

MongoDB interview questions

26 questions · 9 easy · 12 medium · 5 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What is MongoDB? Explain documents, collections and BSON.

Easy



#basics #bson

MongoDB is a **document database**: it stores data as JSON-like documents instead of rows in tables.

- A **document** is one record: a set of field/value pairs that can contain nested objects and arrays.
- A **collection** is a group of documents, like a table, but with **no enforced schema** by default: documents can have different fields.
- A **database** holds collections.

Documents are stored as **BSON** (Binary JSON). It is faster to scan than text JSON and adds types JSON lacks: `ObjectId`, `Date`, `Decimal128`, 32/64-bit integers and binary data.

You can still enforce a structure with **schema validation** (`$jsonSchema`) or at the application level with Mongoose.

Example

```
db.users.insertOne({
  name: 'Asha',
  email: 'asha@x.io',
  joined: new Date(),
  address: { city: 'Pune', zip: '411001' }, // nested document
  skills: ['node', 'react'],              // array
})
db.users.findOne({ 'address.city': 'Pune' })
```

2

How is MongoDB different from a SQL database? When would you choose it?

Easy



#basics #design

Terms map roughly as table → collection, row → document, column → field, join → `$lookup` or embedding.

Key differences:

- **Schema:** flexible per document vs fixed columns.
- **Relations:** you usually **model data around how it is read**, embedding related data, instead of normalising and joining.
- **Scaling:** built-in sharding for horizontal scale; replica sets for availability.
- **Transactions:** single-document writes are always atomic; multi-document transactions exist (4.0+) but cost more.

Choose MongoDB for evolving or nested data, catalogues, content, events and fast iteration. Choose SQL for heavily relational data, complex ad-hoc reporting and many cross-entity transactions, like banking.

? **Likely follow-up:** Is MongoDB "schemaless"? What problems can that cause?

3

What is `_id` and what is inside an `ObjectId`?

Easy



#basics #objectid

Every document must have a unique `_id`, which is the primary key and is **indexed automatically**. If you do not supply one, the driver generates an **ObjectId**.

An ObjectId is 12 bytes:

- a 4-byte **timestamp** (seconds since the epoch)
- a 5-byte random value unique to the machine and process
- a 3-byte incrementing counter

So ids can be created on the client without asking the server, they are roughly sorted by creation time, and you can read the creation date with `getTimestamp()`.

`_id` can be any type except an array (a string, number or compound object), as long as it is unique and immutable.

Example

```
const id = new ObjectId()
id.getTimestamp() // ISODate of creation
db.orders.find().sort({ _id: -1 }).limit(5) // roughly the newest 5
```

4

Show the basic CRUD operations in MongoDB.

Easy



#crud

- **Create:** `insertOne`, `insertMany`
- **Read:** `find` (returns a cursor) and `findOne`, with a filter and an optional **projection** to choose fields
- **Update:** `updateOne`, `updateMany` with update operators, or `replaceOne` to swap the whole document
- **Delete:** `deleteOne`, `deleteMany`

Two details interviewers like:

- an update document needs operators like `$set`; without them you would replace the document
- `upsert: true` inserts the document if nothing matches

`findOneAndUpdate` updates and returns the document in one atomic step, which is useful for counters and queues.

Example

```
db.products.insertMany([{ name: 'Pen', price: 10, stock: 100 }, { name: 'Book', price: 250, stock: 5 }])
db.products.find({ price: { $lt: 100 } }, { name: 1, _id: 0 })
db.products.updateOne({ name: 'Pen' }, { $inc: { stock: -1 } })
db.products.updateOne({ name: 'Ink' }, { $set: { price: 40 } }, { upsert: true })
db.products.deleteMany({ stock: 0 })
```

5

What query operators do you use most?

Easy



#queries #operators

- **Comparison:** `$eq`, `$ne`, `$gt`, `$gte`, `$lt`, `$lte`, `$in`, `$nin`
- **Logical:** `$and` (implicit when you list several fields), `$or`, `$not`, `$nor`
- **Element:** `$exists`, `$type`
- **Array:** `$all`, `$size`, `$elemMatch`
- **Other:** `$regex`, `$expr` (compare two fields of the same document)

A common trap with arrays of objects: `{ 'items.qty': { $gt: 5 }, 'items.color': 'red' }` can match **two different array elements**. Use `$elemMatch` when both conditions must hold for the same element.

Dot notation (`'address.city'`) queries nested fields.

Example

```
db.users.find({ age: { $gte: 18, $lt: 30 }, role: { $in: ['admin', 'editor'] } })
db.users.find({ $or: [{ city: 'Pune' }, { verified: true }] })
db.users.find({ phone: { $exists: false } })
db.orders.find({ items: { $elemMatch: { qty: { $gt: 5 }, color: 'red' } } })
db.budgets.find({ $expr: { $gt: ['$spent', '$limit'] } })
```

6

Explain the main update operators.

Medium



#updates #operators

- `$set` / `$unset`: set or remove a field
- `$inc` / `$mul`: atomic arithmetic, ideal for counters
- `$min` / `$max`: update only if the new value is smaller or larger
- `$rename`, `$currentDate`
- Arrays: `$push` (with `$each`, `$slice`, `$sort`), `$addToSet` (push only if absent), `$pull` (remove matching), `$pop`
- Positional: `$` updates the first matched array element, `$[]` all elements, `$[id]` with `arrayFilters` the matching ones

These run atomically on the server, so `$inc` avoids the race of reading a value, adding one in your app and writing it back.

Example

```
db.posts.updateOne({ _id: id }, {
  $inc: { views: 1 },
  $addToSet: { tags: 'mongodb' },
  $push: { comments: { $each: [{ by: 'Ben', text: 'Nice' }], $slice: -50 } },
  $currentDate: { updatedAt: true },
})
db.orders.updateOne({ _id: id, 'items.sku': 'A1' }, { $set: {
  'items.$.qty': 3 } })
```

7

Embedding vs referencing: how do you decide?

Medium



#modeling #design

Embed related data inside the parent document when it is read together, belongs to the parent, and is bounded in size: an order's line items, a user's addresses. One read, and the write is atomic.

Reference (store the other document's `_id`) when:

- the related data is large or grows without limit (a post's millions of comments)
- it is shared by many parents (a product used in many orders)
- it is often read or updated on its own

Rule of thumb: "data that is accessed together should be stored together". Hybrids are common: reference the author but embed a copy of the author's name to avoid a lookup (accepting that it can go stale).

Example

```
// embedded: one read, atomic
{ _id: 1, customer: 'Asha', items: [{ sku: 'A1', qty: 2 }, { sku: 'B7',
qty: 1 }] }

// referenced: comments live in their own collection
{ _id: 'c91', postId: 1, by: 'Ben', text: 'Nice' }
```

🔗 Likely follow-up: What happens when an embedded array keeps growing?

8

How would you model a one-to-many relationship?

Medium 🔥🔥 #modeling #relationships

It depends on how many "many" there are:

- **One-to-few** (a user's 3 addresses): **embed** an array in the parent.
- **One-to-many** (a product's hundreds of parts): store an array of child `_id`s in the parent, or a parent id in each child.
- **One-to-squillions** (a server's log lines): put the parent's id in each child document (child referencing) and index it. Never keep an unbounded array in the parent.

Then ask which side you query from, and how often each side changes. For many-to-many, keep id arrays on one or both sides, or a separate linking collection.

Example

```
// one-to-squillions: child references parent
db.logs.insertOne({ hostId: ObjectId('...'), level: 'error', at: new
Date() })
db.logs.createIndex({ hostId: 1, at: -1 })
db.logs.find({ hostId: someId }).sort({ at: -1 }).limit(20)
```

9

Name some MongoDB schema design patterns.

Hard 🔥🔥 #modeling #patterns

- **Subset**: embed only the most-used part (the latest 10 reviews) and keep the rest in another collection.
- **Extended reference**: copy a few frequently read fields of a referenced document (author name) to avoid `$lookup`.
- **Computed**: store precomputed values (total, average rating) updated on write, instead of computing on every read.
- **Bucket**: group many small items into one document per time window (one document per sensor per hour), great for time series.
- **Outlier**: handle the rare huge case separately (a celebrity with millions of followers gets overflow documents).

- **Attribute:** turn many optional fields into an array of `{ k, v }` so one index covers them.
- **Schema versioning:** a `schemaVersion` field lets old and new shapes coexist during migrations.

The common idea: design for your read patterns.

10

What is the aggregation pipeline? Explain the common stages.

Medium 🔥🔥 #aggregation

The **aggregation pipeline** processes documents through a series of stages, each transforming the output of the previous one, like Unix pipes.

- `$match`: filter (like `WHERE`); put it first so it can use indexes
- `$group`: group by `_id` with accumulators like `$sum`, `$avg`, `$max`, `$push`
- `$project` / `$addFields`: reshape, rename or compute fields
- `$sort`, `$limit`, `$skip`
- `$unwind`: turn an array into one document per element
- `$lookup`: left outer join with another collection
- `$facet`, `$bucket`, `$count`, `$out` / `$merge` to save results

Each stage has a 100 MB memory limit unless `allowDiskUse` is on.

Example

```
// revenue per category for paid orders, top 5
db.orders.aggregate([
  { $match: { status: 'paid' } },
  { $unwind: '$items' },
  { $group: { _id: '$items.category', revenue: { $sum: { $multiply:
['$items.price', '$items.qty'] } } }, orders: { $sum: 1 } } },
  { $sort: { revenue: -1 } },
  { $limit: 5 },
  { $project: { _id: 0, category: '$_id', revenue: 1, orders: 1 } } },
])
```

🔗 Likely follow-up: Why should `$match` come before `$group`?

11

How do you "join" collections with `$lookup`, and what does `$unwind` do?

Medium



#aggregation #joins

`$lookup` performs a **left outer join**: for each input document it finds matching documents in another collection and puts them in a new **array field**.

Basic form: `localField`, `foreignField`, `from` and `as`. The pipeline form (`let` + `pipeline`) allows extra conditions and projections inside the join.

`$unwind` deconstructs an array into one document per element. After a `$lookup` that should return a single match, `$unwind` turns the one-element array into a plain object. Use `preserveNullAndEmptyArrays: true` to keep documents with no match.

Make sure `foreignField` is indexed. If you `$lookup` on every request, reconsider the schema (embed or extended reference).

Example

```
db.orders.aggregate([
  { $lookup: { from: 'users', localField: 'userId', foreignField: '_id',
    as: 'user' } },
  { $unwind: { path: '$user', preserveNullAndEmptyArrays: true } },
  { $project: { total: 1, 'user.name': 1, 'user.email': 1 } },
])
```

12

What types of indexes does MongoDB support?

Medium



#indexes #performance

Indexes are **B-trees** that let queries avoid a collection scan (`COLLSCAN`).

- **Single field:** `{ email: 1 }`; direction does not matter for one field.
- **Compound:** `{ status: 1, createdAt: -1 }`; order matters (prefix rule).
- **Multikey:** created automatically when the field is an array; one entry per element.
- **Text:** full-text search with `$text` (one per collection; Atlas Search is stronger).
- **Geospatial:** `2dsphere` for location queries.
- **Hashed:** for hashed shard keys.
- **TTL:** **deletes documents automatically** after a time on a date field (sessions, OTPs).
- Options: `unique`, `partial` (index only matching documents), `sparse`.

Every index slows writes and uses RAM, so index for your real queries only.

Example

```
db.users.createIndex({ email: 1 }, { unique: true })
db.orders.createIndex({ userId: 1, createdAt: -1 })
db.articles.createIndex({ title: 'text', body: 'text' })
db.sessions.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 })
// TTL
db.users.createIndex({ phone: 1 }, { partialFilterExpression: { phone: {
  $exists: true } } })
```

13

What is the ESR rule for compound indexes?

Hard



#indexes #performance

ESR is the recommended order of fields in a compound index: **Equality, Sort, Range**.

- **Equality** fields first (`status: 'paid'`): they narrow to one contiguous part of the index.
- **Sort** fields next (`createdAt`): inside that part the index is already sorted, so no in-memory sort is needed.
- **Range** fields last (`total: { $gt: 100 }`): ranges break the ordering of the fields after them.

For `find({ status: 'paid', total: { $gt: 100 } }).sort({ createdAt: -1 })` the best index is `{ status: 1, createdAt: -1, total: 1 }`.

A compound index also serves queries on its **prefixes** (`status` alone), but not on later fields alone.

14

How do you check whether a query uses an index?

Medium



#performance #explain

Run `.explain('executionStats')` on the query (or the aggregation).

What to read:

- **winningPlan stage:** `IXSCAN` (index used) vs `COLLSCAN` (full scan); `SORT` means an in-memory sort; `FETCH` means it went to the documents

- **nReturned** vs **totalKeysExamined** vs **totalDocsExamined**: ideally all close. Examining 100 000 documents to return 10 means a missing or poor index
- **executionTimeMillis**

A **covered query** shows no `FETCH` and `totalDocsExamined: 0`: the projection only uses indexed fields and excludes `_id`.

In production, use the **profiler** or Atlas Performance Advisor to find slow queries, and `$indexStats` to find unused indexes.

Example

```
db.orders.find({ userId: uid }).sort({ createdAt: -1
}).explain('executionStats')
// look at: queryPlanner.winningPlan.stage,
executionStats.totalDocsExamined, nReturned
```

15

What is a replica set and how does failover work?

Medium



#replication #architecture

A **replica set** is a group of `mongod` servers holding the same data: one **primary** takes all writes, and **secondaries** copy its **oplog** (operation log) and apply it.

If the primary stops responding, the remaining members hold an election and promote a secondary, usually within seconds. Drivers discover the new primary and retry writes automatically (retryable writes).

- Use an **odd number** of voting members (3 is typical) so a majority can be reached; an **arbiter** votes but holds no data.
- Secondaries can serve reads (read preference), but they may lag.
- Replication gives high availability and backups, not more write capacity; that is sharding.

Likely follow-up: Why do you need a majority for an election?

16

What is sharding and how do you choose a shard key?

Hard



#sharding #scaling

Sharding splits a collection across several replica sets (**shards**) so data and write load scale horizontally. Apps connect to **mongos** routers, which use metadata from the **config servers** to send each query to the right shard(s). Data is divided into chunks by the **shard key**, and a balancer moves chunks between shards.

A good shard key has:

- **high cardinality** (many distinct values)
- **even distribution** (no single hot value)
- **query isolation**: most queries include it, so they hit one shard instead of a scatter-gather to all

Bad: a monotonically increasing key like a timestamp or default **_id** in a range key, because **every insert goes to the last shard**. Options: a hashed key, or a compound key such as `{ customerId: 1, createdAt: 1 }`.

17

Does MongoDB support transactions?

Medium



#transactions

Yes, and two levels matter:

- **Single-document** operations are **always atomic**, even when they change many fields or nested arrays. Good schema design (embedding) means most writes never need more than this.
- **Multi-document ACID transactions** have been available since 4.0 on replica sets and 4.2 on sharded clusters. You start a session, run operations with it, then commit or abort.

Costs and limits: they are slower, hold locks, have a default 60-second runtime limit, and should touch a modest number of documents. If you need them everywhere, your data may be a better fit for a relational database or for a different schema.

Example

```
const session = db.getMongo().startSession()
const accounts = session.getDatabase('bank').accounts
session.startTransaction({ writeConcern: { w: 'majority' } })
try {
  accounts.updateOne({ _id: 'A' }, { $inc: { balance: -100 } })
  accounts.updateOne({ _id: 'B' }, { $inc: { balance: 100 } })
  session.commitTransaction()
} catch (e) {
  session.abortTransaction()
}
```

18

What are write concern, read concern and read preference?

Hard



#replication #consistency

Write concern says how many members must acknowledge a write before it counts as successful:

- `w: 1`: the primary only (fast; can be lost on failover)
- `w: 'majority'`: a majority of members, **safe across failover** (default since 5.0)
- `j: true`: also written to the on-disk journal

Read concern controls what data a read may see: `local` (latest on that node), `majority` (data that cannot be rolled back), `snapshot` (inside transactions).

Read preference says which member serves reads: `primary` (default), `primaryPreferred`, `secondary`, `secondaryPreferred`, `nearest`. Reading from secondaries spreads load but may return stale data.

19

What is Mongoose and why use it?

Easy



#mongoose #node

Mongoose is an ODM (Object Document Mapper) for Node.js on top of the MongoDB driver. It adds what MongoDB leaves to the application:

- **Schemas** with types, `required`, defaults, enums and custom validators
- **Models** with a clean query API (`User.find()`, `User.create()`)
- **Middleware** (hooks) such as `pre('save')`
- `populate()` for references

- virtuals, instance and static methods, `timestamps: true`

The trade-off: an extra layer, some overhead, and validators that do not run on updates by default (pass `runValidators: true`). For simple scripts or maximum performance, the native driver is fine.

Example

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true, trim: true },
  email: { type: String, required: true, unique: true, lowercase: true },
  role: { type: String, enum: ['user', 'admin'], default: 'user' },
}, { timestamps: true })

const User = mongoose.model('User', userSchema)
const u = await User.create({ name: 'Asha', email: 'ASHA@x.io' })
```

20

What is Mongoose middleware? Give a real example.

Medium



#mongoose #middleware

Middleware (hooks) are functions that run before or after certain operations: `pre` and `post` hooks on `save`, `validate`, `deleteOne`, query operations like `find` and `findOneAndUpdate`, and `aggregate`.

Common uses:

- hashing a password in `pre('save')`
- setting audit fields, slugs or soft-delete filters
- cleaning up related documents after delete
- logging

Gotchas:

- in **document** middleware `this` is the document; in **query** middleware `this` is the query
- `save` hooks do **not** run for `updateOne` or `findOneAndUpdate`
- use a regular `function`, not an arrow function, so `this` works

Example

```
userSchema.pre('save', async function () {
  if (!this.isModified('password')) return
  this.password = await bcrypt.hash(this.password, 12)
})

userSchema.pre(/^find/, function () {
  this.where({ deletedAt: null }) // hide soft-deleted users
})
```

21

What does `populate()` do in Mongoose, and what are its downsides?

Medium



#mongoose #relationships

`populate()` replaces a stored reference (an `ObjectId` with `ref: 'User'`) with the actual referenced document.

Under the hood it is **not a database join**: Mongoose runs an extra query, `find({ _id: { $in: ids } })` on the referenced collection, and stitches the results together in Node. It batches ids, so it is not N+1 per document, but each populated path is another round trip.

Tips:

- select only needed fields: `populate('author', 'name avatar')`
- nested populate is possible but gets slow
- for heavy or filtered joins, use an aggregation with `$lookup`, or embed the fields you always need

Example

```
const postSchema = new mongoose.Schema({
  title: String,
  author: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
})

const posts = await Post.find().populate('author', 'name email').lean()
```

22

What does `.lean()` do in Mongoose?

Easy



#mongoose #performance

By default Mongoose turns every result into a full **Mongoose document**, with change tracking, getters and setters, virtuals and methods like `save()`. That costs memory and CPU.

`.lean()` returns **plain JavaScript objects** straight from the driver. Queries are noticeably faster and lighter, which matters for read-only endpoints returning lists.

What you lose: `save()`, virtuals, getters, defaults applied on read, and document middleware. Use `lean()` when you only send data to the client, and normal documents when you will modify and save them.

23

How do you paginate efficiently in MongoDB?

Medium



#pagination #performance

Offset pagination: `find().sort({ _id: -1 }).skip((page - 1) * size).limit(size)`.

Simple, and it supports jumping to page N, but `skip` still walks over all skipped documents, so **deep pages get slow**, and inserts can shift items between pages.

Cursor (keyset) pagination: remember the last item's sort value and ask for items after it: `find({ _id: { $lt: lastId } }).sort({ _id: -1 }).limit(size)`. It uses the index, so every page costs the same. Ideal for infinite scroll and APIs. For a non-unique sort field, add `_id` as a tie-breaker.

For a total count, use `countDocuments` (or `$facet` to get data and count in one aggregation), knowing it can be expensive on large collections.

Example

```
// page 1
const page = db.posts.find({}).sort({ _id: -1 }).limit(20).toArray()
// next page: continue after the last id we saw
const lastId = page[page.length - 1]._id
db.posts.find({ _id: { $lt: lastId } }).sort({ _id: -1 }).limit(20)
```

24

What is the maximum document size, and how do you deal with it?

Easy



#limits #modeling

A single BSON document can be at most **16 MB**. Nesting is limited to 100 levels.

The limit exists to stop documents growing endlessly, which hurts memory and network use. In practice you hit it with an **unbounded array**: comments, logs or followers pushed into one document forever.

Fixes:

- reference instead of embed (put the parent id in the child)
- the **bucket** or **outlier** pattern to spread items over several documents
- **GridFS** for files bigger than 16 MB: it splits them into 255 KB chunks in `fs.chunks` with metadata in `fs.files` (though object storage like S3 is often a better home for files)

25

What is a capped collection?

Easy



#collections

A **capped collection** has a fixed maximum size (bytes, and optionally document count). Documents are stored in insertion order, and when it is full **the oldest documents are overwritten** automatically, like a circular buffer.

Good for: recent logs, caches of the latest events, and queues. You can follow it with a **tailable cursor**, which stays open and receives new documents as they arrive (this is how the oplog works).

Limits: you cannot delete individual documents or grow a document's size, and capped collections cannot be sharded. For time-based expiry use a **TTL index** instead; for time series, MongoDB 5.0+ has native time-series collections.

Example

```
db.createCollection('recentLogs', { capped: true, size: 10 * 1024 * 1024,
max: 5000 })
db.recentLogs.find().sort({ $natural: -1 }).limit(10) // newest 10
```

How would you improve a slow MongoDB-backed API?

Hard

#performance

Work from evidence, then fix the biggest cost first:

- **Find the slow queries:** the profiler, slow query logs, Atlas Performance Advisor.
- **Explain them:** replace `COLLSCAN` with the right compound index (ESR rule); remove unused indexes that slow writes.
- **Return less:** projections, `lean()` in Mongoose, cursor pagination instead of big `skip`.
- **Fix the schema** for the read pattern: embed or use extended references to avoid repeated `$lookup` / `populate`, precompute totals (computed pattern).
- **Aggregations:** `$match` and `$project` early so less data flows through.
- **Working set in RAM:** indexes and hot data should fit in memory.
- **Cache** hot reads (Redis), and use read replicas for analytics.

Mention that you would **measure before and after** each change.