Best Practices in MongoDB

8 rules for documents at scale. Journal for crashes. Oplog for the replica set. Everything timestamped.

1 Journal is durability, oplog is replication

j:true waits for the journal on disk. The oplog is a separate, ordered record that secondaries tail, two logs, two promises.

Deep dive ↓ Level 0 · Journal vs oplog

ONE WRITE, TWO DESTINATIONS

`insert on primary` →

`journal/WiredTigerLog.0000000123 · crash safety`

`local.oplog.rs · secondaries tail this`

Delete the journal and you lose the last **~60 s** since checkpoint. Fall out of the oplog window and a secondary must **resync from scratch**.

2 ESR: Equality, Sort, Range, in that order

Index keys are stored sorted field by field. Equality collapses to one range, sort rides free inside it, and a range ends the usable ordering, so range goes last.

Deep dive ↓ The ESR derivation, from the key layout

ONE COMPOUND INDEX, BUILT THE ESR WAY

`status: 1 · equality` → `createdAt: -1 · sort` → `amount: 1 · range`

✓ no SORT stage in the plan

✗ range first → scans every key in the range, then sorts in memory

3 The examined-to-returned ratio is the verdict

totalDocsExamined ÷ nReturned = 1 means a perfect index. At 10× the index is weak. At 1,000× it is decoration.

Deep dive ↓ explain("executionStats"), annotated

EXPLAIN("EXECUTIONSTATS"): READ THESE THREE NUMBERS

NRETURNED	TOTALDOCSEXAMINED	RATIO	VERDICT
42	42	1.0×	✓ index fits the query
42	420	10×	low-cardinality leading field
42	10,000,000 · COLLSCAN	-	✗ no usable index · ~20 GB read at 10M docs

4 When the cache is full of dirty pages, your threads evict

WiredTiger eviction keeps the cache usable. If application threads get pulled into evicting, query latency spikes with no slow query to blame.

Deep dive ↓ Level 1 · WiredTiger cache eviction

WIREDTIGER CACHE: THE TWO THRESHOLDS THAT MATTER



`cache ~50% of RAM − 1 GB`

`dirty bytes climb → eviction pressure`

`app threads evicting ✗ latency spike`

```
db.serverStatus().wiredTiger.cache  ← "bytes currently in the cac
```

5 The oplog window is your maximum allowed downtime

Oplog size ÷ write rate = hours. A secondary down longer than the window cannot catch up, full resync, at full data size.

Deep dive ↓ Level 5 · The oplog window math

THE ONLY OPLOG FORMULA THERE IS

`50 GB oplog` ÷ `2 GB/hour of writes` = `25 hours of window`

Compare that one number to your longest maintenance. **rs.printReplicationInfo()** prints it directly as "log length start to end".

6 A checkpoint every ~60 s bounds the crash

Checkpoints make the .wt files consistent. The journal only ever has to cover the gap since the last one, that gap is the recovery work.

Deep dive ↓ Level 3 · WiredTiger checkpoints & timestamps

RECOVERY AFTER A CRASH, ON A TIMELINE

`checkpoint ✓ collection-*.wt consistent` →

`journal replays ≤ ~60 s of writes` → `node rejoins the set`

No checkpoint design removes the journal's job, it **shrinks** it, from the whole dataset to one minute of writes.

7 w:majority is what survives an election

A write acknowledged by one node can vanish when that node loses an election. A majority-acknowledged write exists on a majority, the next primary must have it.

Deep dive ↓ Level 4 · Elections, write & read concern

3-NODE REPLICA SET, ONE WRITE

`primary · has it` `secondary 1 · has it ✓`

`secondary 2 · not yet` →

`2 of 3 = majority · survives any single election ✓`

```
db.orders.insertOne(doc, { writeConcern: { w: "majority", j: true
```

8 A jumbo chunk cannot move, and blocks balancing

Chunks split at ~128 MB. A chunk that cannot split (one shard key value holds it all) can never migrate, so one shard keeps growing while others idle.

Deep dive ↓ Level 5 · Chunk migration & jumbo chunks

SHARD KEY CHOICE, VISIBLE IN CHUNK SIZES

GOOD KEY: CHUNKS SPLIT & MOVE

`64 MB` `96 MB`

`128 MB → splits ✓`

BAD KEY: ONE VALUE, ONE CHUNK

`jumbo · 900 MB · cannot split ✗`

Balancer skips it forever. Cause is almost always a low-cardinality or ever-increasing key chosen at design time.