Replica Sets (Availability)

Q1) When do we say “MongoDB becomes distributed”?

Answer:

- MongoDB becomes distributed when you run it as **Replica Sets** (availability) or **Sharded Clusters** (horizontal scaling).
- Most production deployments combine **both** for availability + scale.
- Distribution changes behavior of reads/writes and failure handling (elections, routing).
  Example: A large e-commerce app uses sharding for scale + replica sets for failover.

Q2) Define a Replica Set. What is its structure?

Answer:

- A replica set = **1 primary + multiple secondaries**.
- Primary is the main writer; secondaries are copies that replicate changes.
- This provides redundancy and high availability via failover.
  Example: If the primary server dies, a secondary can take over.

Q3) Give 4 purposes of a replica set.

Answer:

- **Fault tolerance**: system survives node failure.
- **Automatic failover**: new primary elected automatically.
- **Read scalability**: reads can be served from secondaries (if configured).
- **Durability**: stronger write concerns reduce data loss risk.
  Example: Analytics reads go to secondaries to reduce load on primary.

Q4) How do writes work in a replica set?

Answer:

- **All writes go to the primary.**
- Primary records changes in the **oplog** and secondaries replicate from it.

- Secondaries replay operations to become exact copies.
  Example: Insert user → primary logs it → secondaries apply same insert.

Q5) How do reads work in a replica set? What is the default?

Answer:

- Default reads go to **primary**.
- You *can* read from secondaries using read preference like `"secondary"`.
- Reading from secondaries increases read capacity but can return stale data depending on replication lag.
  Example: Product catalog search reads from secondary; checkout reads from primary.

Q6) What is the oplog? Why is it essential?

Answer:

- Oplog = **operations log** storing inserts/updates/deletes.
- Secondaries **replay** oplog entries to stay consistent with primary.
- It's what makes replication possible and ordered.
  Example: `op:"u"` for updating a user's phone is replayed by secondaries.

Q7) Explain automatic failover (elections). What triggers it?

Answer:

- If primary dies, secondaries detect it via **heartbeats every ~2 seconds**.
- Election starts using **Raft-like consensus**, and a new primary is chosen.
- Clients **auto-reconnect**, downtime is around **2–5 seconds**.
  Example: Payment service sees a brief pause, then continues after new primary.

Q8) Scenario: Primary crashes mid-traffic. What exactly happens step-by-step?

Answer:

- Secondaries miss heartbeats and declare primary unavailable.

- They run an election; one becomes new primary (Raft-like).
- Clients reconnect; app resumes with short downtime (~2–5 sec).
  Example: Your API gets a few "retry" errors, then works again.

Write Concern (Consistency Guarantees)

Q9) What is Write Concern? Why do we need it?

Answer:

- Write concern controls **how many nodes must confirm** a write.
- It balances **speed vs safety** (durability).
- Stronger write concern reduces rollback risk during failover but adds latency.
  Example: Bank transfer uses `w:majority`, click-logging might use `w:1`.

Q10) Compare `w:1`, `w:majority`, `w:0`.

Answer:

- `w:1`: only primary confirms → **fast, weak**.
- `w:majority`: most nodes confirm → **safer**.
- `w:0`: fire-and-forget → **no confirmation**, riskiest.
  Example: Logging page views might use `w:0`, but orders should never.

Q11) Scenario: You used `w:1` and primary dies immediately. What can go wrong?

Answer:

- Primary confirmed, but secondaries may not have replicated yet → write may be lost.
- New primary might not contain that write; client thinks it succeeded.
- Using `w:majority` reduces this risk (more durable).
  Example: User paid, but order record disappears after failover → disaster.

Read Concern (Read Consistency)

Q12) What is Read Concern in MongoDB?

Answer:

- Read concern defines the **consistency/isolation** level for reads.
- It determines **which version** of data your query sees.
- It's a tradeoff: stronger consistency usually costs performance.
  Example: For balance checks, use stronger read concern than for browsing.

Q13) Explain `local`, `available`, `majority`, `linearizable`.

Answer:

- `local`: default; may read **uncommitted** data.
- `available`: very low consistency, prioritizes availability.
- `majority`: reads only **majority-committed** data.
- `linearizable`: **strongest**, but slowest.
  Example: Checkout uses majority/linearizable; homepage uses local.

Q14) Scenario: Why can reading from secondary with `local` be risky?

Answer:

- Secondary may lag → you can read stale data.
- `local` may allow reads before writes are majority committed.
- Use `majority` read concern when correctness matters.
  Example: User updates password but reads old profile from secondary—confusing bug.

Replica Set Deployment Variants

Q15) Typical production replica set deployment: what nodes and why?

Answer:

- Typical: **3 nodes** → 1 primary + 2 secondaries.
- Ensures quorum for elections and redundancy.

- Supports read scaling and safe write concerns.
  Example: If one node fails, still two nodes remain for majority.

Q16) What is an Arbiter? Why do people add it?

Answer:

- Arbiter **votes only** in elections and holds **no data**.
- Used to maintain an odd number of votes (avoid ties).
- It improves election reliability but does not improve durability (no data copy).
  Example: 2 data nodes + 1 arbiter gives 3 votes.

Q17) What is a Hidden node and why use it?

Answer:

- Hidden node is used for **analytics** without affecting primary performance.
- It can be kept out of normal read preference to avoid serving user traffic.
- Useful for heavy reporting queries.
  Example: Business intelligence dashboards run on hidden node.

Q18) What is a Delayed node and what problem does it solve?

Answer:

- Delayed node replicates with a time delay for **rollback protection**.
- Helps recover from accidental deletes/updates.
- Not used for real-time reads; it’s for safety.
  Example: Someone deletes orders table—delayed node still has old data from 1 hour ago.

Sharding (Horizontal Scaling)

Q19) What is sharding and when do we need it?

Answer:

- Used when data is too large for a single machine → split across multiple shards.
- It scales **storage** and **reads/writes** horizontally.
- Reduces load on any single node.
  Example: Social app with billions of users cannot fit on one server.

Q20) Name and explain the 3 core sharding components.

Answer:

- **mongos router**: lightweight query router that receives client queries and routes to correct shard(s).
- **Config servers (CSRS)**: store metadata like chunk locations; must be 3-node replica set.
- **Shards**: each shard is itself a replica set (scalability + availability).
  Example: Client connects to mongos, not directly to shards.

Q21) Explain shard key and why every sharded collection needs it.

Answer:

- Shard key determines **which shard stores which document**.
- It is used to route queries efficiently to a specific shard.
- Bad shard key causes hotspotting and scatter-gather queries.
  Example: Shard by `userId` so user profile queries hit one shard.

Q22) What are chunks? Mention default size and why chunks exist.

Answer:

- MongoDB breaks sharded data into **chunks**.
- Default chunk size is **64 MB**.
- Chunks can move across shards to balance load.
  Example: If Shard1 has too many chunks, some move to Shard2.

Q23) Explain balancer and “online chunk migration”.

Answer:

- Balancer monitors chunk distribution and detects imbalance.
- It moves chunks between shards when uneven.
- Migration happens **online** while cluster stays active.
  Example: Traffic continues while chunks shift in background.

Reads/Writes in Sharded Cluster

Q24) Write flow in sharded cluster: explain the full path.

Answer:

- Client sends write to **mongos**.
- Mongos routes to the correct shard based on shard key.
- Write goes to **primary of that shard's replica set**.
  Example: Insert `{userId:123}` routes only to shard that owns userId range/hash.

Q25) Read flow: shard key query vs non-shard-key query.

Answer:

- If query includes shard key, it targets **one shard** only.
- Without shard key, mongos broadcasts to all shards (**scatter-gather**).
- Scatter-gather is slower → this is why shard key matters.
  Example: `find({userId:123})` fast; `find({age:20})` hits all shards.

Q26) Explain "scatter-gather" in one perfect paragraph.

Answer:

- Scatter-gather means mongos must **query every shard** because it cannot know where the data is without shard key.
- Each shard does work and returns partial results; mongos merges them.
- It increases latency and load linearly with number of shards.
  Example: Searching "age=20" across billions of users becomes slow since it fans out everywhere.

Choosing a Good Shard Key

Q27) List 4 properties of a good shard key.

Answer:

- **High cardinality** (many unique values).
- **Evenly distributed** to avoid hotspots.
- **Queried often** so routing is targeted.
- Not **monotonically increasing** (or use hashed).
  Example: `userId` is great; `boolean` is terrible.

Q28) Why are monotonically increasing keys bad? What happens?

Answer:

- New inserts keep going to the same “end” of key range.
- This creates **hotspotting** where one shard gets most writes.
- Cluster becomes unbalanced even if you have many shards.
  Example: Shard key `timestamp` sends all new events to one shard → slows everything.

Q29) How does hashing the shard key help?

Answer:

- Hashing spreads inserts **randomly** across shards.
- Prevents monotonic hotspotting by breaking key order.
- Improves write distribution and parallelism.
  Example: Hash `email` or `userId` so new user signups distribute across shards.

Q30) Give 3 good shard key examples and 3 bad ones.

Answer:

- Good: `userId`, `emailHash`, `region + timestamp` (composite).
- Bad: `timestamp` (monotonic hotspot), `boolean` (low cardinality), `category with few values`.

- Reason: good keys distribute and are used in queries; bad keys cluster data unevenly.
  Example: Shard by `timestamp` = one shard burns; shard by `userId` = smooth.

Consistency Model (CP-like vs AP-like)

Q31) What is MongoDB's default consistency behavior in distributed setup?

Answer:

- Default writes: **primary only (w:1)**.
- Default reads: **primary (local)**.
- You can tune toward CP-like (stronger) or AP-like (more available).
  Example: Strong settings for financial systems; weaker for event tracking.

Q32) Explain: "Better consistency = slower; better performance = weaker guarantees."

Answer:

- Stronger read/write concerns require **more confirmations** or stricter visibility rules.
- That adds network + waiting time (latency).
- Weaker concerns return faster but may read stale/uncommitted data.
  Example: `w:majority` slows order insert but prevents losing orders.

Distributed Transactions (ACID across shards)

Q33) What kind of transactions does MongoDB support and since when?

Answer:

- Supports **multi-document ACID transactions across shards** from **v4.2**.
- Useful for atomic updates across collections/shards.
- But they are slower and should be avoided unless necessary.
  Example: Transfer money: debit in one doc and credit in another must be atomic.

Q34) Explain two-phase commit (2PC) in MongoDB distributed transactions.

Answer:

- Uses **two-phase commit** to ensure atomicity across shards.
- Guarantees: either **all operations succeed**, or **none are applied**.
- Extra coordination makes it slower than normal single-document writes.
  Example: Updating customer + order + inventory across shards must commit together.

Q35) What is “retryable writes” and why is it important?

Answer:

- Retryable writes allow clients to safely retry certain writes after network errors/failover.
- Helps avoid duplicate side effects when a response is lost but write may have happened.
- Especially useful during elections or transient disconnects.
  Example: Client times out during insert; retryable write prevents double insert.

Failover + Quorum in Sharded Clusters

Q36) What happens if one shard’s primary fails in a sharded cluster?

Answer:

- That shard’s replica set elects a new primary.
- Only that shard is temporarily degraded; other shards continue normally.
- System remains mostly available (partial impact).
  Example: Users mapped to shard2 may see delays; shard1 users are fine.

Q37) What happens if config servers fail? Why is quorum critical?

Answer:

- Config servers must maintain **quorum**.
- Without quorum, cluster **metadata access stops**, affecting routing and balancing.
- That’s why config servers are a 3-node replica set.
  Example: If 2 of 3 config servers die, mongos can’t safely route chunks → cluster breaks.

Backup + Monitoring

Q38) List 4 backup/restore options and when to use them.

Answer:

- `mongodump/mongorestore` for **small databases**.
- **Filesystem snapshots** for quick consistent backups.
- **Cloud Manager / Ops Manager** for managed backups.
- For sharded clusters: backup **each shard + config server**.
  Example: Atlas/Ops Manager schedules daily backups across all shards.

Q39) List monitoring tools from the lecture and what they show.

Answer:

- Tools: **mongostat**, **mongotop**, **Atlas metrics**, **Ops Manager dashboards**.
- They help track throughput, connections, latency, and system health.
- Monitoring is essential in distributed systems because failures/lag are common.
  Example: Use mongostat to spot replication lag spikes during heavy writes.

Q40) Name 5 critical metrics to watch in distributed MongoDB and explain why.

Answer:

- **Replication lag**: shows how behind secondaries are.
- **Chunk imbalance**: indicates uneven shard distribution.
- **Page faults**: memory pressure / slow disk access.
- **QPS** (queries/sec): workload intensity.
- **Connections / cache usage**: capacity + performance stability.
  Example: If replication lag grows, reading from secondaries becomes stale and risky.