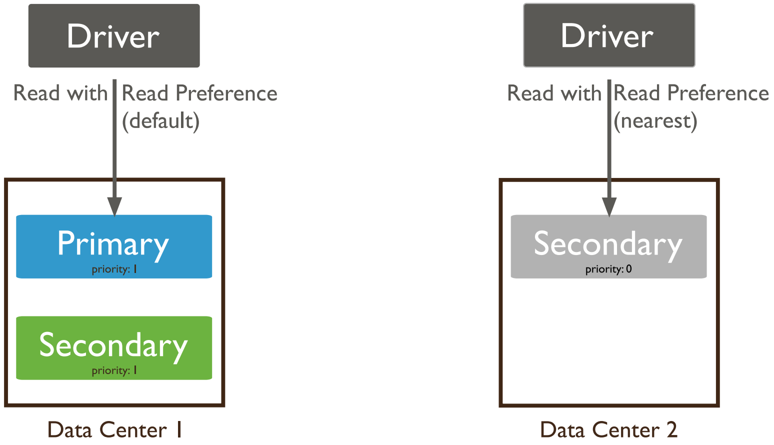


Read Preference: determines the server node a read request is routed to.
Can be set as

- query option
- connection option
- driver option

MongoDB Read Preference

Modes:
Primary (default),
primaryPreferred,
Secondary,
SecondaryPreferred,
Nearest.



Possible goals:

workload balancing: if we have write heavy application then we need to make primary default or primaryPreffered in order to give secondaries more room to catch up.
if we have read heavy application then we may say secondary or secondaryPreferred.

if we select nearst then we support locality and decrease latency

MongoDB Read Operations

MongoDB writeConcern

- controls durability of writes.
- sets different levels of what the acknowledgement back to the client means.

MongoDB readConcern

- controls how "up-to-date" a read request return is.
- controls the consistency level of a read
- for example: guarantee of "read-your-own-writes" or guarantee of monotonic reads.

writeConcern and readConcern are independent of each other.
Even with a w: majority it is possible to get a stale read and with a w:1 it is possible to get an up-to-date read.

with w:majority we can get old data from the secondary which has not replicated yet and with w:1 we can get up-to-date read if it's from one secondary which replicated or one primary

If the read does not need to be absolutely up-to-date: readConcern: local faster

If the read needs to be up-to-date: readConcern: majority

MongoDB Read Concern

https://www.mongodb.com/docs/manual/reference/read-concern/

"The readConcern option allows you to control the consistency ... of the data read from replica sets and replica set shards. ...

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

Important read concerns are: local, majority, linearizable.

local — fastest read of most recent writes, but no consistency guarantees (may be stale, even dirty read)

majority: — no dirty reads (stale reads may be possible)

linearizable: — only for reads on the primary, only for reads of single documents

if application has sensitive data and we only need concistent reads we use linearizable

Default readConcern is local

KUTAISI
INTERNATIONAL
UNIVERSITY

MongoDB Read Concern

ReadConcern can be set on

- query level
- session level
- database level

Read concern as query option:

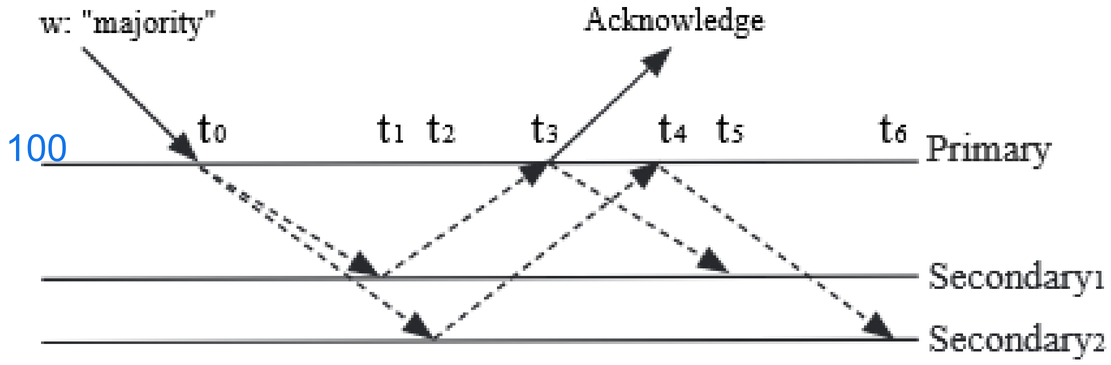
```
db.teacher.insertOne({"t_id":5,"t_name":"Angel","t_mail":"angel@galopp.xx","t_postalcode":5600,"t_dob": new Date(2000-10-20), "t_gender":"f", "t_education":"HighSchool", "t_counter":0})
```

```
db.teacher.find( {t_name: "Angel"}).readConcern("majority").maxTimeMS(1000)
```

What may the query return?

query may return old value but probability that it will return latest value is much higher

MongoDB Read Concern



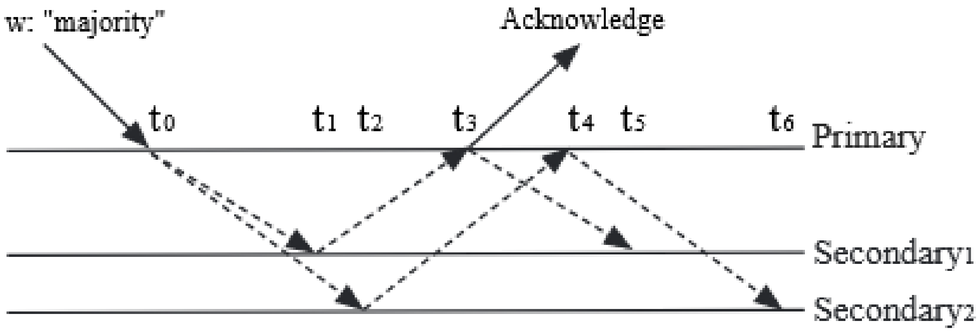
t0 – t6: time points

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

Which value do the folllowing reads return with readConcern("local") Reads the most recent data available on the node, whether or not it has been replicated or acknowledged. No guarantee of durability or consistency across the replica set. Data might be uncommitted or not replicated yet.

1. read on Primary after time t0 200
2. read on Secondary 1, before time t1: 100
3. read on Secondary 1, after time t1: 200
4. read on Secondary 2, before time t2: 100
5. read on Secondary 2, after time t2: 200

MongoDB Read Concern



t0 – t6: time points

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

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

1. read on Primary before t3 100
2. read on Primary after t3: 200
3. read on Secondary 1, before t5: 100
4. read on Secondary 1, after t5: 200
5. read on Secondary 2, before t6: 100
6. read on Secondary 2, after t6: 200

Ensures that the read returns only data that has been acknowledged by the majority of replica set members. This gives stronger consistency than readConcern("local"). The read waits if necessary until the data is confirmed by a majority before returning it.