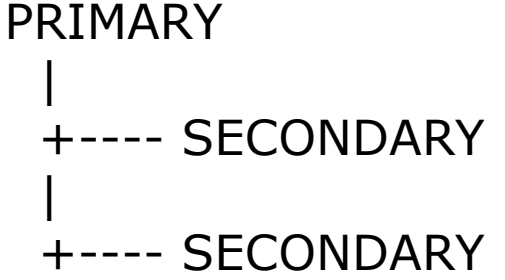
Distributed MongoDB: Most of it

When?

- MongoDB becomes *distributed* when you run it in **Replica Sets** (for availability) and **Sharded Clusters** (for horizontal scaling).
- Most production deployments combine **both**.

What is Replica Set?

1)A **replica set** = **one primary + multiple secondaries**.



Purpose:

- Fault tolerance
- Automatic failover
- Read scalability
- Durability

What is Replica Set?

How Writes Work

- All writes go to **primary**
- Primary **replicates oplog** to secondaries

How Reads Work:

- By default, reads go to **primary**
- But you can read from secondaries with:

```
db.collection.find().readPref("secondary")
```

Replica Set Internals: Oplog

Each node has an **oplog** (operations log), storing writes like:

```
{ op: "i", ns: "test.users", o:
{name:"Ali"} }
{ op: "u", ns: "test.users", o: {...} }
```

Secondaries **replay the oplog** → making them exact copies.

Automatic Failover (Elections)

If primary dies:

- Secondaries detect it using heartbeats (every 2 sec).
- Election starts using Raft-like consensus.
- One of them becomes the new primary.
- Clients auto-reconnect.

Downtime ≈ 2–5 seconds.

Write Concerns (Consistency Guarantees)

Controls how many nodes must confirm a write:

Write Concern	**Meaning**
w:1	Only primary confirms (fast, weak)
w:majority	Most nodes confirm (safe)
w:0	Fire-and-forget

Read Concerns

Read Concern	Meaning
local	Default, may see uncommitted data
available	Low consistency
majority	Only sees majority-committed data
linearizable	Strongest consistency, slowest

Read concern specifies the consistency and isolation level for read operations in MongoDB.

It determines which version of data your query reads.

Helps balance consistency vs. performance.

Typical Replica Set Deployment

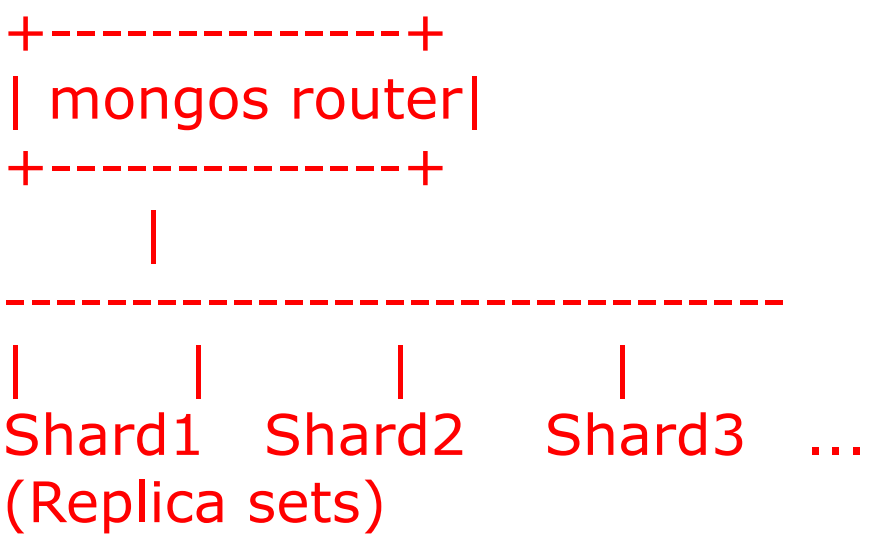
3 nodes: 1 primary + 2 secondaries

May add:

- **Arbiter** (votes only, no data)
- **Hidden node** (analytics)
- **Delayed node** (rollback protection)

Sharding

When data becomes too large for a single machine, MongoDB uses **sharding**.



Why Sharding

- Scale **storage**
- Scale **reads/writes**
- Reduce load on any single node

Sharding Components

mongos (Query Router)
- Lightweight router
- Receives client queries
- Routes them to the right shard(s)

Config servers (CSRS)
- Store metadata (chunk locations)
- Three-node replica set required

Shards
- Each shard is **itself a replica set**
- Provides both **scalability + availability**

How Sharding Works

Sharding Key: Each collection must choose a **shard key**

```
{ userId: 1 }
```

This key determines *which shard* stores which document.

Chunks: MongoDB breaks data into **chunks** (64 MB default).

```
Shard1: userId 1–10000
Shard2: userId 10001–20000
Shard3: userId 20001–30000
```

Chunks can move between shards based on load.

How Sharding Works

Balancing: A background process called the **balancer**:

- Monitors chunk distribution
- Moves chunks between shards when uneven
- Chunk migrations happen **online** while the cluster stays active.

Reads and Writes in a Sharded Cluster

Write Flow: Client → mongos → correct shard → primary of that shard

Read Flow: Depends on query

If query includes shard key: → Query goes to only one shard

```
db.users.find({ userId: 123 })
```

Without shard key: → Query is broadcast to all shards (scatter-gather)

Scatter–gather is slower → this is why good shard keys matter.

```
db.users.find({ age: 20 })
```

Choosing a good Shard Key

A good shard key:

- High cardinality (many unique values)
- Evenly distributed
- Queried often
- Not monotonically increasing (or use hashed)

Monotonic keys lead to hotspotting (single shard)

Hash: Spreads inserts randomly across shards

Choosing a good Shard Key

Examples:

✓ `userId`

✓ `emailHash`

✓ `region + timestamp`

Bad keys:

✗ timestamp (monotonic → hotspot)

✗ boolean (only two values)

✗ category with few values

Consistency Model

MongoDB default consistency:
- Writes: primary only (`w:1`)
- Reads: primary (`local`)

You can tune it to:
- CP-like (strong consistency)
- AP-like (high availability)

Better consistency = slower
Better performance = weaker guarantees

Distributed Transactions

MongoDB supports multi-document ACID transactions across shards (from v4.2).

Transactions work like:

- two-phase commit: 2PC to ensure atomicity across multiple documents or multiple shards.
- Used to update multiple documents in different collections or shards as a single atomic transaction (pre-4.2) or across shards in sharded clusters.
- It guarantees either all operations succeed, or none are applied.)

Distributed Transactions

MongoDB supports multi-document ACID transactions across shards (from v4.2).

Transactions work like:

- two-phase commit
- oplog coordination
- retryable writes (allow certain write operations to be safely retried by the client if a network error or primary failover occurs)

But:

- slower
- should be avoided unless necessary

Failover in Sharded Clusters

If one shard's primary fails:

- That replica set elects a new primary
- Only that shard is temporarily degraded
- Other shards continue normally

If config servers fail:

- Must maintain **quorum**
- Without quorum, cluster metadata access stops

Backup and Restore

Options:

- mongodump/mongorestore (small databases)
- filesystem snapshots
- Cloud Manager
- Ops Manager

For sharded clusters: backup **each shard + config server**.

Monitoring

Tools:

- Mongostat
- Mongotop
- MongoDB Atlas metrics
- Ops Manager dashboards

Metrics to watch:

- replication lag
- chunk imbalance
- page faults
- qps (queries/sec)
- connections
- cache usage