1) What is MongoDB and why is it called a “document database”?

Answer:

- MongoDB stores data as **documents (JSON-like)** instead of rows/columns.
- Documents are stored inside **collections** (similar to tables, but schema-flexible).
- It supports **high performance + high availability + scaling** (esp. via replication/sharding).
  Example: A “user profile” document can contain name, city, skills array, nested address — all in one read.

---

2) Explain BSON. Why not plain JSON?

Answer:

- BSON is **Binary-encoded JSON-like** data format (faster to store/parse).
- Supports extra types like **ObjectId, Date, Timestamp, Binary**, etc.
- Efficient for indexing + traversal in DB engines.
  Example: Storing profile_picture as **binary** + createdAt as **Date** is natural in BSON.

---

3) What does “flexible schema” mean in MongoDB? Why is it useful in Big Data?

Answer:

- Different documents in same collection can have **different fields/structures**.
- New fields can appear **without migrations** (unlike rigid SQL).
- Great for **fast-changing data** like logs/IoT/clickstream.
  Example: Sensor readings may include temperature today, humidity tomorrow, pressure later.

---

4) Compare “collection” vs “table” (MongoDB vs SQL).

Answer:

- Table enforces a **fixed schema**; collection allows **schema variation**.
- SQL uses **joins** for relationships; MongoDB often uses **embedding** to avoid joins.
- Scaling in SQL often vertical; MongoDB supports **horizontal scaling** using sharding.
  Example: Orders + order_items: SQL join vs Mongo embedding items inside order doc.

---

5) Why are joins “expensive” in distributed systems, and how does MongoDB handle it?

Answer:

- Joins across machines mean **network hops + data shuffle** (slow).
- MongoDB uses **document model**: embed related data together.
- Result: **one read loads all relevant info**.
  Example: Customer + orders embedded → checkout page loads in one query.

---

6) Embedding vs Linking: when do you choose which?

Answer:

- **Embedding**: best when data is frequently read together (one read, fewer joins).
- **Linking (referencing)**: best when data is shared widely OR grows huge (avoid document bloat).
- MongoDB allows referencing but **does not enforce foreign keys**.
  Example: Embed “address” inside user; link “author” in many books via author_id.

---

7) Scenario: Library system — “one book can be checked out by one student at a time; a student can check out many books.” Model it in MongoDB.

Answer:

- Make **student document** with checked_out array (book_id + date).
- Ensures “one student → many books” naturally (array).
- To enforce “one book at a time”, store current_checkout in book OR maintain separate checkouts collection with unique constraint logic at app level.

Example: `student.checked_out: [{_id:"123", checked_out:"2012-10-15"}]`

---

8) What does a DB do “before indexing” when you query?

Answer:

- Without an index, MongoDB must **scan every document** (collection scan).
- This is inefficient for **large volumes** (Big Data scale).
- Query latency grows as data grows (bad scalability).
  Example: `find({score:{$lt:30}})` scans all users if no score index.

---

9) Define an index in MongoDB (perfect definition).

Answer:

- An index is a **special data structure** storing a small portion of collection data.
- Stored in an **easy-to-traverse form** to speed up lookups.
- Enables queries to avoid full collection scans.
  Example: Index on `score` lets DB quickly locate scores < 30.

---

10) Show the standard operations around indexes (create/show/drop).

Answer:

- Create: `db.users.ensureIndex({score:1})`
- Show: `db.users.getIndexes()`
- Drop: `db.users.dropIndex({score:1})`
  Example: Add index before running leaderboard queries on `score`.

---

11) What is `.explain()` and why is it exam-relevant?

Answer:

- Explain tells you how MongoDB executed the query (index used or not).
- Helps verify performance: **COLLSCAN vs IXSCAN** (conceptually).
- Used to justify why indexing improved query time.
  Example: `db.users.find({...}).explain()` proves score index is being used.

---

12) What is `hint()` and when is it dangerous?

Answer:

- Hint forces MongoDB to use a specific index.
- Useful for testing performance or when planner picks wrong index.
- Dangerous if you force a bad index → slower queries.
  Example: `find().hint({score:1})` forces score index even when not needed.

---

13) Explain single-field vs compound vs multikey indexes.

Answer:

- Single-field: index on one field (e.g., score).
- Compound: index on multiple fields (e.g., userid + score).
- Multikey: indexes elements inside arrays (powerful for tags/skills).
  Example: Search users by `(department, score desc)` → compound index.

---

14) Why does each collection have its “own index set”?

Answer:

- Indexes are built per collection because each collection has different query patterns.
- Separates performance tuning per dataset.
- Prevents unnecessary indexes on unrelated collections.
  Example: `users` needs email index, `logs` needs timestamp index.

---

15) What is an “aggregation pipeline”? Give the core pipeline stages.

Answer:

- Aggregation processes data through **stages** (pipeline).
- Key stages: `$match, $project, $group, $sort, $limit, $skip`.
- Advanced: `$lookup, $unwind, $facet, $bucket, $merge`.
  Example: Find top 5 cities by user count using `$group` + `$sort` + `$limit`.

---

16) Differentiate `$match` and `$project` using a scenario.

Answer:

- `$match` filters documents (like WHERE).
- `$project` selects/transforms fields (like SELECT + computed fields).
- Order matters: early `$match` reduces data, improves speed.
  Example: `$match: {city:"Karachi"}` then `$project:{name:1, city:1}`.

---

17) What does `$group` do and what operators are commonly used with it?

Answer:

- `$group` aggregates documents by key.
- Uses operators like `$sum, $avg, $min, $max`.
- Builds analytics inside DB (reduces need for external Spark/Hadoop for basic rollups).
  Example: Group sales by month: `$group` + `$sum(price)`.

---

18) Explain `$lookup` vs “embedding” (VERY common trick question).

Answer:

- `$lookup` performs a join between collections in pipeline.
- Embedding avoids join entirely (single read loads all info).
- In distributed Big Data, embedding is often faster; `$lookup` can be heavier.
  Example: Embed order items in order doc; use `$lookup` only when items are huge/shared.

---

19) What does `$unwind` do and why is it used?

Answer:

- `$unwind` "deconstructs" an array into multiple documents.
- Enables grouping/filtering per array element.
- Useful for analytics on embedded arrays.
  Example: An order with items array → unwind to count item frequencies.

---

20) Explain `$facet` in one exam-perfect answer.

Answer:

- `$facet` runs **multi-branch pipelines** in one query.
- Useful to compute multiple outputs simultaneously (e.g., stats + paginated results).
- Avoids multiple queries → faster + consistent snapshots.
  Example: Same dataset → one facet for "top cities", another for "average age".

---

21) What are update operators and why are they safer than replacing the whole document?

Answer:

- Operators like `$set, $unset, $inc` update only parts of doc.
- Prevent accidental overwriting of other fields.
- More efficient: smaller writes, less bandwidth.
  Example: `$inc` score by 1 instead of rewriting full user profile.

---

22) Explain `$setOnInsert` with upsert (common scenario).

Answer:

- With `upsert:true`, if doc doesn't exist → insert.
- `$setOnInsert` applies only on insert (not on update).
- Allows "create default fields once" while still updating others.
  Example: New user gets role=admin only at creation, not overwritten later.

---

23) Explain array update operators: `$push` vs `$addToSet`.

Answer:

- `$push` always adds element to array (duplicates allowed).
- `$addToSet` adds only if value doesn't already exist.
- Use `$addToSet` for unique lists (skills/tags).
  Example: Skills array: don't want "Python" repeated → use `$addToSet`.

---

24) What is a replica set? List its goals.

Answer:

- Replica set = **one primary + multiple secondaries**.
- Goals: **fault tolerance, automatic failover, read scalability, durability**.
- Core idea: keep copies so system survives node failure.
  Example: If primary crashes during exam season app usage, another becomes primary.

---

25) Explain how writes and reads work in a replica set.

Answer:

- Writes go to **primary only**.
- Primary replicates operations to secondaries via **oplog**.

- Reads by default from primary, but can read from secondaries if configured.
  Example: Analytics dashboard can read from secondary to reduce load on primary.

---

26) What is the oplog and why is it essential?

Answer:

- Oplog = operations log of writes (insert/update/delete) on each node.
- Secondaries **replay oplog** to become exact copies.
- Enables replication + consistency across nodes.
  Example: If primary writes `{name:"Ali"}`, secondaries replay same insert from oplog.

---

27) Describe elections + heartbeats in MongoDB failover (scenario-style).

Answer:

- Secondaries send **heartbeats** (detect primary failure).
- If primary dies, an **election** happens (Raft-like consensus).
- One secondary becomes new primary; clients reconnect automatically.
  Example: Primary server power-off → within seconds system recovers with new primary.

---

28) What is an arbiter and why does it exist?

Answer:

- Arbiter participates only in **voting** to maintain quorum.
- **Cannot be primary** and **does not store data**.
- Used to avoid ties when data-bearing nodes are even.
  Example: 2 data nodes + 1 arbiter → still odd votes for elections.

---

29) Explain a “delayed secondary”. Why is it important?

Answer:

- Secondary that replicates data with a **delay** (seconds/hours).
- Stores data (unlike arbiter) and protects against accidental deletes/updates.
- Acts like a “time machine” backup.
  Example: Admin accidentally deletes users at 2pm; delayed secondary still has 1pm state.

---

30) Why should replica set deployment usually have an odd number of members?

Answer:

- Elections require majority; odd avoids ties.
- Improves availability for quorum decisions.
- Helps stable failover (clear winner).
  Example: 3 nodes is better than 2 nodes (2 nodes can deadlock in split-brain scenarios).

---

31) Define write concern and explain w:1 vs w:majority vs w:0.

Answer:

- Write concern = how many nodes must confirm a write.
- `w:1`: only primary confirms (fast, weaker durability).
- `w:majority`: most nodes confirm (safer, slower).
  Example: Banking transaction uses `w:majority`; click tracking might use `w:1`.

---

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

Answer:

- Read concern decides what “version of data” you are allowed to read.

- `local`: may see uncommitted data (default, faster).
- `majority/linearizable`: stronger consistency, but slower.
  Example: Exam seat allocation needs strong reads; social feed can tolerate local.

---

33) What is sharding (definition + purpose)?

Answer:

- Sharding = split a collection across multiple machines (horizontal scaling).
- Enables automatic data distribution + load balancing.
- Used when TB/PB scale overwhelms single-machine DBs.
  Example: E-commerce orders across shards instead of one giant server.

---

34) What is `mongos` and why is it required in sharding?

Answer:

- `mongos` is the **query router**.
- Receives client requests and routes them to correct shard(s).
- Prevents clients from needing to know shard topology.
  Example: App sends query to `mongos`; mongos forwards to shard holding that user's data.

---

35) Define shard key and list 3 properties of a good shard key.

Answer:

- Shard key decides which shard stores which document.
- Good shard key has **high cardinality** (many unique values).
- Must avoid hotspotting; should distribute writes evenly.
  Example: Use userId or hashed email; avoid boolean field as shard key.

---

36) Explain "cardinality" in shard keys (exam wording).

Answer:

- Cardinality = number of distinct values a field can take.
- Higher cardinality → better distribution across shards.
- Low cardinality → few chunks → overloaded shards.
  Example: `gender` has low cardinality → terrible shard key; `userId` high cardinality → great.

---

37) What is hotspotting and why is it dangerous?

Answer:

- Hotspotting = too many reads/writes hit the same shard/chunk.
- Causes performance imbalance: one shard overloaded, others idle.
- Makes "sharding" useless because cluster behaves like bottlenecked single node.
  Example: Shard key = timestamp (monotonic) → newest inserts all land on one shard.

---

38) When do MongoDB workloads become slow (trigger for sharding)?

Answer:

- When data size becomes **greater than RAM**.
- When data exceeds a single machine's disk capacity.
- When write/transaction rate is very high and needs linear scalability.
  Example: Clickstream events per second spike → single server can't keep up → shard.

---

39) "MongoDB is suitable for Big Data." Justify with 3 reasons.

Answer:

- Document model reduces joins; one read loads relevant info.
- Flexible schema handles fast-changing data (IoT/logs).
- Sharding provides horizontal scaling + routing to correct shard only.
  Example: Social media posts with reactions/comments scale by sharding on userId.

40) Operators in MongoDB queries: what makes them powerful vs SQL WHERE at Big Data scale?

Answer:

- Operators work directly on BSON documents and nested fields.
- Can search arrays/objects/text/geospatial efficiently.
- Designed for distributed searches; huge-table WHERE becomes heavy without proper indexing/sharding.
  Example: Find users in Lahore/Karachi age>20 using `$and`, `$gt`, `$in`.