0) What MongoDB actually is (foundation in 60 seconds)

MongoDB is a **document database**.

- Data is stored as **documents** (like JSON objects), grouped in **collections**.
- Internally MongoDB stores documents as **BSON (Binary JSON)**, which is faster and supports more types than plain JSON.

JSON vs BSON (why BSON matters)

- **JSON** = human readable, used in web APIs.
- **BSON** = binary format Mongo uses internally for speed + richer type support.

Biggest Mongo idea: “Embed instead of Join”

In SQL, you often **JOIN** tables. Mongo prefers **embedding** related data inside the same document to avoid joins.

Example from slide idea:

- SQL join: users + addresses
- Mongo: user document contains an array of addresses (embedded).

Exam line: “MongoDB reduces joins by embedding documents; referencing exists but no enforced foreign keys like SQL.”

---

1) Data modeling (embedding vs referencing) — this is asked a LOT

A) Embedding (put child inside parent)

Use when:

- You usually read the data together
- Child data belongs mostly to one parent
- Child list is not exploding infinitely

Example idea (books + authors embedded):
A book document contains authors array; query can search inside array fields.

B) Referencing (store IDs, like foreign keys)

Use when:

- Many-to-many
- Data reused by many parents
- Data changes frequently and you don't want duplication

Exam trap: Mongo allows referencing, but it **doesn't enforce** foreign keys.

Modeling scenario (very exam-likely)

"Book checkout system"

- One book can be checked out by one student at a time
- Student can check out many books

A clean Mongo approach:

- Put `checked_out` array inside student (embed checkout history), each item stores book_id + date.

Exam answer structure (3 points):

1. Why embed? faster reads, fewer joins
2. What's the risk? duplication / large document
3. When reference? many-to-many / shared entities

---

2) Indexing (highly likely short question)

What happens without index?

Mongo scans **every document** → slow.

With index

Mongo uses an index to jump directly to matching docs.

Your lecture lists what they can ask:

- single field index
- compound index
- multikey index

- `explain`
- compare with/without index

Exam-friendly points (memorize):

- Index = extra data structure that speeds reads
- Tradeoff = more storage + slower writes (because index updates)
- Use `explain()` to see if query used index and how efficient

---

3) Aggregation vs MapReduce (common “compare” question)

Aggregation pipeline (preferred)

Aggregation = operations that process records and return results.
Pipeline provides:

- filters like queries
- transformations for output
- grouping/sorting/array aggregation/operators

Examples from slides (understand meaning, no need to memorize full code):

- `$group` for totals/average
- `$match` filter then `$project` show selected fields
- `$sort` then `$limit` top k

MapReduce (older / heavier)

- map stage emits intermediate key-values
- reduce stage combines results
- optional finalize stage
- uses custom JavaScript functions, flexible but less efficient/complex
- can output bigger results than pipeline 16MB limit

Exam compare (3 points):

1. Pipeline simpler + faster; MapReduce more flexible but heavy
2. Pipeline is optimized inside MongoDB; MapReduce uses JS functions
3. MapReduce handles bigger outputs; pipeline has 16MB output limit

---

NOW THE MAIN EXAM PART: “Distributed MongoDB”

Your guideline says this is **most important**: primary/secondary, reading/writing, oplog, read/write concern, arbiter, sharding, load balancing.

4) Replication = Replica Set (high availability + redundancy)

What is replication?

A **replica set** = group of MongoDB servers that maintain the same data.

- **One Primary**: handles writes (and reads by default)
- **Secondaries**: replicate data from primary; can serve reads if allowed

Primary node job

- accepts writes
- sends changes to secondaries using **oplog**

Secondary node job

- replicates from primary using oplog
- can serve reads if read preference allows
- cannot accept writes unless promoted to primary

---

5) Oplog (the “log” that makes replication possible)

Oplog = operations log: an ordered record of changes on the primary.

Story:

1. Primary receives write (insert/update/delete)
2. Primary records it in oplog
3. Secondaries “tail” the oplog and apply the same operations

Exam line: “Secondaries replicate data from the primary using oplog.”

6) Automatic failover + Elections (very exam-likely scenario)

What happens when primary dies?

Slides give the flow:

- Secondaries detect failure using **heartbeats** (every 2 seconds)
- Election starts (Raft-like consensus)
- One secondary becomes new primary
- Clients auto-reconnect
- downtime about **2–5 seconds**

Extra election knobs (important terms):

- election timeout default 10s (`electionTimeoutMillis`)
- priority affects who becomes primary
- arbiters help maintain odd votes

Exam scenario answer (write this style):
"When primary fails, secondaries notice via heartbeats, trigger election, choose new primary based on voting/priority, and clients reconnect automatically."

---

7) Arbiter (aik aur node) — what it is and why we use it

Arbiter:

- participates in voting (helps quorum)
- **does not store data**
- **cannot become primary**
- used when you have even number of data-bearing nodes to avoid tie

Exam 3 points:

1. Adds a vote → helps elect a primary (quorum)
2. No data storage → cheaper, but not a backup
3. Cannot be primary → only decision-maker, not worker

---

8) Read preference vs Read concern (don’t mix!)

These sound similar but are different.

A) Read preference = “From where do I read?”

- Default reads are from **primary** (stronger consistency)
- You *can* read from secondary if read preference allows

B) Read concern = “How fresh/committed must the data be?”

From Lecture 15:

- `local` (default) may see uncommitted
- `available` low consistency
- `majority` only majority-committed data
- `linearizable` strongest, slowest

Story example (very exam):
If you read from a secondary right after a write, that secondary may not have caught up (replication lag). Using stronger read concern like `majority` ensures you only see data confirmed by majority nodes, but it may be slower.

---

9) Write concern = “How many nodes must confirm my write?”

From lecture:

- `w:1` → only primary confirms (fast, weaker safety)
- `w:majority` → most nodes confirm (safer)
- `w:0` → fire-and-forget (fastest, risky)

Exam scenario: Bank transfer

- Use `w:majority` because you cannot lose a confirmed write if primary crashes right after acknowledging.

---

10) Mongo consistency model (CP-like vs AP-like)

Lecture 15 says Mongo default:

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

And you can tune it:

- CP-like (stronger consistency)
- AP-like (more availability)
- tradeoff: better consistency = slower; better performance = weaker guarantees

Exam one-liner: "MongoDB can be tuned along the consistency–availability tradeoff using read/write concerns and read preference."

---

11) Sharding (the BIG second half)

What is sharding?

Sharding = **horizontal scaling**: split one big dataset across many machines.

Why?
When workloads become slow if:

- data > RAM
- data > single machine disk
- high write/transaction rate
- want linear scalability

Sharded cluster components (must know diagram words)

1) `mongos` (query router)

- lightweight router
- receives client queries
- routes to correct shard(s)

2) Config servers (CSRS)

- store cluster metadata (chunk locations)
- must maintain quorum; without quorum metadata access stops
- three-node replica set required

3) Shards

- each shard is itself a **replica set**
- gives scalability + availability

---

12) Shard key (super important exam topic)

Every sharded collection chooses a **shard key**; it decides which shard stores which document.

Good shard key properties (memorize)

- high cardinality (many unique)
- evenly distributed
- frequently used in queries
- not monotonically increasing (or use hashed)

Bad keys:

- timestamp (monotonic → hotspot)
- boolean (only two values)
- small-category field

Hotspotting (term sir LOVES)

Hotspotting = too many reads/writes hit same shard/chunk → that shard overloaded, others idle → performance imbalance.

---

13) Chunks + Balancer = load balancing

Mongo breaks data into **chunks** (default 64MB).
Chunks can move between shards based on load.

Balancer:

- background process
- monitors chunk distribution

- moves chunks when uneven
- migrations happen online while cluster stays active

Exam line: “Sharding scales; balancer ensures even distribution by moving chunks online.”

---

14) Read/write flow in sharded cluster (the scenario question)

Write flow

Client → `mongos` → correct shard → primary of that shard

Read flow depends on query

If query includes shard key → goes to only one shard (fast)
If query doesn’t include shard key → broadcast to all shards (**scatter-gather**, slow)

Exam tip sentence: “A good shard key avoids scatter-gather; otherwise queries hit all shards and become slow.”

---

15) Failover inside sharded clusters (short but important)

If one shard’s primary fails:

- that replica set elects new primary
- only that shard temporarily degraded
- other shards continue normally

If config servers fail:

- must maintain quorum
- without quorum cluster metadata access stops

---

16) Distributed transactions (advanced but examinable)

Mongo supports multi-document ACID transactions across shards (v4.2).
Works like **two-phase commit (2PC)** to ensure atomicity across docs/shards.
But it's slower; avoid unless necessary.

Also mentioned:

- oplog coordination
- retryable writes (safe retry after network error/primary failover)

Exam scenario:
"Updating two collections on different shards must be atomic" → use transactions (2PC), but mention performance tradeoff.

---

17) Backup + monitoring (easy marks)

Backup options:

- `mongodump/mongorestore` (small DB)
- filesystem snapshots
- Cloud Manager / Ops Manager
  For sharded cluster: backup each shard + config server

Monitoring tools:

- mongostat, mongotop
- Atlas metrics, Ops Manager dashboards
  Metrics:
- replication lag
- chunk imbalance
- page faults
- qps, connections, cache usage

High-yield "Sir-style" questions (with perfect answer skeletons)

Q1) Explain replica set and role of primary/secondary + oplog.

Answer skeleton (3 points + example):

- Replica set maintains same data for availability; one primary handles writes; secondaries replicate
- Primary records changes in **oplog** and secondaries apply them
- Example: if primary crashes, secondary becomes new primary via election → service continues

Q2) What is arbiter and why used?

- Votes in election (quorum)
- No data, cannot be primary
- Used to avoid tie (odd votes)

Q3) Define write concern and compare w:1 vs w:majority.

- Controls #nodes that confirm write
- w:1 fast but weaker
- w:majority safer but slower
  Example: banking uses majority.

Q4) Define read concern and compare local vs majority vs linearizable.

Use meanings from slide, mention tradeoff.

Q5) What is sharding and why needed?

Mention scale-out, data too big, high writes, linear scalability.

Q6) Explain sharded cluster architecture (mongos, config servers, shards).

Use exact roles.

Q7) What makes a good shard key? What is hotspotting?

Give properties + define hotspotting.

Q8) Explain scatter-gather and why it is slow.

When query lacks shard key, it broadcasts to all shards.

5-minute cram list (if time is dying)

- Replica set: primary writes, secondary replicates with **oplog**

- Failover: heartbeats → election → new primary in seconds
- Arbiter: vote only, no data
- Write concern: w:1 vs w:majority
- Read concern: local/majority/linearizable
- Sharding: mongos + config servers + shards (replica sets)
- Shard key: high cardinality, even distribution; avoid hotspotting
- Scatter-gather slow without shard key
- Chunks + balancer moves data online