

● STEP 4: find() WITH CONDITIONS (FILTERING DATA)

Think of this as **asking questions to your data**.

◆ 1. Basic idea (VERY IMPORTANT)

Syntax:

```
db.collection.find({ condition })
```

The condition is written as a **JSON object**.

◆ 2. Equality condition (MOST COMMON)

Situation:

👉 “Give me students whose branch is CSE”

```
db.students.find({ branch: "CSE" })
```

SQL equivalent:

```
SELECT * FROM students WHERE branch = 'CSE';
```

✓ Exact match

✓ Case-sensitive

◆ 3. Number condition

Situation:

👉 “Give students whose age is 20”

```
db.students.find({ age: 20 })
```

✖ Numbers are **NOT in quotes**

✗ "20"

✓ 20

◆ 4. Multiple conditions (AND)

Situation:

👉 “Students who are CSE **AND** age is 21”

```
db.students.find({
```

```
branch: "CSE",
```

```
age: 21
```

```
)
```

 MongoDB treats multiple fields as **AND**

◆ 5. Comparison operators (VERY IMPORTANT)

MongoDB uses **operators with \$**

► Greater than (\$gt)

 “Age greater than 20”

```
db.students.find({
```

```
    age: { $gt: 20 }
```

```
)
```

► Greater than or equal (\$gte)

```
db.students.find({
```

```
    age: { $gte: 20 }
```

```
)
```

► Less than (\$lt)

```
db.students.find({
```

```
    age: { $lt: 22 }
```

```
)
```

► Less than or equal (\$lte)

```
db.students.find({
```

```
    age: { $lte: 21 }
```

```
)
```

◆ 6. Range condition (BETWEEN)

Situation:

👉 "Students aged between 20 and 22"

```
db.students.find({  
    age: { $gte: 20, $lte: 22 }  
})
```

🧠 Multiple operators can be used on same field.

◆ 7. NOT EQUAL (\$ne)

Situation:

👉 "Students NOT from CSE"

```
db.students.find({  
    branch: { $ne: "CSE" }  
})
```

◆ 8. IN condition (\$in)

Situation:

👉 "Students from CSE or ECE"

```
db.students.find({  
    branch: { $in: ["CSE", "ECE"] }  
})
```

SQL:

WHERE branch IN ('CSE','ECE')

◆ 9. NOT IN (\$nin)

```
db.students.find({  
    branch: { $nin: ["CSE", "ECE"] }  
})
```

◆ 10. OR condition (\$or) 🔥

Situation:

👉 “Age is 20 OR branch is ECE”

```
db.students.find({  
  $or: [  
    { age: 20 },  
    { branch: "ECE" }  
  ]  
})
```

📌 \$or takes an **array of conditions**

◆ 11. AND + OR together (REAL WORLD)

Situation:

👉 “CSE students whose age is above 20 OR ECE students”

```
db.students.find({  
  $or: [  
    { branch: "CSE", age: { $gt: 20 } },  
    { branch: "ECE" }  
  ]  
})
```

◆ 12. Match text (string)

Exact match only:

```
db.books.find({ title: "Mongodb basics" })
```

MongoDB is **case-sensitive** by default.

◆ 13. Match only specific fields (PROJECTION)

Situation:

👉 “Show only name and branch, hide _id”

```
db.students.find(  
  { branch: "CSE" },  
  { name: 1, branch: 1, _id: 0 }  
)  
 1 = show  
 0 = hide
```

◆ 14. LIMIT results

Situation:

 “Show only 2 students”

```
db.students.find().limit(2)
```

◆ 15. SORT results

Ascending:

```
db.students.find().sort({ age: 1 })
```

Descending:

```
db.students.find().sort({ age: -1 })
```

◆ 16. REAL WORLD EXAMPLES

 **Example 1: Books cheaper than 500**

```
db.books.find({ price: { $lt: 500 } })
```

 **Example 2: Books by author TG**

```
db.books.find({ author: "TG" })
```

 **Example 3: Books priced between 200–400**

```
db.books.find({  
  price: { $gte: 200, $lte: 400 }  
})
```

FINAL MENTAL MODEL (REMEMBER THIS)

Database → Collections → Documents

Documents → JSON objects

find() → filters documents

● STEP 5: UPDATING DOCUMENTS IN MONGODB

Updating = **changing existing data** inside a document.

◆ 1. IMPORTANT RULE (READ THIS FIRST)

 MongoDB NEVER updates without a filter

If you don't tell **which document**, MongoDB doesn't know what to update.

◆ 2. updateOne() (MOST COMMON)

Syntax:

```
db.collection.updateOne(  
  { filter },  
  { update }  
)
```

◆ 3. Example 1: Update ONE document

Situation:

 Change price of one book to 350

```
db.books.updateOne(  
  { title: "Mongodb basics" },  
  { $set: { price: 350 } }  
)
```

 \$set = change only that field

 Other fields stay untouched

- ◆ **4. What happens internally?**

Before:

```
{  
    title: "Mongodb basics",  
    price: 300,  
    author: "TG"  
}
```

After:

```
{  
    title: "Mongodb basics",  
    price: 350,  
    author: "TG"  
}
```

- ◆ **5. Why \$set is IMPORTANT** 

 **WRONG (DON'T DO THIS):**

```
db.books.updateOne(  
    { title: "Mongodb basics" },  
    { price: 400 }  
)
```

 This **replaces the entire document**, removing other fields.

 **ALWAYS** use update operators like \$set

- ◆ **6. updateMany() (BULK UPDATE)**

Situation:

 Increase price to 400 for **ALL books by TG**

```
db.books.updateMany(  
    { author: "TG" },
```

```
{ $set: { price: 400 } }  
})
```

👉 All matching documents get updated.

◆ 7. Increment / Decrement values (\$inc) 🔥

Situation:

👉 Increase price by 50

```
db.books.updateOne(  
  { title: "Mongodb basics" },  
  { $inc: { price: 50 } }  
)
```

If price = 300 → becomes **350**

◆ 8. Add a NEW field while updating

Situation:

👉 Add publishedYear

```
db.books.updateOne(  
  { title: "Mongodb basics" },  
  { $set: { publishedYear: 2024 } }  
)
```

👉 MongoDB allows **schema-less updates**

◆ 9. Rename a field (\$rename)

Situation:

👉 Change author → writer

```
db.books.updateMany(  
  {},  
  { $rename: { author: "writer" } }  
)
```

- ◆ **10. Remove a field (\$unset)**

Situation:

👉 Remove publishedYear

```
db.books.updateOne(  
  { title: "Mongodb basics" },  
  { $unset: { publishedYear: "" } }  
)
```

- ◆ **11. Update using conditions**

Situation:

👉 Increase price by 100 for books cheaper than 300

```
db.books.updateMany(  
  { price: { $lt: 300 } },  
  { $inc: { price: 100 } }  
)
```

- ◆ **12. Upsert (UPDATE + INSERT) 🔥**

Situation:

👉 If book exists → update

👉 If NOT → insert new document

```
db.books.updateOne(  
  { title: "Advanced MongoDB" },  
  { $set: { price: 500, author: "TG" } },  
  { upsert: true }  
)
```

- ◆ **13. Real-world thinking**

Situation	MongoDB command
Update profile	updateOne()
Change salary of all employees	updateMany()
Add new column	\$set
Increase value	\$inc
Delete column	\$unset

MEMORY RULE (VERY IMPORTANT)

updateOne → single document

updateMany → multiple documents

\$set → change/add field

\$inc → number change

\$unset → remove field

upsert → update or insert

STEP 6: DELETING DOCUMENTS IN MONGODB (FULL MASTER GUIDE)

Deleting = permanently removing data

 NO UNDO (unless you have backups)

◆ 1. Delete basics (DEFINITION)

MongoDB deletes **DOCUMENTS**, not databases or fields directly.

◆ 2. deleteOne() (SAFE DELETE)

Definition:

Deletes **ONLY ONE matching document**

Syntax:

db.collection.deleteOne({ filter })

Example 1: Delete one book

```
db.books.deleteOne({ title: "Mongodb basics" })
```

- ✓ Deletes first matching document
 - ✓ Other documents stay
-

When to use:

- Delete user account
 - Remove one order
 - Delete single record
-

◆ 3. deleteMany() (BULK DELETE ⚠)

Definition:

Deletes **ALL documents** matching condition

Syntax:

```
db.collection.deleteMany({ filter })
```

Example 2: Delete all books by author TG

```
db.books.deleteMany({ author: "TG" })
```

⚠ DANGEROUS if filter is wrong

◆ 4. MOST DANGEROUS COMMAND 🚨

This deletes EVERYTHING:

```
db.books.deleteMany({})
```

✗ Collection becomes empty
✗ Data is gone forever

◆ 5. Delete using conditions

Example: Delete books cheaper than 200

```
db.books.deleteMany({ price: { $lt: 200 } })
```

◆ **6. Delete using _id (SAFEST WAY)**

_id is UNIQUE

```
db.books.deleteOne({  
  _id: ObjectId("696cd0e1f8f0bb0e388b87cb")  
})
```

✓ Guaranteed single deletion

✓ Best practice

◆ **7. Delete result explanation**

MongoDB returns:

```
{  
  acknowledged: true,  
  deletedCount: 1  
}
```

Field	Meaning
-------	---------

acknowledged MongoDB received request

deletedCount Number of documents deleted

◆ **8. SAFE DELETE PRACTICE (VERY IMPORTANT)**

ALWAYS check before deleting:

```
db.books.find({ author: "TG" })
```

Then:

```
db.books.deleteMany({ author: "TG" })
```

◆ **9. Delete vs Drop (BIG DIFFERENCE)**

Command	What it does
---------	--------------

deleteOne Removes documents

deleteMany Removes documents

Command What it does

drop() Removes entire collection

◆ 10. Drop collection (EXTREME)

db.books.drop()

✗ Deletes collection

✗ Deletes indexes

✗ Cannot be undone

◆ 11. Drop database (NUCLEAR 💣)

use practice_db

db.dropDatabase()

✗ Deletes database

✗ All collections gone

◆ 12. REAL WORLD SCENARIOS

👤 User deletes account

db.users.deleteOne({ _id: ObjectId("...") })

🛒 Delete expired orders

db.orders.deleteMany({ status: "expired" })

🧹 Cleanup test data

db.logs.deleteMany({ environment: "test" })

◆ 13. Soft Delete (INDUSTRY PRACTICE) ⭐

Instead of deleting:

db.users.updateOne(

{ _id: ObjectId("...") },

```
{ $set: { isDeleted: true } }
```

)

- ✓ Data is safe
 - ✓ Reversible
 - ✓ Used in real systems
-

DELETE GOLDEN RULES

1. Always use find() before delete()
 2. Prefer deleteOne() over deleteMany()
 3. Use _id when possible
 4. Never run deleteMany({}) casually
 5. Soft delete in production
-

CRUD COMPLETED

You now know:

- CREATE → insertOne, insertMany
- READ → find
- UPDATE → updateOne, updateMany
- DELETE → deleteOne, deleteMany

STEP 7: DATATYPES IN MONGODB (COMPLETE GUIDE)

MongoDB stores data in **BSON** (Binary JSON).

BSON supports **more data types than normal JSON** .

◆ 1. MOST COMMON MONGODB DATATYPES (YOU MUST KNOW)

Type	Example	Used for
String	"Ganesh"	Names, text
Number (Int)	20	Age, count
Number (Double)	99.5	Price, marks

Type	Example	Used for
Boolean	true / false	Flags
Object	{ city: "Hyd" }	Embedded docs
Array	["Java", "Python"]	Lists
ObjectId	ObjectId("...")	Primary key
Date	ISODate()	Time-based data
Null	null	Empty value

◆ 2. STRING

```
{  
  name: "Ganesh",  
  course: "MongoDB"  
}
```

✓ Always in **quotes**

◆ 3. NUMBER TYPES (IMPORTANT)

MongoDB automatically decides number type.

```
age: 21    // Int32  
price: 299.99 // Double
```

Explicit number types (advanced)

```
age: NumberInt(21)  
count: NumberLong(100000)
```

Used in **large-scale systems**

◆ 4. BOOLEAN

```
isActive: true  
isDeleted: false
```

- ✓ Used for **status, flags, soft delete**
-

◆ **5. ARRAY (VERY POWERFUL)**

skills: ["Java", "Python", "MongoDB"]

You can search inside arrays easily later.

◆ **6. OBJECT (EMBEDDED DOCUMENT)**

address: {

 city: "Hyderabad",

 pincode: 500001

}

✓ No JOIN needed

✓ Faster reads

◆ **7. ObjectId (PRIMARY KEY)**

MongoDB automatically creates `_id`.

`_id: ObjectId("696cd0e1f8f0bb0e388b87cb")`

Contains:

- timestamp
 - machine id
 - process id
 - counter
-

◆ **8. NULL**

`middleName: null`

Means field exists but value is empty.

 **MOST IMPORTANT: DATE TYPE IN MONGODB**

◆ **9. DATE datatype (VERY IMPORTANT)**

MongoDB stores date as **ISODate**

createdAt: ISODate("2026-01-18T07:30:00Z")

◆ **10. Insert CURRENT date/time**

```
db.users.insertOne({  
  name: "Ganesh",  
  createdAt: new Date()  
})
```

✓ Stores current timestamp

✓ Best practice

◆ **11. Insert CUSTOM date**

```
db.users.insertOne({  
  name: "Tarun",  
  joinedOn: new Date("2025-06-01")  
})
```

◆ **12. View how MongoDB stores date**

MongoDB stores date as **milliseconds since Jan 1, 1970**

But shows it in readable format.

◆ **13. Query using DATE**

Find users created after a date

```
db.users.find({  
  createdAt: { $gt: new Date("2026-01-01") }  
})
```

Between two dates

```
db.users.find({  
    createdAt: {  
        $gte: new Date("2026-01-01"),  
        $lte: new Date("2026-01-31")  
    }  
})
```

◆ **14. Update DATE**

```
db.users.updateOne(  
    { name: "Ganesh" },  
    { $set: { lastLogin: new Date() } }  
)
```

◆ **15. TTL (Time To Live) using DATE** 

Auto-delete after time.

```
db.sessions.createIndex(  
    { createdAt: 1 },  
    { expireAfterSeconds: 3600 }  
)
```

✓ Auto deletes after 1 hour

✓ Used in sessions, OTPs

◆ **16. DATE vs STRING (VERY IMPORTANT)**

✗ WRONG:

createdAt: "2026-01-18"

✓ RIGHT:

createdAt: new Date("2026-01-18")

Reason:

- String → no date comparison

- Date → supports sorting & filtering
-

◆ 17. REAL WORLD DOCUMENT EXAMPLE

```
{  
  name: "Ganesh",  
  age: 21,  
  skills: ["Java", "MongoDB"],  
  isActive: true,  
  address: {  
    city: "Hyderabad",  
    pincode: 500001  
  },  
  createdAt: new Date(),  
  lastLogin: null  
}
```

MENTAL MODEL (REMEMBER THIS)

MongoDB = JSON-like documents

Fields can have ANY datatype

No fixed schema

Date = new Date()