MongoDB Cheat Sheet

MongoDB 8.x · mongosh / Query API · mongodb.com/docs/manual

RUN & CONNECT

```
mongod --dbpath /data/db --port 27017
mongod --config /etc/mongod.conf   # YAML
mongosh "mongodb://localhost:27017"
```

Connection string

```
mongodb://user:pw@h1:27017,h2:27017/mydb?
replicaSet=rs0&authSource=admin
mongodb+srv://user:pw@cluster.mongodb.net/?
retryWrites=true&w=majority
```

Shell: `show dbs` · `use mydb` · `show collections` · `it` (next batch) · `load("f.js")`

DOCUMENT & BSON TYPES

Field/value pairs, binary **BSON**; max **16 MB**/doc, ~100 nesting levels.

```
Double String Object Array BinData ObjectId
Bool Date Null Regex Int32 Int64 Decimal128
Timestamp MinKey MaxKey
```

_id unique per doc, auto `ObjectId` = 4-byte time + 5 random + 3 counter (roughly increasing).

```
ObjectId("6512...")  db.c.insertOne({_id:
"sku-1", ...})
```

Dot notation: `"addr.city"`, `"tags.0"`. Extended JSON: `{"$oid":..},{"$date":..},{"$numberDecimal":..}`

INSERT & WRITE CONCERN

```
db.c.insertOne({ x: 1 })
db.c.insertMany([{...},{...}], { ordered:
false })
db.c.bulkWrite([
  { insertOne:  { document: {...} } },
  { updateOne:  { filter:{...}, update:{
$set:{...} } } },
  { deleteOne:  { filter:{...} } }
], { ordered: true })
```

Write concern `{ w, j, wtimeout }`: `w:1` (primary) · `w:"majority"` · `j:true` (journal).

Retryable writes: driver retries once on primary failover; single writes stay idempotent.

FIND — FILTER & PROJECTION

```
db.c.find({ status: "A", qty: { $gte: 10 } },
          { item: 1, qty: 1, _id: 0 })
db.c.findOne({ _id: 1 })
```

Implicit equality + implicit `$and`. Projection: `1` include / `0` exclude, `$slice`, `$elemMatch`, `$`.

Operators

```
$eq $ne $gt $gte $lt $lte $in $nin
$regex:/^a/i   $exists:true   $type:"string"
$mod:[4,0]
$elemMatch:{ $gt:5, $lt:20 }   $all:[a,b]
$size:3
$expr:{ $gt:["$a","$b"] }
```

Logical `$and $or $not $nor`. Array: match element, `"arr.0.f"`, `$all`, `$size`.

CURSOR METHODS

```
db.c.find(q).sort({ ts: -1
}).skip(20).limit(10)
db.c.find(q).hint({ ts: 1 }).collation({
locale:"en", strength:2 })
db.c.countDocuments(q)
db.c.distinct("field", q)
```

Avoid large `skip` — page with a range filter on the last seen key.

UPDATE & DELETE

```
db.c.updateOne({ _id:1 }, { $set:{ a:1 },
$inc:{ n:1 } })
db.c.updateMany({ s:"old" }, { $set:{ s:"new"
} })
db.c.replaceOne({ _id:1 }, { a:1, b:2 })
db.c.updateOne(f, [{ $set:{ full:{ $concat:
["$first"," ","$last"] } } }])  // pipeline
```

Field `$set $unset $rename $inc $mul $min $max $currentDate $setOnInsert`

Array `$push` (+`$each $slice $sort $position`) `$addToSet $pop $pull $pullAll`

```
{ $push: { tags: { $each:["x","y"],
$slice:-10 } } }
{ $set: { "items.$.qty": 0 } }          //
positional, needs match in filter
{ $inc: { "items.$[].n": 1 } }          //
all elements
{ $set: { "items.$[e].done": true } }, {
arrayFilters:[{ "e.id": 42 }] }
```

Upsert `{ upsert:true }`: filter equalities + `$setOnInsert` seed the new doc.

```
db.c.findOneAndUpdate(f, u, {
returnDocument:"after" })
db.c.deleteOne(f)   db.c.deleteMany(f)
db.c.drop()
```

INDEXES

```
db.c.createIndex({ a: 1 })
// single
db.c.createIndex({ a: 1, b: -1 })
// compound
db.c.createIndex({ "loc": "2dsphere" })
db.c.createIndex({ tags: "text" })
// one text idx / coll
db.c.createIndex({ k: "hashed" })
db.c.getIndexes()   db.c.dropIndex("a_1")
```

Types: single · compound · multikey (auto on arrays, one array field/index) · text · 2dsphere · hashed · TTL · wildcard.

Properties `unique:true` · `partialFilterExpression` · `sparse` · `expireAfterSeconds` (TTL) · `hidden` · `collation`.

ESR rule — compound key order: **E**quality → **S**ort → **R**ange.

```
db.c.find(q).explain("queryPlanner" |
"executionStats" | "allPlansExecution")
```

IXSCAN > COLLSCAN; watch `totalKeysExamined` vs `nReturned`; covered query = all fields in index.

AGGREGATION PIPELINE

```
db.c.aggregate([
  { $match:  { status: "A" } },
  { $group:  { _id: "$cust", total: { $sum:
"$amt" }, n: { $sum: 1 } } },
  { $sort:   { total: -1 } },
  { $limit:  10 },
  { $merge:  { into: "rollup" } }
])
```

Stages `$match $project $addFields/$set $unset $group $sort $limit $skip $unwind $lookup $unionWith $facet $bucket/$bucketAuto $graphLookup $replaceRoot $sample $setWindowFields $out $merge $count`

Accumulators `$sum $avg $min $max $push $addToSet $first $last $count $mergeObjects $top $bottom`

Paths `"$field"` · vars `$$ROOT $$NOW $$REMOVE` · `$cond $switch $ifNull`. `$expr` runs expressions inside `find`. Map-reduce is deprecated.

TRANSACTIONS

Single-doc writes are always atomic — use a transaction only for a multi-doc invariant. Needs a replica set / sharded cluster.

```
const s = db.getMongo().startSession()
s.withTransaction(() => {
  const a = s.getDatabase("bank").accounts
  a.updateOne({ _id: 1 }, { $inc: { bal: -100
} })
  a.updateOne({ _id: 2 }, { $inc: { bal:  100
} })
}, { readConcern: { level: "snapshot" },
     writeConcern: { w: "majority" } })
s.endSession()
```

Keep short: 60 s default runtime, 16 MB oplog entry. Same API on sharded clusters (extra cost).

REPLICA SET

```
rs.initiate({ _id:"rs0", members:[
  { _id:0, host:"h1:27017" },
  { _id:1, host:"h2:27017" },
  { _id:2, host:"h3:27017" } ] })
rs.add("h4:27017")   rs.addArb("h5:27017")
rs.remove("h4:27017")
rs.status()   rs.conf()   rs.stepDown()
```

One **primary** + **secondaries** tailing the **oplog**; majority elects a new primary on failover.

Read preference `primary | primaryPreferred | secondary | secondaryPreferred | nearest`

Write concern `w: 1 | "majority" | <n> | <tag>`, `wtimeout`. **Read concern** `local | available | majority | linearizable | snapshot`.

SHARDING

```
sh.enableSharding("shop")
sh.shardCollection("shop.orders", { custId:
1, _id: 1 })   // ranged
sh.shardCollection("shop.events", { _id:
"hashed" })        // hashed
sh.status()   sh.getBalancerState()
sh.startBalancer()
```

App → `mongos` routers → shards (each a replica set) + config-server replica set. Data split into **chunks**; the **balancer** spreads them.

Shard key: high cardinality, low frequency, non-monotonic (avoid the ascending-key hotspot — hash it).

Key can be **refined**; a collection can be **resharded**. Use **zones** for locality.

BACKUP & DATA TOOLS

```
mongodump   --uri="mongodb://localhost" --
out=dump/
mongorestore --uri="mongodb://localhost"
dump/
mongoexport --collection=c --out=c.json
# or --type=csv --fields=a,b
mongoimport --collection=c --file=c.json --
mode=upsert
```

Also: filesystem/volume snapshots (PITR), Atlas/Ops Manager continuous backup. Sharded backup needs the balancer stopped. `bsondump`, `mongofiles` (GridFS > 16 MB).

EMBED VS. REFERENCE

- **Embed** when: read together, 1:1 or 1:few, bounded array, updated together.
- **Reference** when: queried on its own, 1:many / many:many, unbounded array (16 MB ceiling), large or volatile sub-entity.
- Denormalize hot read paths; accept the duplicate-update cost. Version schemas with a `schemaVersion` field.

Verified against the MongoDB Server Manual (mongodb.com/docs/manual). Companion reference: *MongoDB: The Definitive Guide*, 3rd ed. (O'Reilly, 2019) — bibliography only.