

MongoDB & Mongoose

Documents, schema design, queries, aggregation, indexes

1 Documents and collections

Definition

MongoDB is a document database: JSON-like documents (BSON) in collections, with a flexible schema, rich queries, aggregation pipelines and horizontal scaling (sharding).

SQL	MongoDB
database	database
table	collection
row	document
column	field
JOIN	<code>\$lookup</code> / embedding
primary key	<code>_id</code> (ObjectId)

One order document

```
{
  _id: ObjectId("665f..."),
  customerId: ObjectId("664a..."),
  status: "paid",
  items: [                                // embedded: always read with the
order
    { sku: "KB-01", name: "Keyboard", qty: 1, price: 400 },
    { sku: "MS-02", name: "Mouse", qty: 2, price: 300 }
  ],
  total: 1000,
  createdAt: ISODate("2026-10-19T09:00:00Z")
}
```

2

Embed or reference?

<i>Embed when...</i>	<i>Reference when...</i>
data is read together (order + items)	data is shared by many documents (a product)
one-to-few, bounded size	one-to-many unbounded (comments, logs)
the child has no life of its own	the child is queried on its own

- ✎ "Design for your queries": model documents around how the app reads them.
- ✎ Document size limit: 16 MB — never grow arrays without bound.
- ✎ Duplicate small, rarely changing fields (customer name on an order) to avoid lookups.

CRUD with operators

```
db.orders.insertOne({ customerId, items, total: 1000, status: 'pending' })

db.orders.find(
  { status: { $in: ['paid', 'shipped'] }, total: { $gte: 500 } },
  { items: 0 } // projection: hide items
).sort({ createdAt: -1 }).limit(20)

db.orders.updateOne(
  { _id: id, status: 'pending' }, // condition = optimistic guard
  { $set: { status: 'paid' }, $inc: { version: 1 }, $push: { history: { at:
new Date(), to: 'paid' } } }
)

db.products.updateOne({ sku: 'KB-01', stock: { $gte: 1 } }, { $inc: { stock:
-1 } }) // atomic decrement
db.orders.deleteMany({ status: 'cancelled', createdAt: { $lt: cutoff } })
```

- Comparison: `$eq` `$ne` `$gt` `$gte` `$lt` `$lte` `$in` `$nin`; logic: `$and` `$or` `$not`; arrays: `$elemMatch`, `$all`, `$size`.
- Update operators: `$set` `$unset` `$inc` `$push` `$pull` `$addToSet`; `upsert: true` inserts if missing.
- Single-document updates are atomic; multi-document changes need a **transaction** (replica set).

Common mistake

NoSQL injection: passing `req.body` straight into a query lets `{ "$gt": "" }` match everything. Validate types, and use Mongoose `sanitizeFilter` or strip keys starting with `$`.

4

Aggregation pipeline

\$match

→

\$lookup

→

\$unwind

→

\$group

→

\$sort

→

\$project

→

\$limit

Each stage transforms the documents and passes them on

Revenue per customer with names

```
db.orders.aggregate([
  { $match: { status: 'paid', createdAt: { $gte: ISODate('2026-01-01') } } },
  // filter early (uses indexes)
  { $group: { _id: '$customerId', revenue: { $sum: '$total' }, orders: { $sum:
1 } } },
  { $sort: { revenue: -1 } },
  { $limit: 10 },
  { $lookup: { from: 'customers', localField: '_id', foreignField: '_id', as:
'customer' } },
  { $unwind: '$customer' },
  { $project: { _id: 0, name: '$customer.name', revenue: 1, orders: 1 } },
])
```

Indexes and scaling

- ✎ Every collection has an `_id` index; add indexes for your query filters and sorts.
- ✎ Compound index field order: Equality → Sort → Range (the ESR rule).
- ✎ Special indexes: unique, TTL (auto-expire sessions), text, geospatial, partial.
- ✎ Check with `.explain("executionStats")`: you want IXSCAN, not COLLSCAN.
- ✎ **Replica set** = copies for high availability (one primary, secondaries). **Sharding** = splitting data across machines by a shard key.

Schema, validation, populate

```
import mongoose from 'mongoose'

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: ['admin', 'user'], default: 'user' },
}, { timestamps: true })

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

const User = mongoose.model('User', userSchema)
const posts = await Post.find({ published: true })
  .populate('author', 'name')           // reference → document
  .lean()                               // plain objects: faster for
reads
```

Revision

Model documents around queries: embed what is read together and bounded, reference what is shared or unbounded. Use update operators for atomic changes, aggregation pipelines for reports (match early), ESR indexes, and sanitise query input.

Q. SQL vs MongoDB?

A. SQL uses fixed-schema tables with joins and strong transactions; MongoDB stores flexible JSON-like documents, favours embedding, and scales horizontally with sharding.

Q. When do you embed vs reference?

A. Embed data read together that is bounded (order items); reference data that is shared or grows without limit (comments, products).

Q. What is the aggregation pipeline?

A. A sequence of stages (`$match`, `$group`, `$lookup`, `$project`, `$sort...`) that transform documents, like a SQL query built in steps.

Q. What is `$lookup`?

A. A left outer join to another collection inside an aggregation pipeline.

Q. What is the ESR rule?

A. Order compound index fields Equality first, then Sort, then Range.

Q. Are MongoDB updates atomic?

A. Yes for a single document; multi-document changes need a transaction on a replica set.

Q. Replica set vs sharding?

A. A replica set copies the same data for availability; sharding splits data across servers for scale.

Q. What does `populate` do in Mongoose?

A. Replaces a referenced `ObjectId` with the referenced document by running an extra query.

Q. How do you prevent NoSQL injection?

A. Validate input types and reject or strip keys beginning with `$` (e.g. Mongoose `sanitizeFilter`).

Q. What does `.lean()` do?

A. Returns plain JavaScript objects instead of Mongoose documents — faster and lighter for read-only queries.