Introduction to NoSQL Databases:

What is NoSQL?

- **Not Only SQL** – alternative to traditional relational databases (MySQL, Oracle).
- Designed for flexibility, scalability, and handling unstructured data.

Types of NoSQL Databases:

1. **Document-based** (MongoDB, CouchDB) → store JSON-like docs
2. **Key-Value** (Redis)
3. **Column-based** (Cassandra)
4. **Graph-based** (Neo4j)

What is MongoDB?

- **MongoDB** is a **NoSQL database** (Not Only SQL).
- Instead of storing data in **tables (rows & columns)** like SQL databases, MongoDB stores data in **collections** (similar to tables) and **documents** (similar to rows).
- Each **document** is written in **JSON-like format** (`BSON` (**Binary JSON**) internally).

Example document in MongoDB:

```
{
 "name": "John",
 "age": 25,
 "skills": ["JavaScript", "Node.js", "MongoDB"],
 "location": { "city": "Hyderabad", "country": "India" }
}
```

Why MongoDB?

- Schema-less → no need to define columns in advance.
- Scalable → supports big data & distributed systems.
- Stores nested JSON objects & arrays easily.
- Widely used in **Node.js + Express.js applications**.

Purpose of MongoDB

1. **Flexibility** – No need for fixed schema like SQL. You can add new fields anytime.
2. **Scalability** – Handles large amounts of data (Big Data).
3. **High performance** – Great for real-time apps like chat, e-commerce, IoT.
4. **Document-oriented** – Stores complex data (arrays, objects) naturally.

SQL vs MongoDB (Key Differences)

Feature	SQL (Relational DB)	MongoDB (NoSQL DB)
Data model	Tables (rows & columns)	Collections & Documents
Schema	Fixed schema (strict)	Schema-less (flexible)
Joins	Supports JOIN	No joins (uses embedding or `$lookup`)
Transactions	Yes (ACID)	Yes (since MongoDB 4.0, but mostly used per-document)
Scaling	Vertical (bigger server)	Horizontal (sharding across servers)
Query language	SQL	MongoDB Query Language (MQL)
Best for	Banking, ERP (structured)	Real-time apps, Big Data, IoT

Software Required

You need **MongoDB Server + Client tool**.

(a) **MongoDB Community Edition (Server)**

- Free and open-source version.
- This is the actual **database engine** that stores and processes data.

(b) Client Tools to write commands:

You can write MongoDB commands in **different ways**:

1. **Mongo Shell (mongosh)**
   - Command-line tool for writing MongoDB queries.

Example:

```
mongosh
use myDatabase
db.users.insertOne({ name: "John", age: 25 })
```

2. **MongoDB Compass (GUI Tool)**

- A graphical interface to interact with MongoDB.
- Good for beginners who don’t like only typing commands.
- Example: You can visually create collections, insert documents, and query data.

3. **Drivers (Programming Language Integration)**

- Use MongoDB inside programming languages like Node.js, Python, Java, etc.

  Example (Node.js):

```
const { MongoClient } = require("mongodb");
const client = new MongoClient("mongodb://localhost:27017");
async function run() {
  await client.connect();
  const db = client.db("myDatabase");
  const users = db.collection("users");
  await users.insertOne({ name: "Rakesh", age: 25 });
  console.log(await users.find().toArray());
  await client.close();
}
run();
```

How to Install & Use MongoDB

Step 1: Download MongoDB

- From official site: https://www.mongodb.com/try/download/community
- Install **MongoDB Server** + **MongoDB Compass**.

Step 2: Verify Installation

Open terminal and type: write below commands

```
mongod   --version
mongosh  --version
```

Step 3: Start MongoDB Server

- Usually MongoDB runs as a **service** automatically.
- To start manually: write command - mongod

Step 4: Open MongoDB Shell

- In another terminal: write command –mongosh

Or

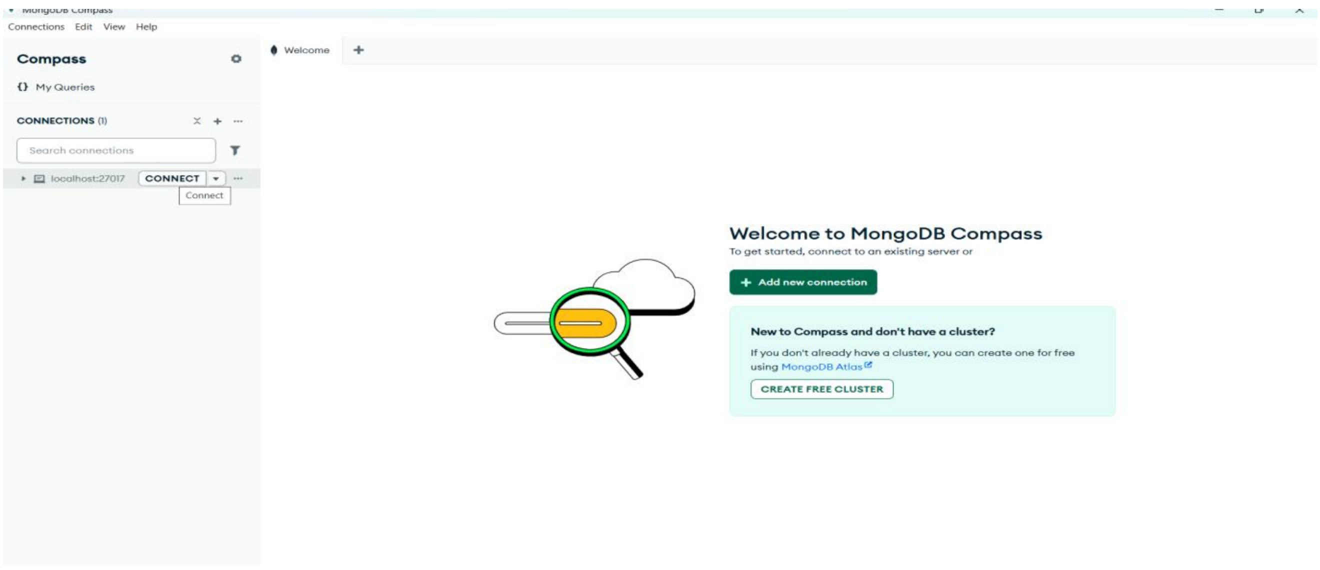
If you missed it, you can download Compass separately and install it like a normal windows program:
☞ https://www.mongodb.com/try/download/compass

How to Use Compass

1. **Open MongoDB Compass** (from Start Menu or Desktop).
2. In the connection box, enter: localhost:27017
   - `localhost` → your computer. • `27017` → default MongoDB port.

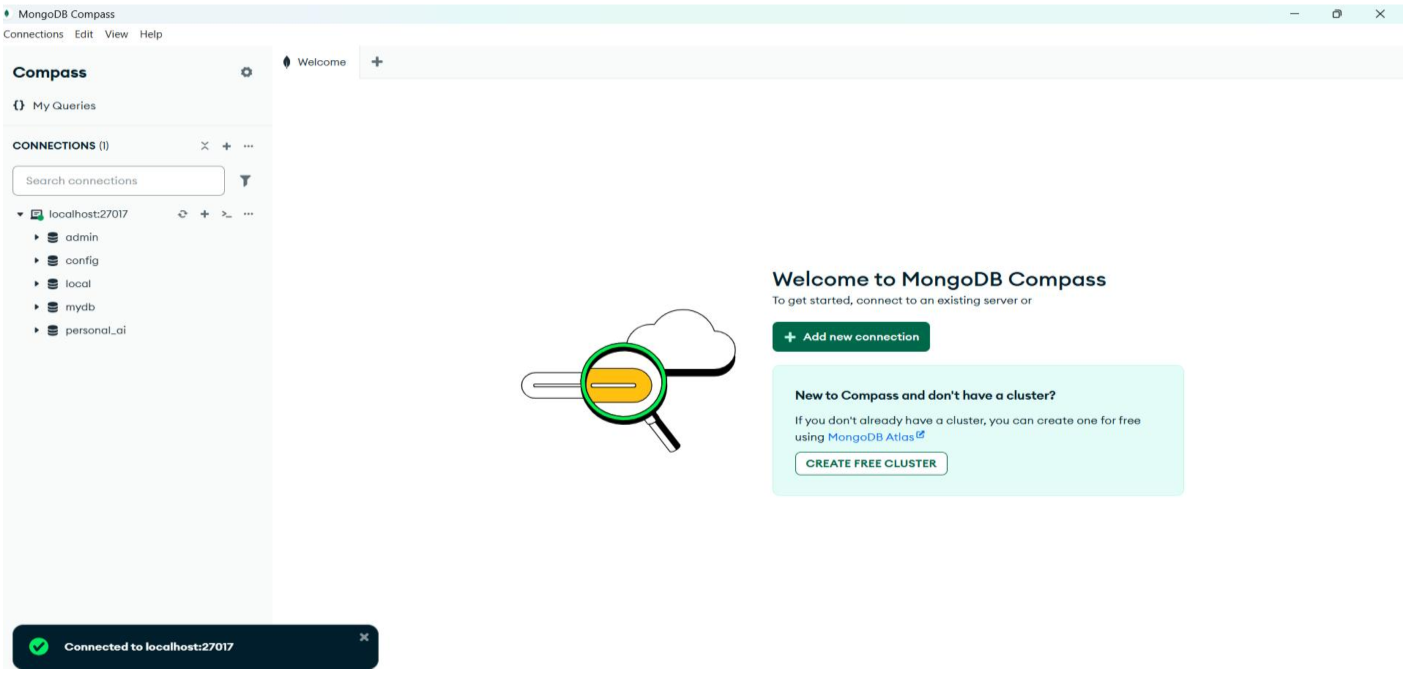
   (Change only if you configured MongoDB differently.)

3. Click **Connect**.
4. Once connected, you'll see:
   - **List of Databases** (e.g., `admin`, `config`, `local`).
   - Option to **Create Database**.
   - Option to browse **Collections**, **Documents**, and run queries visually.



Welcome to MongoDB Compass
To get started, connect to an existing server or

New to Compass and don't have a cluster?
If you don't already have a cluster, you can create one for free using MongoDB Atlas



Welcome to MongoDB Compass
To get started, connect to an existing server or

New to Compass and don't have a cluster?
If you don't already have a cluster, you can create one for free using MongoDB Atlas

Connected to localhost:27017

BSON stands for Binary JSON**:**

It is a **binary-encoded serialization format** used primarily by **MongoDB** to store and transfer documents. BSON extends the JSON (JavaScript Object Notation) model but is designed to be more efficient in terms of **speed** and **space** when dealing with structured data.

Key Features of BSON

1. **Binary Format**
   - Unlike JSON (which is text-based), BSON stores data in binary, making it faster to parse and more compact.
2. **Rich Data Types**
   - Supports all JSON data types (string, number, array, object) **plus additional ones** like:
     - `Date`
     - `Binary data`
     - `ObjectId` (special 12-byte unique identifier in MongoDB)
     - `Decimal128`
     - `Timestamp`
     - `Null`
3. **Efficient Encoding/Decoding**
   - Optimized for fast encoding/decoding within databases like MongoDB.
4. **Traversable**
   - Fields in BSON documents include length prefixes, making it easier to skip fields without decoding everything.

Example:

JSON Document:

```
{
  "name": "John",
  "age": 25,
  "isStudent": false
}
```

BSON Representation (conceptually):

```
16 bytes total
02 name 00 07 00 00 00 52 61 6b 65 73 68 00
10 age 00 19 00 00 00
08 isStudent 00 00
00
```

Here `02` means **string type**, `10` means **integer type**, `08` means **boolean type**.

Where BSON is Used

- **MongoDB:** The core storage and network transfer format.
- **Drivers:** MongoDB drivers use BSON internally for communication.
- **Serialization:** As an alternative to JSON when binary efficiency is needed.

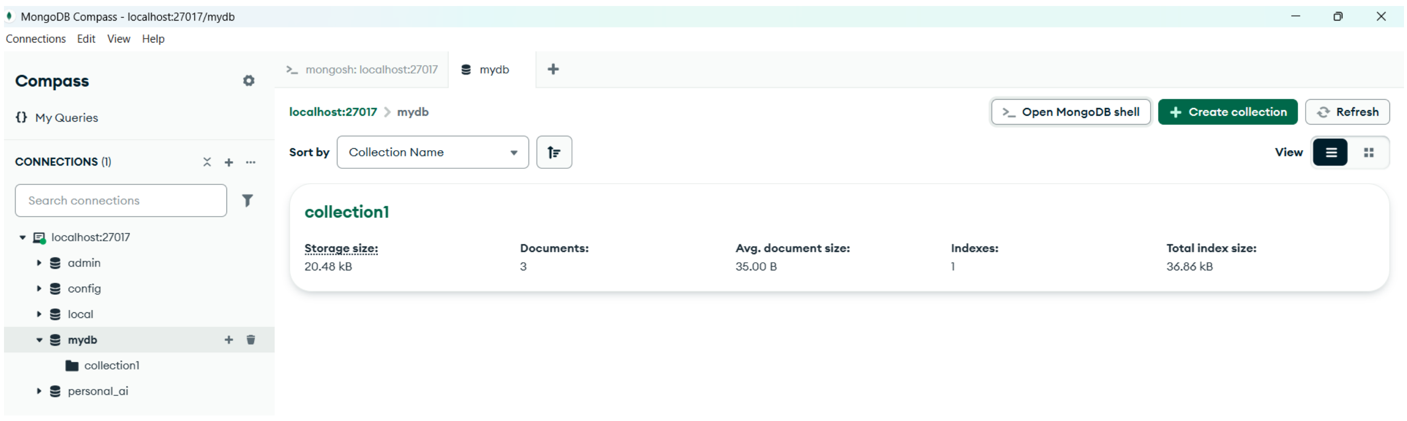
JSON vs BSON;

Feature	JSON	BSON
Full Form	JavaScript Object Notation	Binary JSON
Format Type	Text-based (human-readable)	Binary-encoded (machine-friendly)
Data Types	Limited → string, number, boolean, null, array, object	Rich → All JSON types **plus** Date, ObjectId, Binary data, Timestamp, Decimal128, etc.
Readability	Easy to read & edit (good for config, APIs, logs)	Not human-readable (intended for machines)
Size	Usually smaller for simple data	Can be larger due to type + length metadata overhead
Speed	Slower to parse (needs text parsing)	Faster parsing (binary format is closer to memory representation)
Use Case	Data exchange (APIs, config files, logs)	Storage and communication in MongoDB (databases)
Traversal	Linear scan required	Random access possible (length prefix allows skipping fields quickly)
Example (conceptual)	`{ "age": 25 }` (text)	`10 age 00 19 00 00 00` (binary with type + length info)

Ways to Write Commands in MongoDB

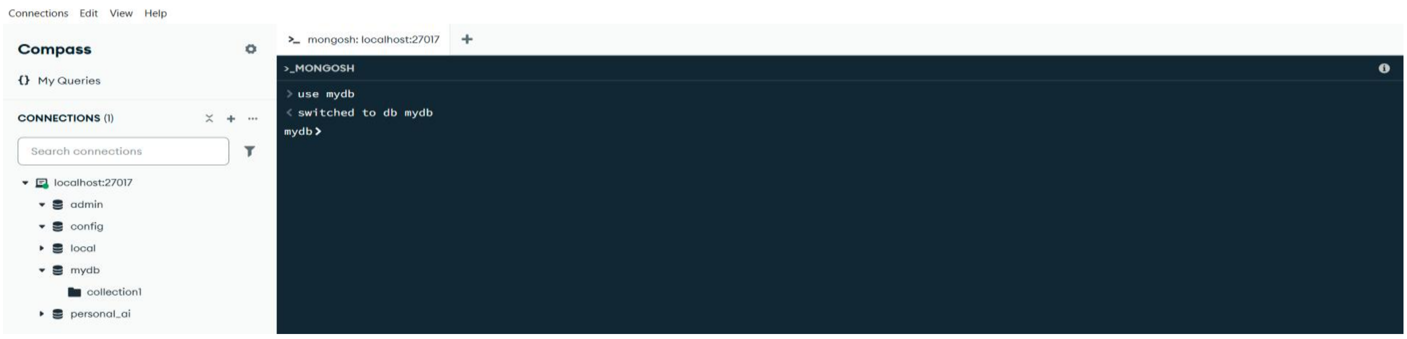
1. **Mongo Shell (mongosh)** → direct commands in terminal.
2. **MongoDB Compass GUI** → point-and-click + queries.
3. **Programming Drivers** (Node.js, Python, Java, C#, etc.) → queries inside code.
4. **Cloud Services** like **MongoDB Atlas** → run queries online without installing locally.

Note: Writing commands in mongoDB shell, click on connection i.e (localhost:27017 here 27017 is the default port number of monogDB) and you will see "open MongoDB shell" button and then click you will get terminal and by default it showing 'use mydb'(here 'mydb' is the user created database) command and it switched to your database directly. For understanding purpose see below screen shots



collection1

Storage size: 20.48 kB
Documents: 3
Avg. document size: 35.00 B
Indexes: 1
Total index size: 36.86 kB



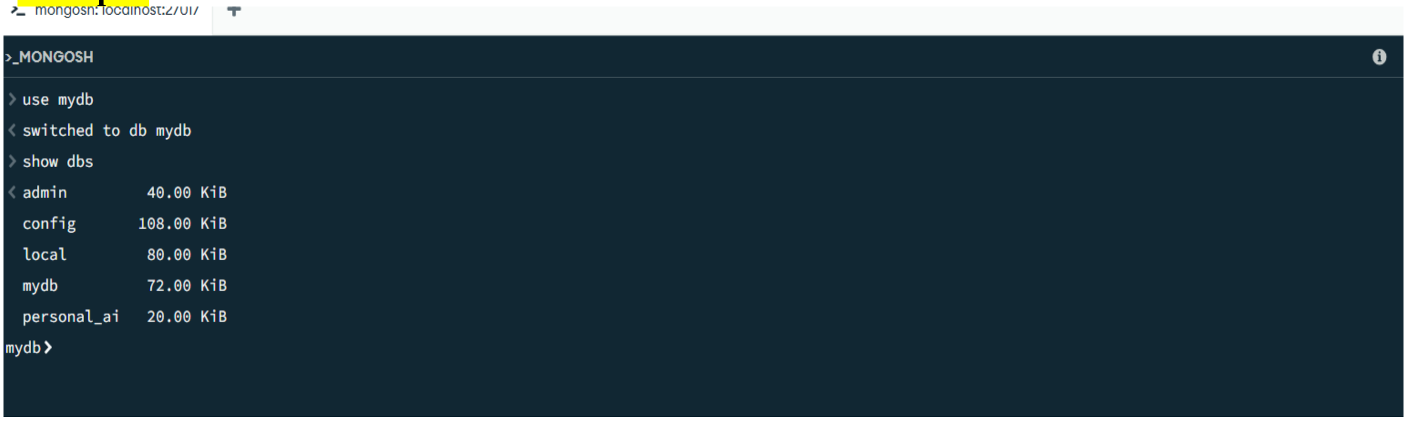
```
> use mydb
< switched to db mydb
mydb >
```

MongoDB Commands (with Examples)

A. Database Commands

1. **Show all databases:** You will get List of all databases.
   Command: show dbs

Example:



```
> use mydb
< switched to db mydb
> show dbs
< admin         40.00 KiB
  config       108.00 KiB
  local         80.00 KiB
  mydb          72.00 KiB
  personal_ai   20.00 KiB
mydb >
```

2. **Use / Create a database:** Switches to `mydb`. If it doesn't exist, it will be created when data is inserted.

   Command: use mydb

3. **Show current database:** it returns your current database name
   Command: db

4. **delete Database Entirely (with all collections inside)** : removes the **database itself**, and automatically removes all collections inside it. After this, `myDatabase` won't exist anymore until you create it again.

   Command: use myDatabase
   db.dropDatabase();

Note: In MongoDB, a database is **not really created** until you **insert at least one collection/document**.

- When you do `use mydb`, MongoDB **switches context** to `mydb`.
- But `show dbs` doesn't list it, because there's nothing inside yet.

B. Collection Commands

1. **Show all collections:** Lists all collections inside the current database.
   Command: **show collections**
   Example: If DB is `mydb` and you created collections `users` and `orders`,
   output will be:
   users
   orders

2. **Create a collection:** Creates an empty collection named `users`.
   Command: db.createCollection("users")

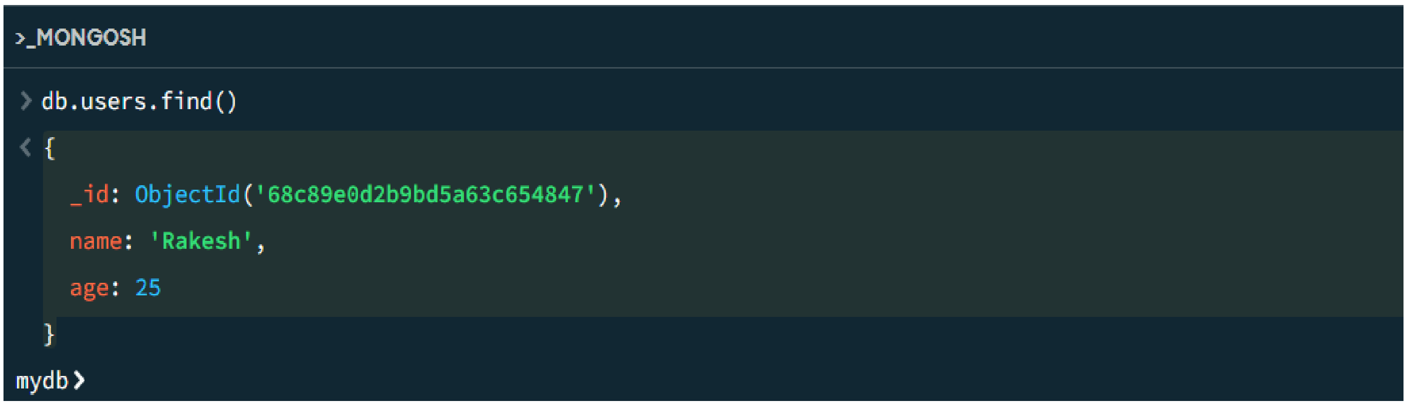
   ⚡**Note:** You don't *always* need this, because MongoDB will auto-create a collection when you insert the first document.

3. **Drop (delete) a collection**: Deletes the collection `users` along with all documents inside it.
   Command: db.users.drop()

   Output:
```
true    // if dropped successfully
```

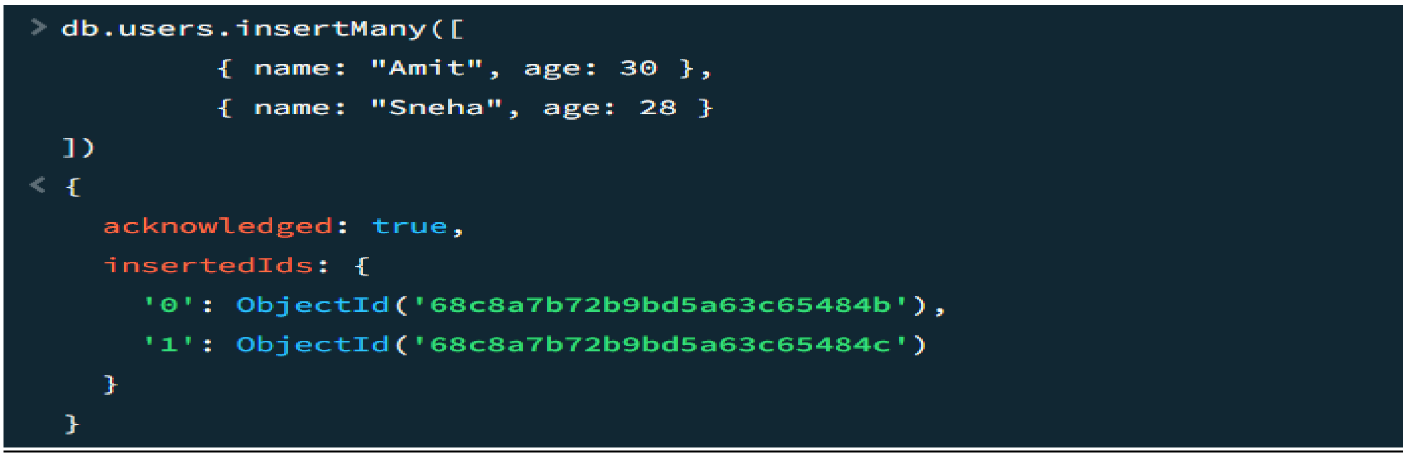
C. Insert Commands

1. **Insert one document:** Adds a single record into the `users` collection. MongoDB will also generate a unique `_id` automatically:
   Command: db.users.insertOne({ name: "John", age: 25 })
   Example: use find command to check inserted document
   Command: db.users.find()



```
>_MONGOSH
> db.users.find()
< {
    _id: ObjectId('68c89e0d2b9bd5a63c654847'),
    name: 'Rakesh',
    age: 25
  }
mydb>
```

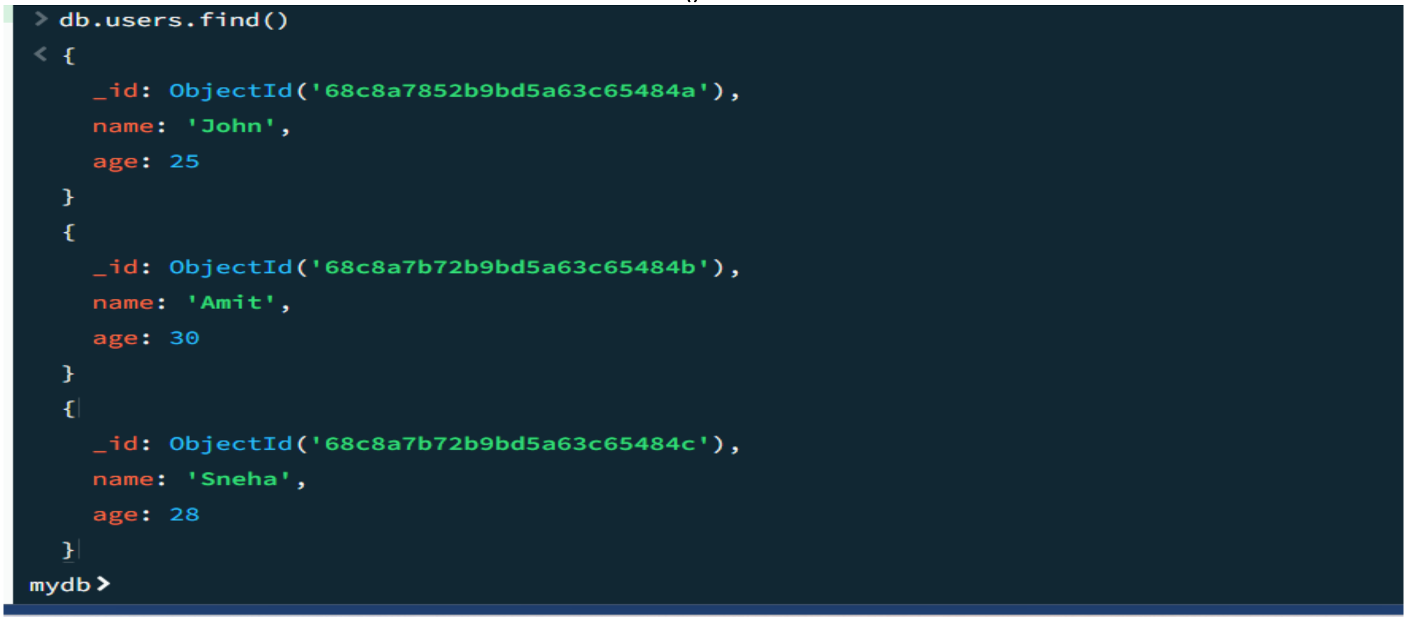
2. **Insert many documents:** Inserts multiple records at once.
   Command: db.users.insertMany([{ name: "Amit", age: 30 },{ name: "Sneha", age: 28 }])



```
> db.users.insertMany([
          { name: "Amit", age: 30 },
          { name: "Sneha", age: 28 }
  ])
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('68c8a7b72b9bd5a63c65484b'),
      '1': ObjectId('68c8a7b72b9bd5a63c65484c')
    }
  }
```

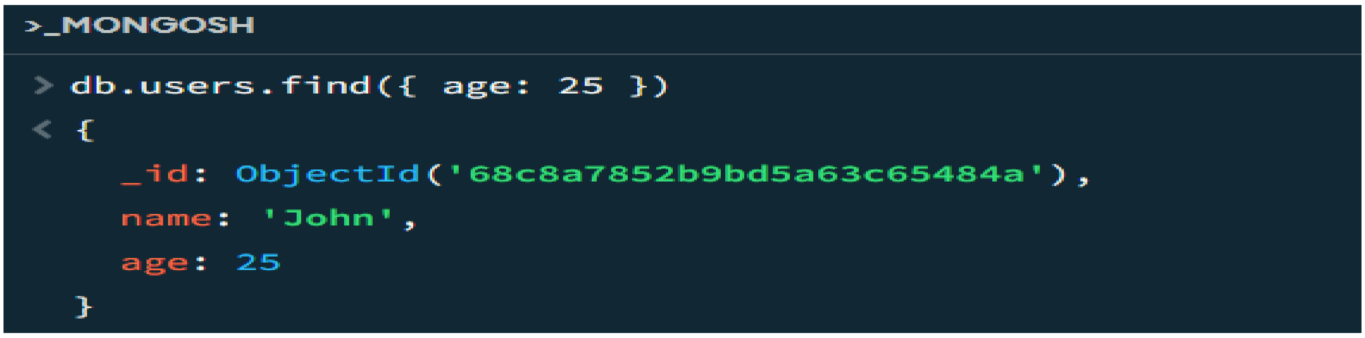
D. Query (Find) Commands

1. **Find all documents:** ☞ Displays all users in the collection.
   Command: db.users.find()



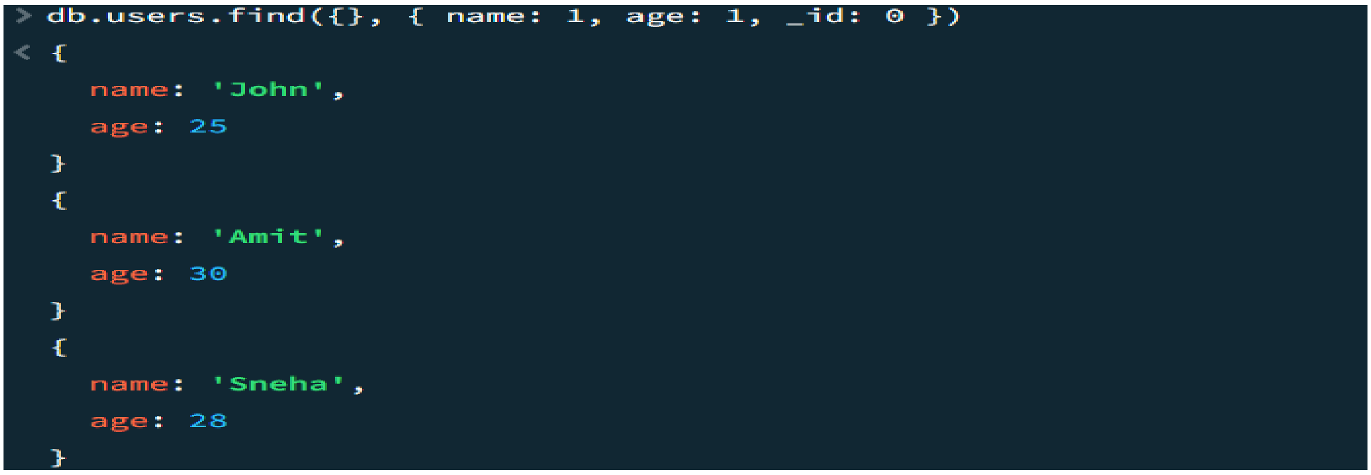
```
> db.users.find()
< {
    _id: ObjectId('68c8a7852b9bd5a63c65484a'),
    name: 'John',
    age: 25
  }
  {
    _id: ObjectId('68c8a7b72b9bd5a63c65484b'),
    name: 'Amit',
    age: 30
  }
  {
    _id: ObjectId('68c8a7b72b9bd5a63c65484c'),
    name: 'Sneha',
    age: 28
  }
mydb>
```

2. **Find with condition:** Returns only the document(s) based on condition
   Example: db.users.find({ age: 25 })



```
>_MONGOSH
> db.users.find({ age: 25 })
< {
    _id: ObjectId('68c8a7852b9bd5a63c65484a'),
    name: 'John',
    age: 25
  }
```

3. **Find with projection (specific fields only):** Shows only name and age (hides _id).
   Example: db.users.find({}, { name: 1, age: 1, _id: 0 })



```
> db.users.find({}, { name: 1, age: 1, _id: 0 })
< {
    name: 'John',
    age: 25
  }
  {
    name: 'Amit',
    age: 30
  }
  {
    name: 'Sneha',
    age: 28
  }
```

MongoDB Operators:

1. Comparison Operators

Operator	Meaning	Example
`$eq`	Equal to	`{ age: { $eq: 25 } }` → age = 25
`$ne`	Not equal	`{ age: { $ne: 25 } }` → age ≠ 25
`$gt`	Greater than	`{ age: { $gt: 25 } }` → age > 25
`$gte`	Greater than or equal	`{ age: { $gte: 25 } }` → age ≥ 25
`$lt`	Less than	`{ age: { $lt: 25 } }` → age < 25
`$lte`	Less than or equal	`{ age: { $lte: 25 } }` → age ≤ 25
`$in`	Matches values in array	`{ age: { $in: [25, 30] } }` → age = 25 or 30
`$nin`	Not in array	`{ age: { $nin: [25, 30] } }` → age ≠ 25, 30

Examples:

1. **db.users.find({ age: { $eq: 25 } }):** ☞ Finds users whose age is **exactly 25**.
2. **db.users.find({ age: { $ne: 25 } }):** ☞ Finds users whose age is **not 25**.
3. **db.users.find({ age: { $gt: 25 } }):** ☞ Finds users whose age is **greater than 25**.
4. **db.users.find({ age: { $gte: 25 } }):** ☞ Finds users whose age is **25 or more**.
5. **db.users.find({ age: { $lt: 25 } })** : ☞ Finds users whose age is **less than 25**.
6. **`db.users.find({ age: { $lte: 25 } })`** : ☞ Finds users whose age is **25 or less**.
7. **db.users.find({ age: { $in: [20, 25, 30] } }):** ☞ Finds users whose age is **20, 25, or 30**.
8. **db.users.find({ age: { $nin: [20, 25, 30] } }):** ☞ Finds users whose age is **NOT 20, 25, or 30**.

2. Logical Operators.

Operator	Meaning	Example
`$and`	All conditions must be true	`{ $and: [{ age: { $gt: 20 } }, { age: { $lt: 30 } }] }`
`$or`	At least one condition true	`{ $or: [{ age: 25 }, { name: "John" }] }`
`$not`	Negates condition	`{ age: { $not: { $gt: 30 } } }` → age ≤ 30
`$nor`	None of conditions true	`{ $nor: [{ age: 25 }, { name: "John" }] }`

3. Element Operators

Operator	Meaning	Example
`$exists`	Field exists or not	`{ age: { $exists: true } }`
`$type`	Matches field by type	`{ age: { $type: "int" } }`

4. Evaluation Operators

Operator	Meaning	Example
`$regex`	Pattern matching	`{ name: { $regex: "^J" } }` → starts with J
`$expr`	Use aggregation expressions	`{ $expr: { $gt: ["$spent", "$budget"] } }`
`$jsonSchema`	Validate against schema	`{ $jsonSchema: { required: ["name", "age"] } }`
`$mod`	Modulo operation	`{ age: { $mod: [5, 0] } }` → age divisible by 5
`$where`	JavaScript condition	`{ $where: "this.age > this.score" }`

5. Array Operators

Operator	Meaning	Example
`$all`	Match all values in array	`{ tags: { $all: ["red", "blue"] } }`
`$elemMatch`	Match element inside array	`{ scores: { $elemMatch: { $gt: 80, $lt: 90 } } }`
`$size`	Match array size	`{ tags: { $size: 3 } }`

6. Projection Operators (used in find queries)

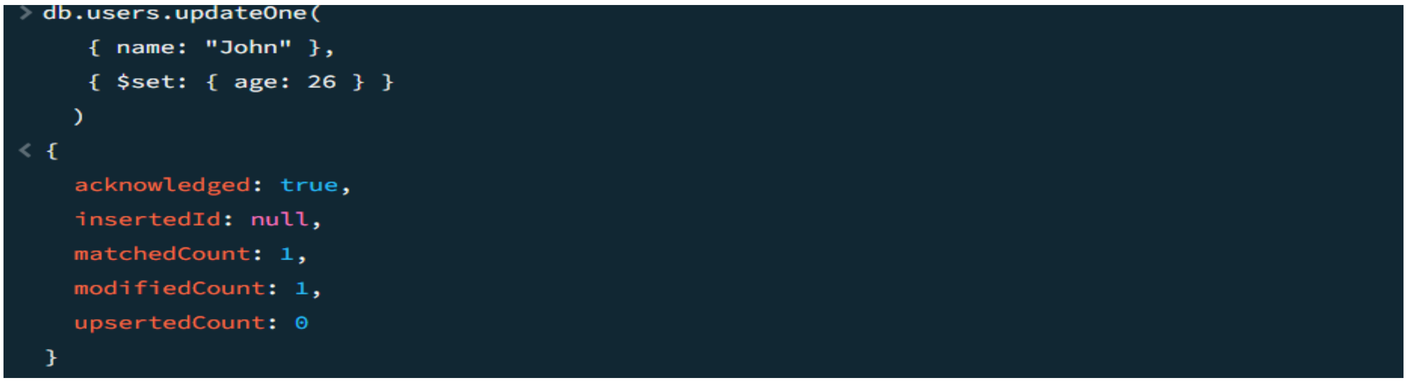
Operator	Meaning	Example
`$`	First matching element in array	`db.students.find({ scores: 90 }, { "scores.$": 1 })`
`$elemMatch`	Project array element match	`db.students.find({}, { scores: { $elemMatch: { $gt: 80 } } })`
`$meta`	Text search metadata	`{ score: { $meta: "textScore" } }`
`$slice`	Limit array elements	`{ comments: { $slice: 5 } }`

7. Update Operators

Operator	Meaning	Example
`$set`	Update field	`{ $set: { age: 30 } }`
`$unset`	Remove field	`{ $unset: { age: "" } }`
`$inc`	Increment value	`{ $inc: { age: 1 } }`
`$mul`	Multiply value	`{ $mul: { age: 2 } }`
`$rename`	Rename field	`{ $rename: { "oldName": "newName" } }`
`$min`	Update if smaller	`{ $min: { age: 20 } }`
`$max`	Update if larger	`{ $max: { age: 40 } }`
`$currentDate`	Set current date	`{ $currentDate: { lastModified: true } }`

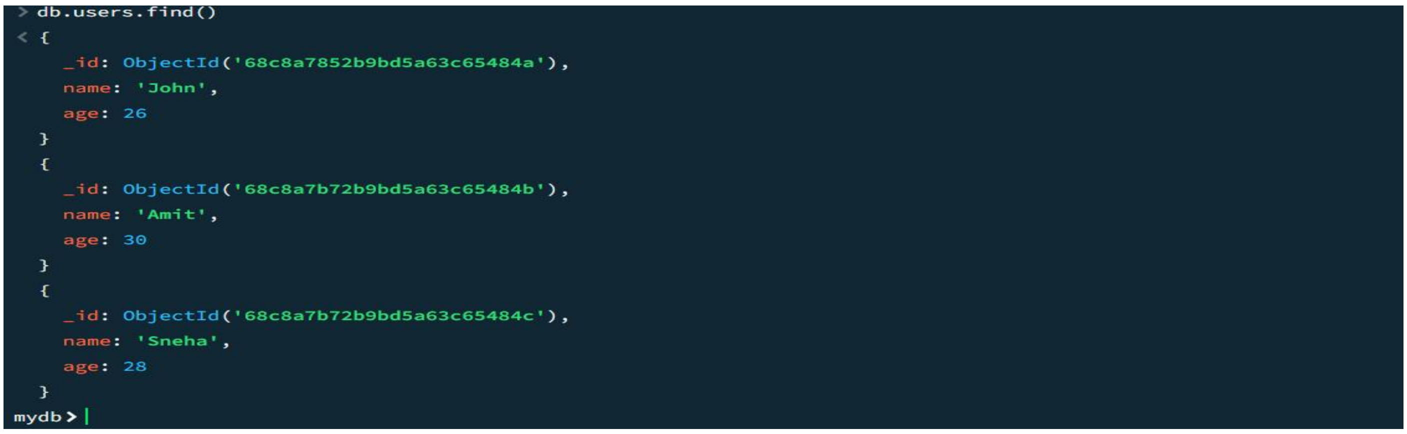
E. Update Commands:

1. **Update one document**: Finds the document based on condition
   Example: db.users.updateOne({ name: "John" }, { $set: { age: 26 } })



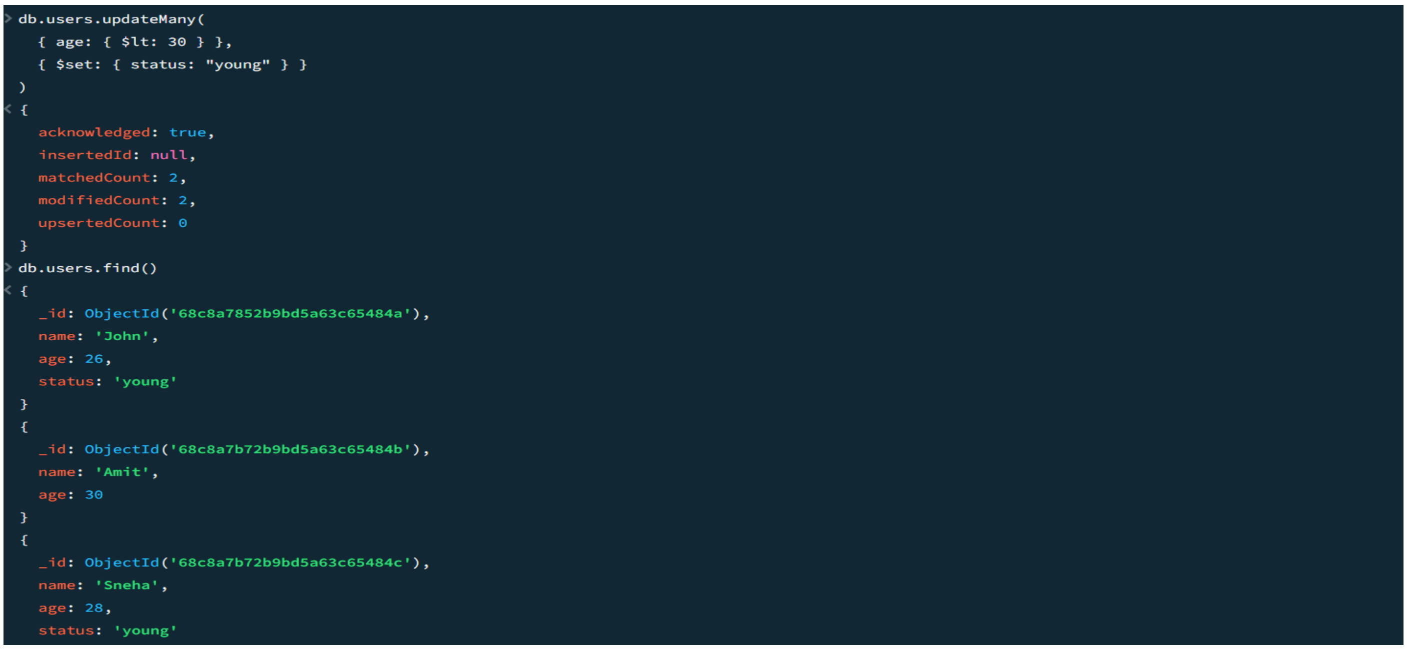
```
> db.users.updateOne(
      { name: "John" },
      { $set: { age: 26 } }
    )
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 1,
    modifiedCount: 1,
    upsertedCount: 0
  }
```

Apply find for checking updates in Documents:



```
> db.users.find()
< {
    _id: ObjectId('68c8a7852b9bd5a63c65484a'),
    name: 'John',
    age: 26
  }
  {
    _id: ObjectId('68c8a7b72b9bd5a63c65484b'),
    name: 'Amit',
    age: 30
  }
  {
    _id: ObjectId('68c8a7b72b9bd5a63c65484c'),
    name: 'Sneha',
    age: 28
  }
mydb > |
```

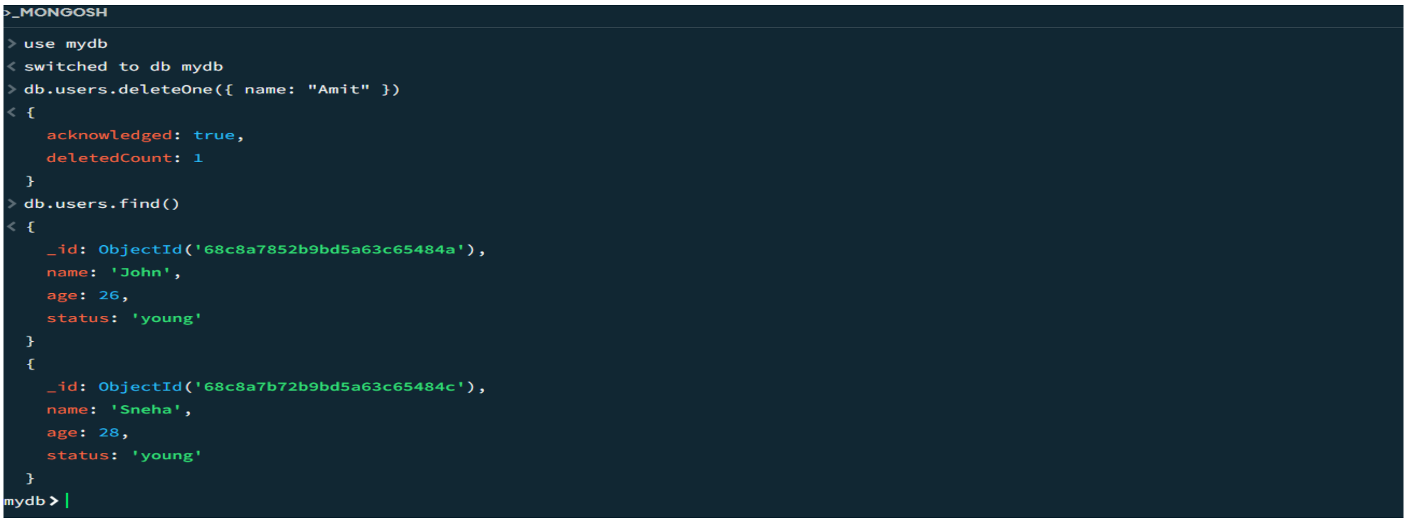
2. **Update many documents:** Updates all users data based on condition
   Example: db.users.updateMany ({ age: { $lt: 30 } }, { $set: { status: "young" } })
   Explanation: For all users with `age < 30`, add a new field `status = "young"`.
   Now Sneha and John will have a `status` field.



```
> db.users.updateMany(
    { age: { $lt: 30 } },
    { $set: { status: "young" } }
  )
< {
    acknowledged: true,
    insertedId: null,
    matchedCount: 2,
    modifiedCount: 2,
    upsertedCount: 0
  }
> db.users.find()
< {
    _id: ObjectId('68c8a7852b9bd5a63c65484a'),
    name: 'John',
    age: 26,
    status: 'young'
  }
  {
    _id: ObjectId('68c8a7b72b9bd5a63c65484b'),
    name: 'Amit',
    age: 30
  }
  {
    _id: ObjectId('68c8a7b72b9bd5a63c65484c'),
    name: 'Sneha',
    age: 28,
    status: 'young'
```

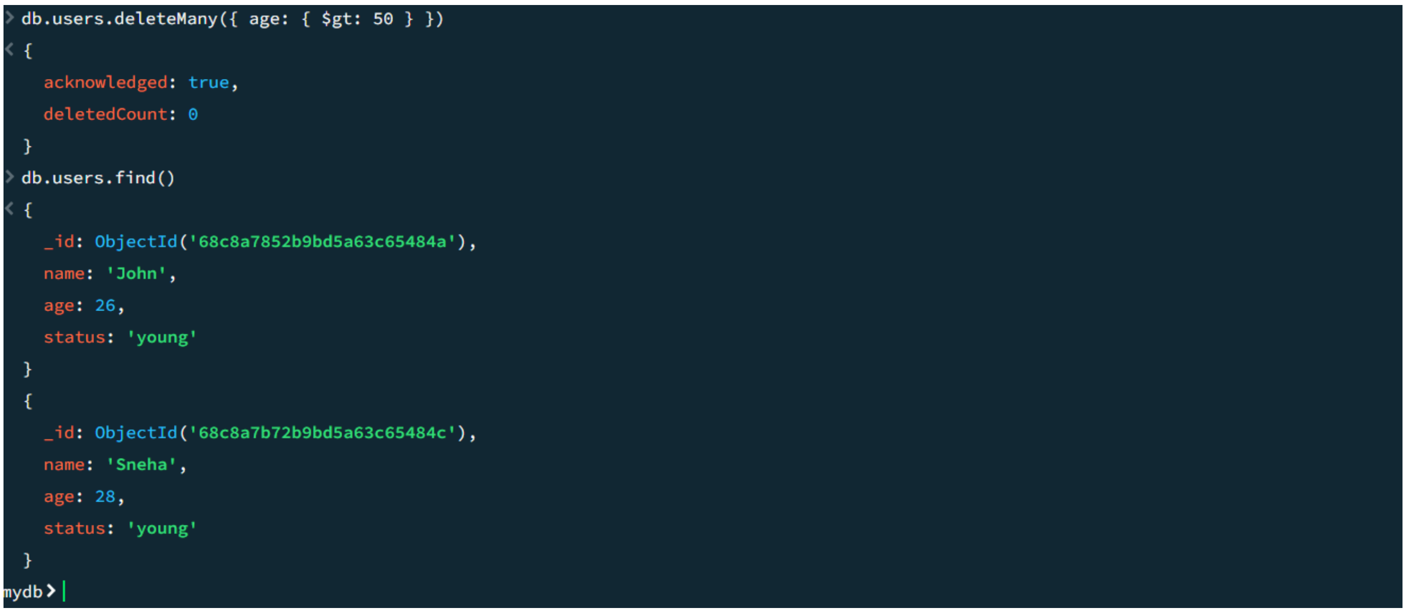
F. Delete Commands

1. **Delete one:** Removes the first document matching based on condition
   Command: db.users.deleteOne({ name: "Amit" })



```
>_MONGOSH
> use mydb
< switched to db mydb
> db.users.deleteOne({ name: "Amit" })
< {
    acknowledged: true,
    deletedCount: 1
  }
> db.users.find()
< {
    _id: ObjectId('68c8a7852b9bd5a63c65484a'),
    name: 'John',
    age: 26,
    status: 'young'
  }
  {
    _id: ObjectId('68c8a7b72b9bd5a63c65484c'),
    name: 'Sneha',
    age: 28,
    status: 'young'
  }
mydb > |
```

2. **Delete many:** Removes all users where age > 50.
   Command: db.users.deleteMany({ age: { $gt: 50 } })



```
> db.users.deleteMany({ age: { $gt: 50 } })
< {
    acknowledged: true,
    deletedCount: 0
  }
> db.users.find()
< {
    _id: ObjectId('68c8a7852b9bd5a63c65484a'),
    name: 'John',
    age: 26,
    status: 'young'
  }
  {
    _id: ObjectId('68c8a7b72b9bd5a63c65484c'),
    name: 'Sneha',
    age: 28,
    status: 'young'
  }
mydb>
```

3. findOneAndDelete()?

- It **finds the first matching document** based on the filter.
- **Deletes** that document.
- **Returns** the deleted document (so you know what was removed).

Syntax:

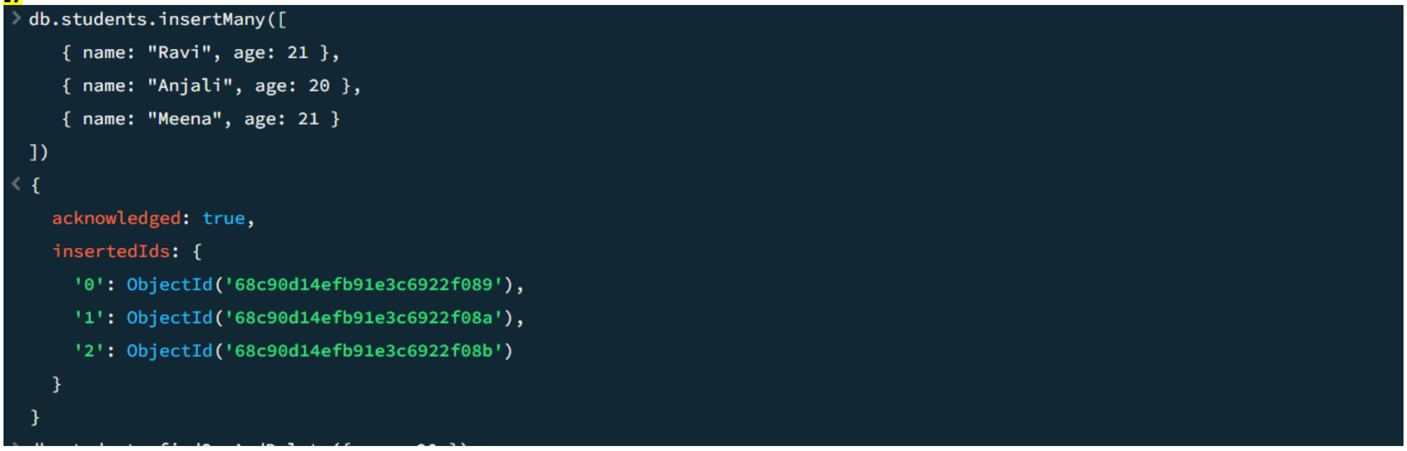
db.collection.findOneAndDelete(<filter>, <options>)

Parameters:

- `<filter>` → Query condition to find the document.
- `<options>` → (Optional) extra settings like `sort`, `projection`, etc.

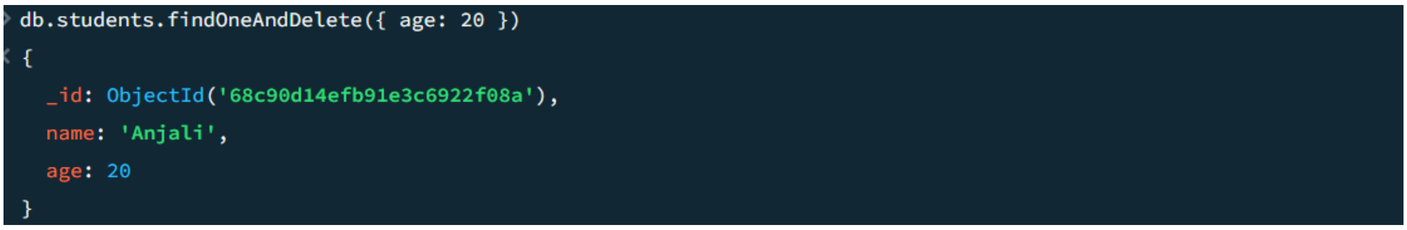
Example:-1 **Delete a document by a field**

db.students.insertMany([
{ name: "Ravi", age: 21 },
{ name: "Anjali", age: 20 },
{ name: "Meena", age: 21 }
])



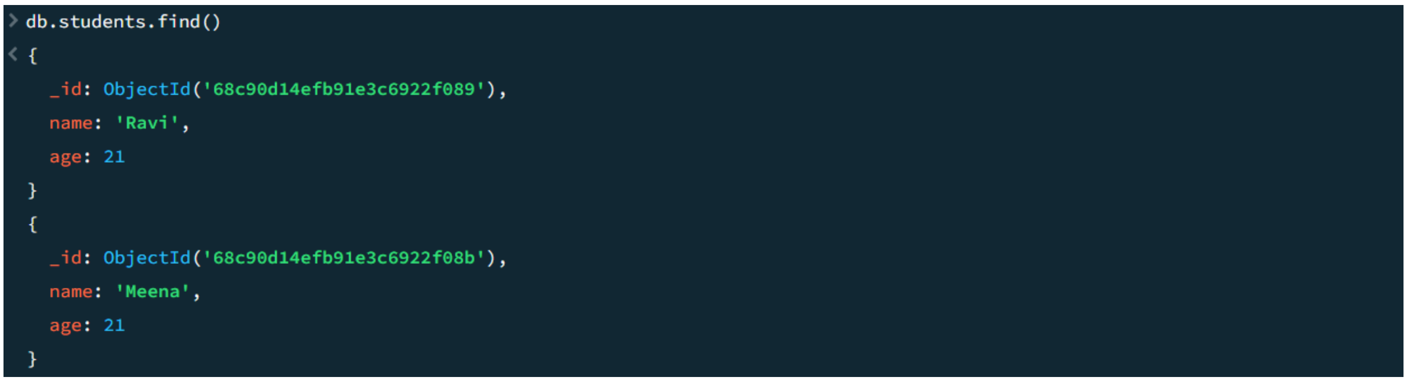
```
> db.students.insertMany([
    { name: "Ravi", age: 21 },
    { name: "Anjali", age: 20 },
    { name: "Meena", age: 21 }
  ])
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('68c90d14efb91e3c6922f089'),
      '1': ObjectId('68c90d14efb91e3c6922f08a'),
      '2': ObjectId('68c90d14efb91e3c6922f08b')
    }
  }
```

db.students.findOneAndDelete({ age: 20 })



```
> db.students.findOneAndDelete({ age: 20 })
< {
    _id: ObjectId('68c90d14efb91e3c6922f08a'),
    name: 'Anjali',
    age: 20
  }
```

Now check the collection

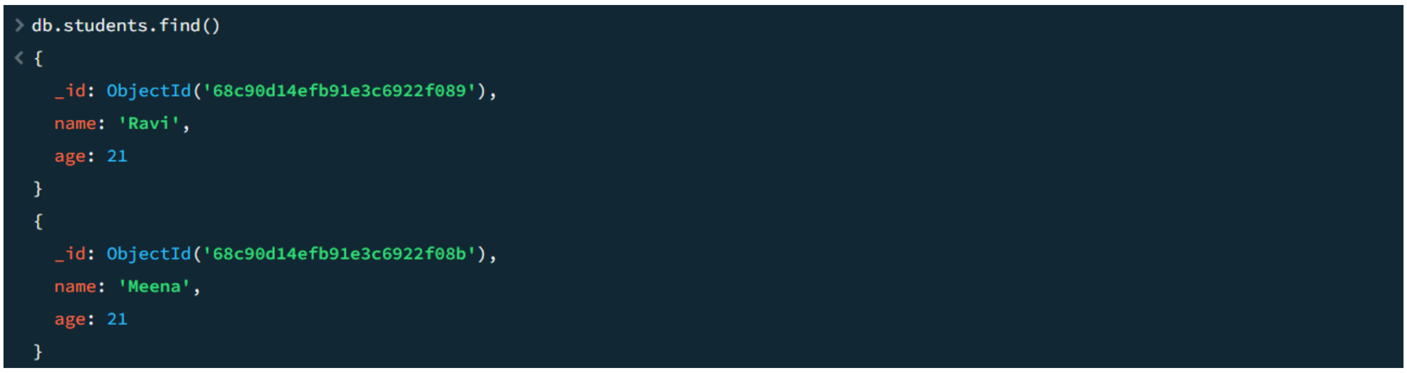


```
> db.students.find()
< {
    _id: ObjectId('68c90d14efb91e3c6922f089'),
    name: 'Ravi',
    age: 21
  }
  {
    _id: ObjectId('68c90d14efb91e3c6922f08b'),
    name: 'Meena',
    age: 21
  }
```

Example:-2 Delete first matching when multiple docs exist

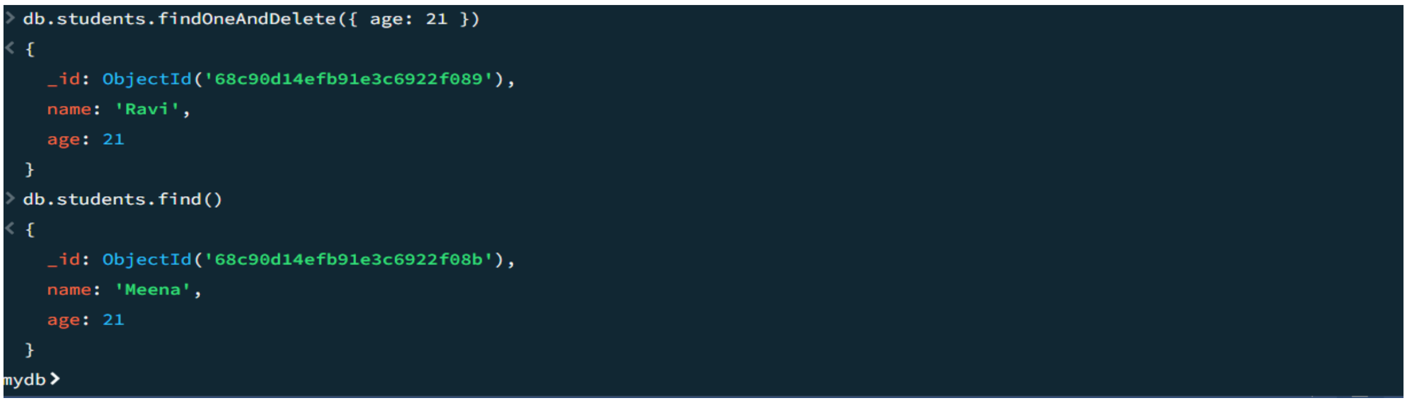
Command : db.students.findOneAndDelete({ age: 21 })

Note: Even though two students have `age: 21`, only the **first one MongoDB finds** gets deleted.



```
> db.students.find()
< {
    _id: ObjectId('68c90d14efb91e3c6922f089'),
    name: 'Ravi',
    age: 21
  }
  {
    _id: ObjectId('68c90d14efb91e3c6922f08b'),
    name: 'Meena',
    age: 21
  }
```

Command : db.students.findOneAndDelete({ age: 21 })

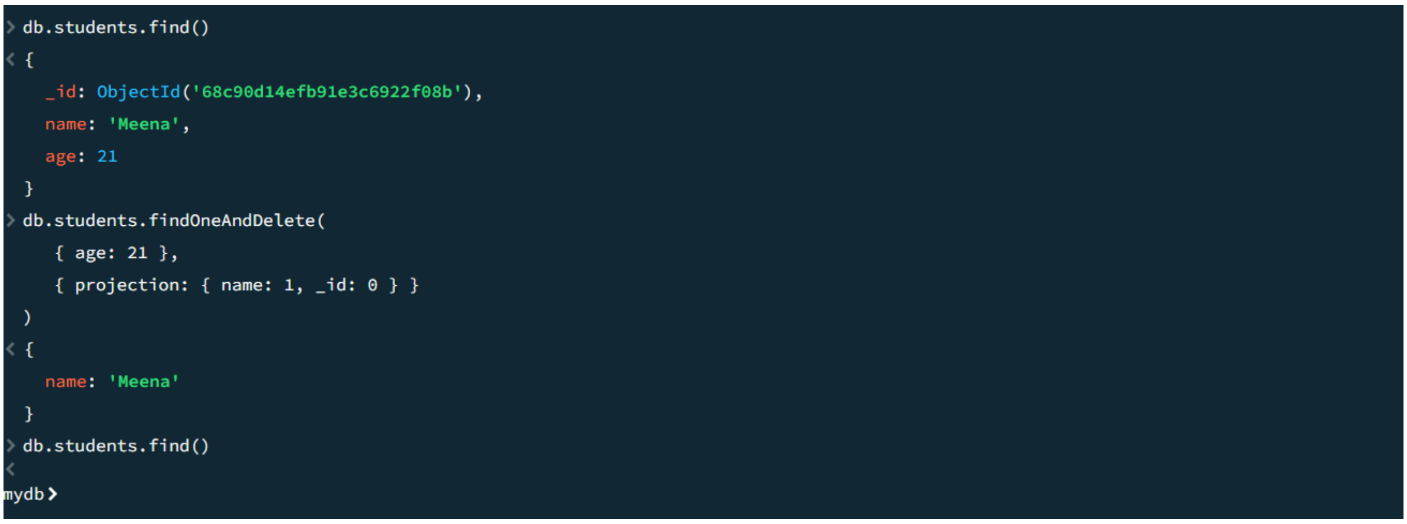


```
> db.students.findOneAndDelete({ age: 21 })
< {
    _id: ObjectId('68c90d14efb91e3c6922f089'),
    name: 'Ravi',
    age: 21
  }
> db.students.find()
< {
    _id: ObjectId('68c90d14efb91e3c6922f08b'),
    name: 'Meena',
    age: 21
  }
mydb>
```

Example-3. Return only specific fields (projection)

```
db.students.findOneAndDelete(
  { age: 21 },
  { projection: { name: 1, _id: 0 } }
)
```

Explanation: It deleted the whole document, but only returned the `name` field.



```
> db.students.find()
< {
    _id: ObjectId('68c90d14efb91e3c6922f08b'),
    name: 'Meena',
    age: 21
  }
> db.students.findOneAndDelete(
    { age: 21 },
    { projection: { name: 1, _id: 0 } }
  )
< {
    name: 'Meena'
  }
> db.students.find()
<
mydb>
```

G. Sorting, Limiting, Skipping

Example: Students collection contain ns 13 documents

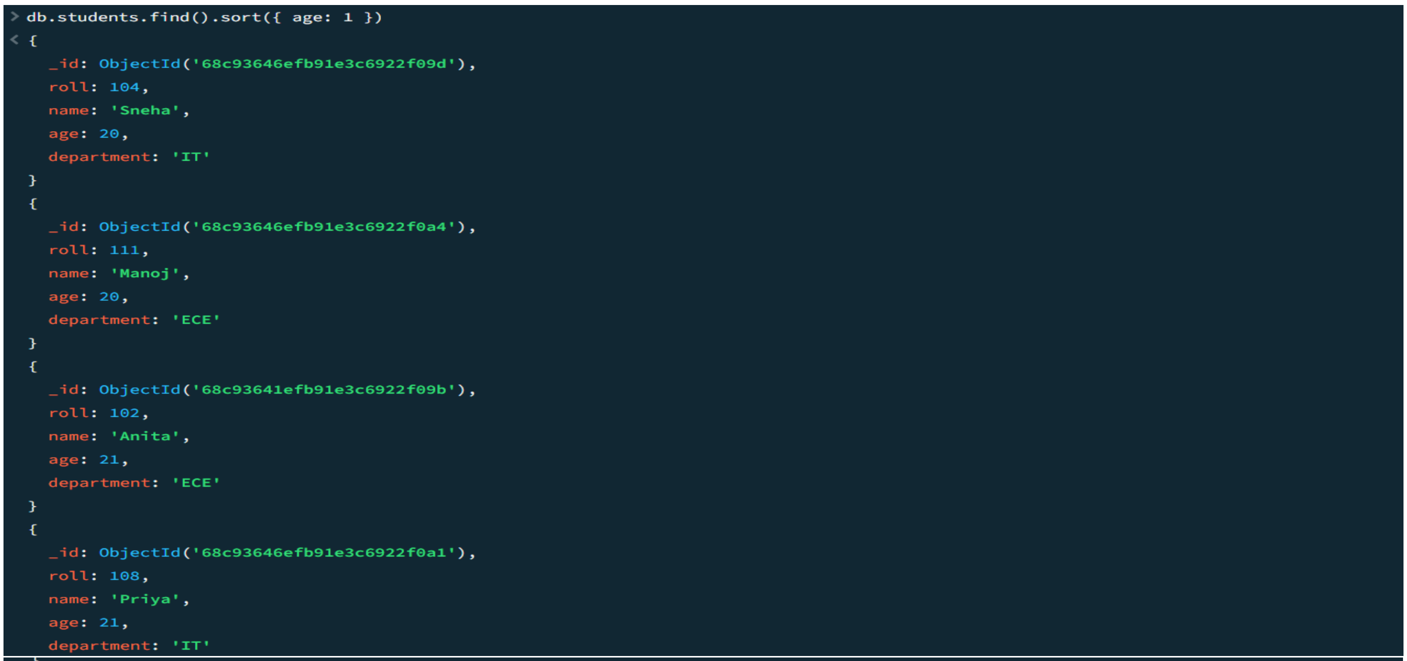
```
db.students.insertMany(
[
 { roll: 101, name: "Rakesh", age: 22, department: "CSE" },
{ roll: 102, name: "Anita", age: 21, department: "ECE" },
 { roll: 103, name: "Rahul", age: 23, department: "CSE" } ],
 { roll: 104, name: "Sneha", age: 20, department: "IT" },
  { roll: 105, name: "Vikas", age: 24, department: "ECE" },
  { roll: 106, name: "Meena", age: 22, department: "EEE" },
  { roll: 107, name: "Arjun", age: 25, department: "CSE" },
  { roll: 108, name: "Priya", age: 21, department: "IT" },
  { roll: 109, name: "Kiran", age: 23, department: "ME" },
  { roll: 110, name: "Divya", age: 22, department: "CSE" },
  { roll: 111, name: "Manoj", age: 20, department: "ECE" },
  { roll: 112, name: "Ritu", age: 24, department: "EEE" },
  { roll: 113, name: "Amit", age: 21, department: "ME" }
]
)
```

1. Sort ascending: Lists users ordered in **ascending** based on key.

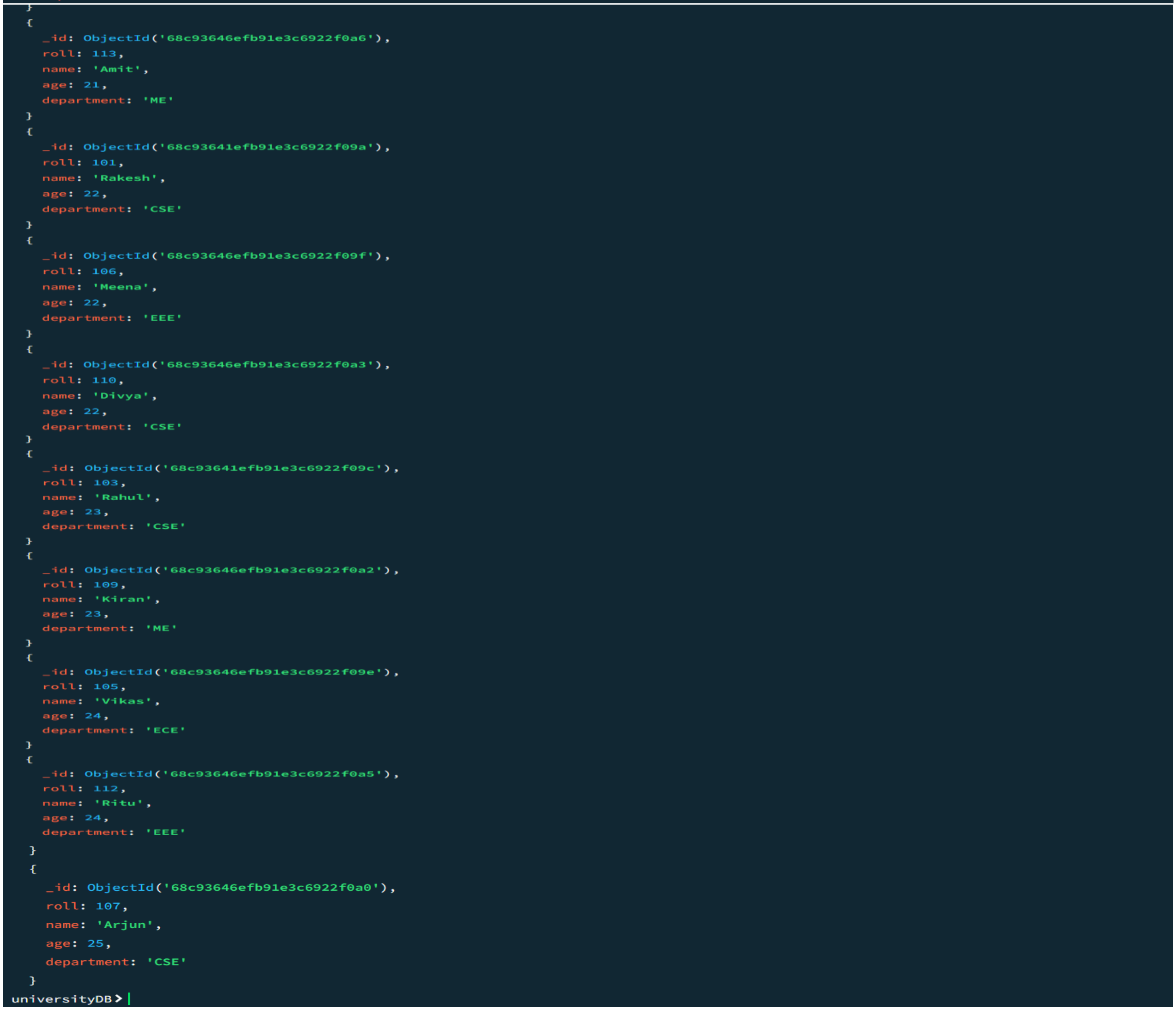
Example: `db.students.find().sort({ age: 1 })`

Explanation: Lists users ordered by age **ascending**.(Smallest age first)

Output:



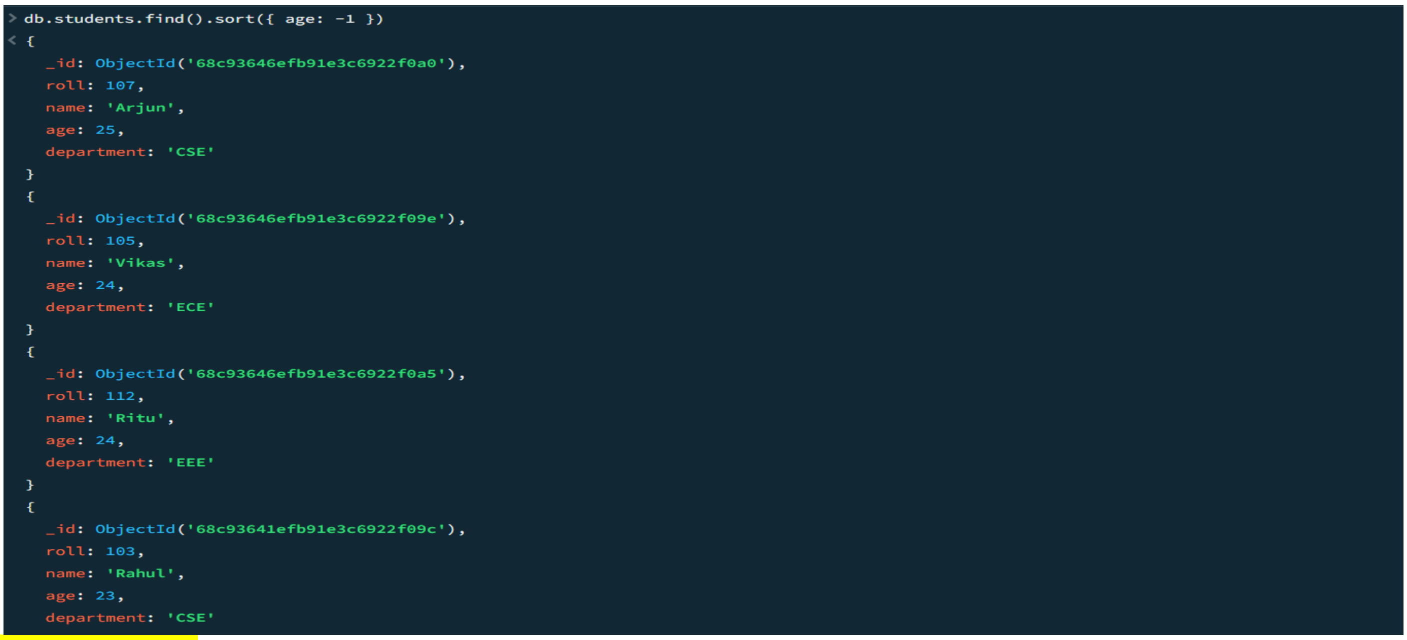
```
> db.students.find().sort({ age: 1 })
< {
    _id: ObjectId('68c93646efb91e3c6922f09d'),
    roll: 104,
    name: 'Sneha',
    age: 20,
    department: 'IT'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a4'),
    roll: 111,
    name: 'Manoj',
    age: 20,
    department: 'ECE'
  }
  {
    _id: ObjectId('68c93641efb91e3c6922f09b'),
    roll: 102,
    name: 'Anita',
    age: 21,
    department: 'ECE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a1'),
    roll: 108,
    name: 'Priya',
    age: 21,
    department: 'IT'
```



```
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a6'),
    roll: 113,
    name: 'Amit',
    age: 21,
    department: 'ME'
  }
  {
    _id: ObjectId('68c93641efb91e3c6922f09a'),
    roll: 101,
    name: 'Rakesh',
    age: 22,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09f'),
    roll: 106,
    name: 'Meena',
    age: 22,
    department: 'EEE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a3'),
    roll: 110,
    name: 'Divya',
    age: 22,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93641efb91e3c6922f09c'),
    roll: 103,
    name: 'Rahul',
    age: 23,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a2'),
    roll: 109,
    name: 'Kiran',
    age: 23,
    department: 'ME'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09e'),
    roll: 105,
    name: 'Vikas',
    age: 24,
    department: 'ECE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a5'),
    roll: 112,
    name: 'Ritu',
    age: 24,
    department: 'EEE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a0'),
    roll: 107,
    name: 'Arjun',
    age: 25,
    department: 'CSE'
  }
universityDB>
```

2. Sort descending: Lists users ordered in descending based on key.

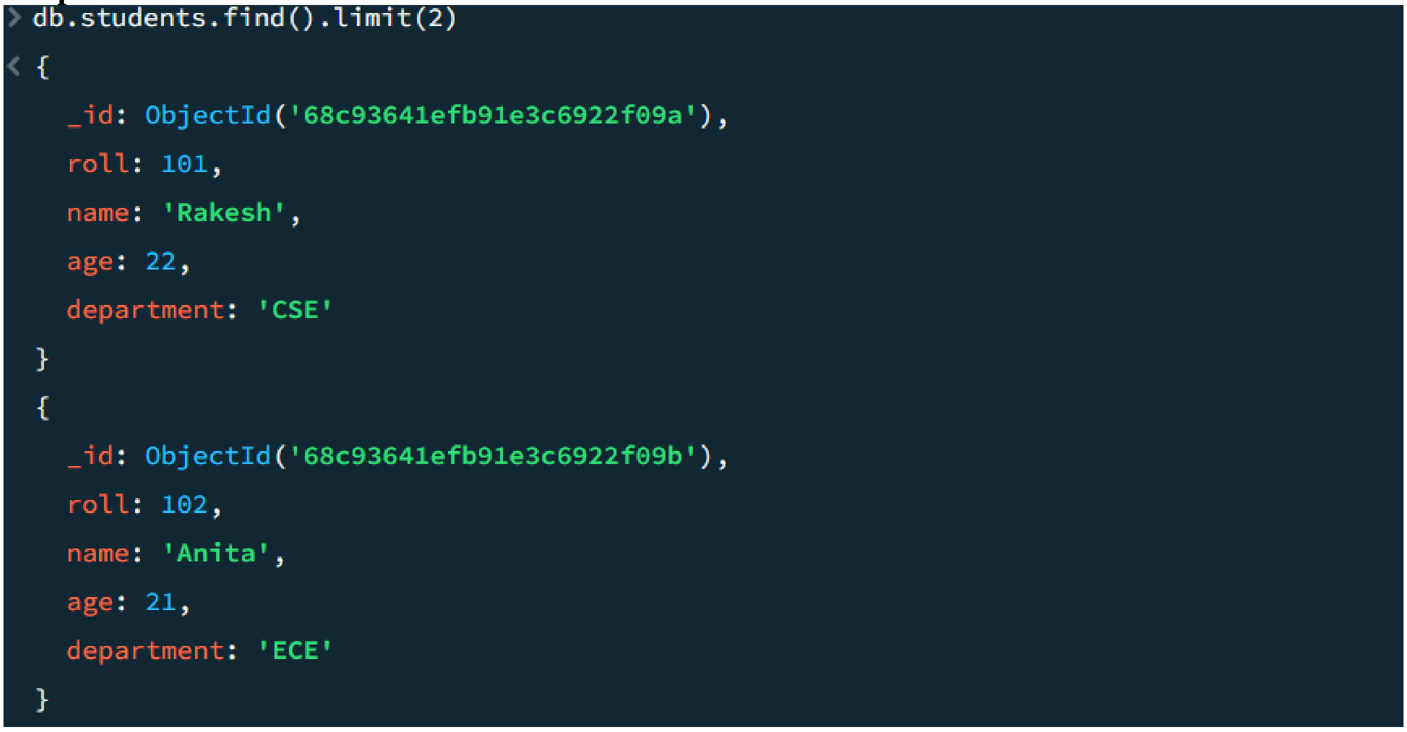
Example: db.students.find().sort({ age: -1 })
Explanation: Lists users ordered by age **descending**. (Largest age first)



```
> db.students.find().sort({ age: -1 })
< {
    _id: ObjectId('68c93646efb91e3c6922f0a0'),
    roll: 107,
    name: 'Arjun',
    age: 25,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09e'),
    roll: 105,
    name: 'Vikas',
    age: 24,
    department: 'ECE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a5'),
    roll: 112,
    name: 'Ritu',
    age: 24,
    department: 'EEE'
  }
  {
    _id: ObjectId('68c93641efb91e3c6922f09c'),
    roll: 103,
    name: 'Rahul',
    age: 23,
    department: 'CSE'
```

3. Limit results: ☞ Shows only limited users based on limit number.

Example: db.students.find().limit(2) // it returns only two results.
Output:

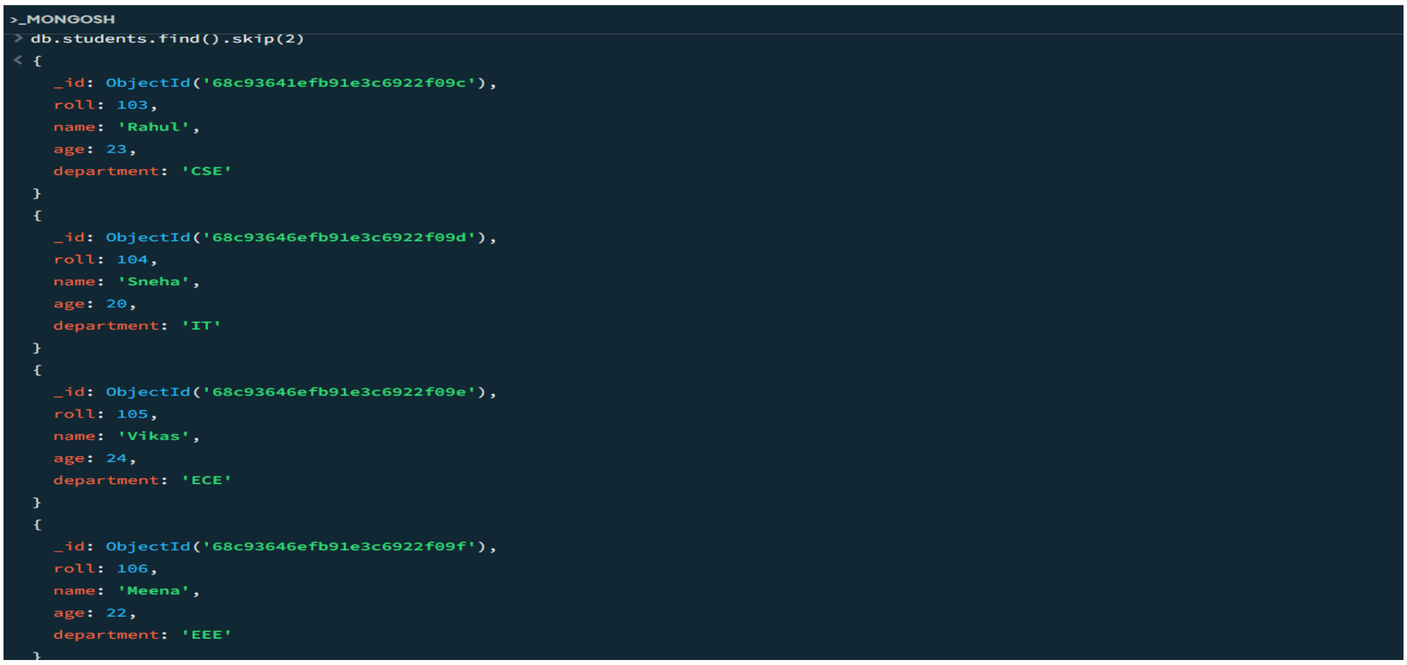


```
> db.students.find().limit(2)
< {
    _id: ObjectId('68c93641efb91e3c6922f09a'),
    roll: 101,
    name: 'Rakesh',
    age: 22,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93641efb91e3c6922f09b'),
    roll: 102,
    name: 'Anita',
    age: 21,
    department: 'ECE'
  }
```

4. Skip results : **Skips the first results based on number and shows the rest.**

Example: db.studnets.find().skip(2) // skip 1st two results and shows rest of the documents

Output:

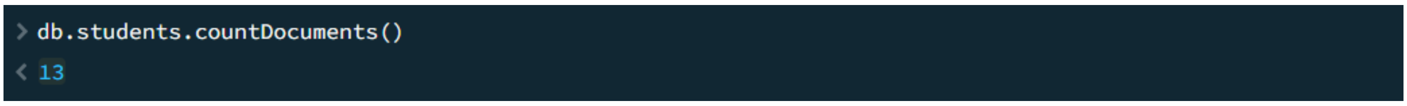


```
>_MONGOSH
> db.students.find().skip(2)
< {
    _id: ObjectId('68c93641efb91e3c6922f09c'),
    roll: 103,
    name: 'Rahul',
    age: 23,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09d'),
    roll: 104,
    name: 'Sneha',
    age: 20,
    department: 'IT'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09e'),
    roll: 105,
    name: 'Vikas',
    age: 24,
    department: 'ECE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09f'),
    roll: 106,
    name: 'Meena',
    age: 22,
    department: 'EEE'
  }
```

Query-1: Suppose you want to **see all students except the last 6**.

Step 1: Count total documents

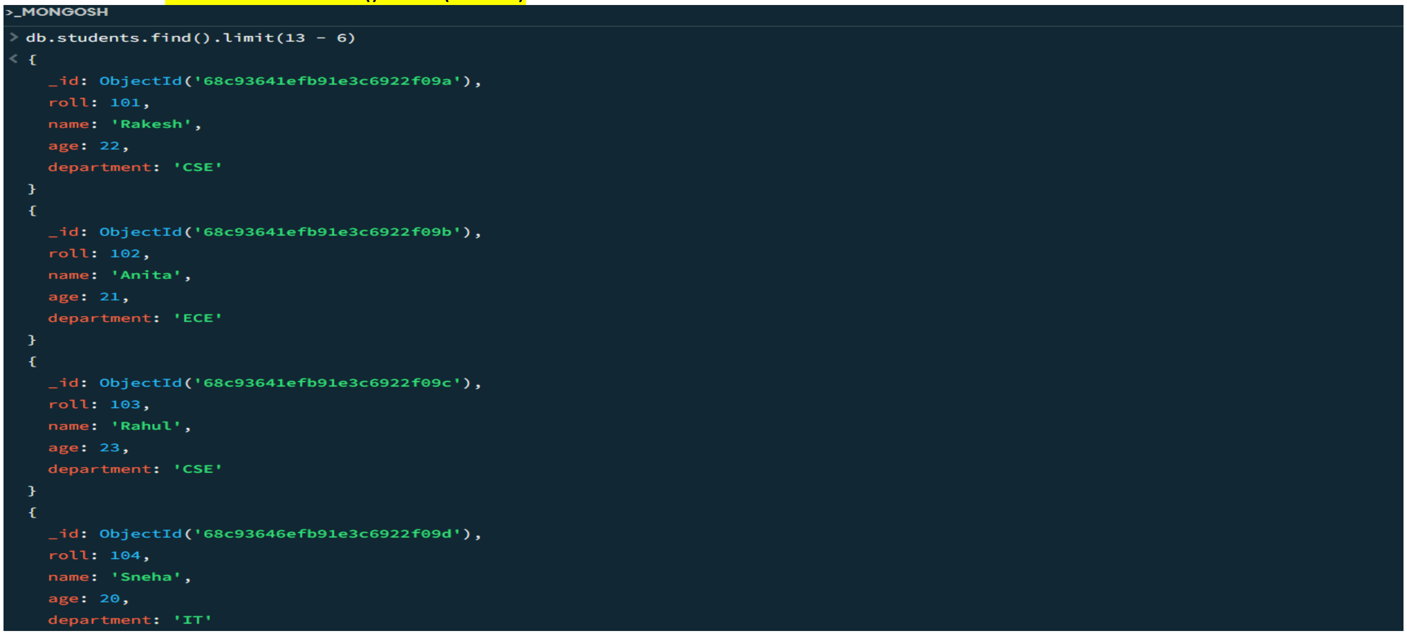
Command: db.students.countDocuments() // it returns **13**.



```
> db.students.countDocuments()
< 13
```

Step 2: Use limit(total - n) If you want to skip last `6`, do:

Command: db.students.find().limit(13 - 6)



```
>_MONGOSH
> db.students.find().limit(13 - 6)
< {
    _id: ObjectId('68c93641efb91e3c6922f09a'),
    roll: 101,
    name: 'Rakesh',
    age: 22,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93641efb91e3c6922f09b'),
    roll: 102,
    name: 'Anita',
    age: 21,
    department: 'ECE'
  }
  {
    _id: ObjectId('68c93641efb91e3c6922f09c'),
    roll: 103,
    name: 'Rahul',
    age: 23,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09d'),
    roll: 104,
    name: 'Sneha',
    age: 20,
    department: 'IT'
```

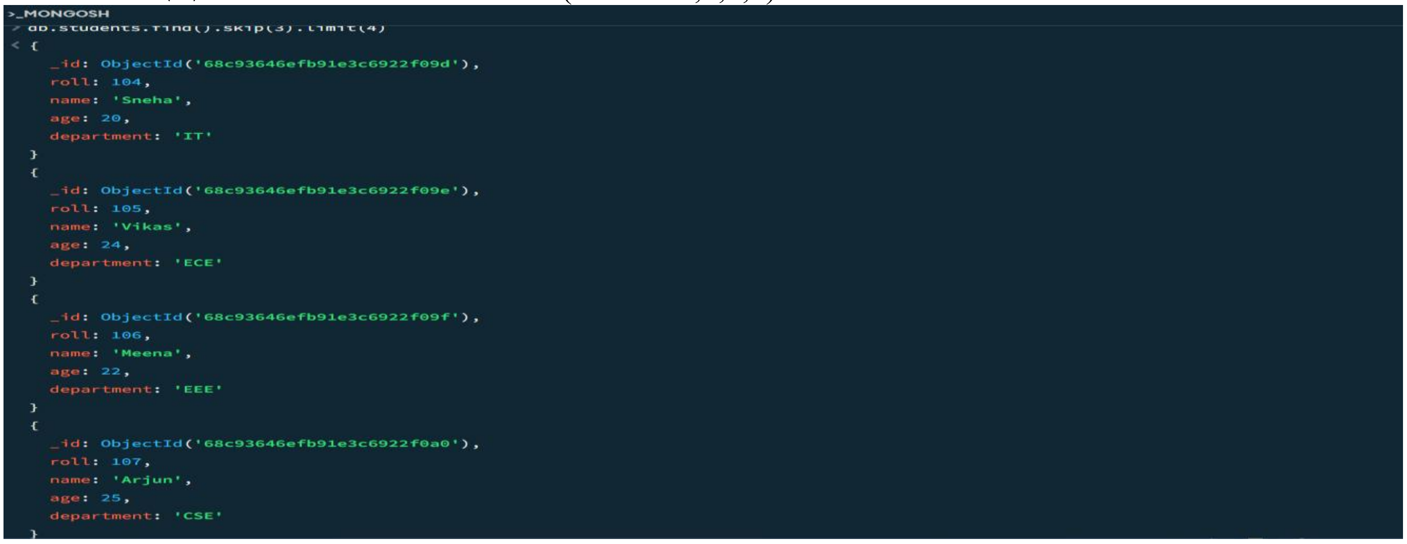
Query-2: If you want results between **index X and Y** (like arrays in JavaScript):

Command: db.students.find().skip(X).limit(Y - X + 1)

Example: **Records from index** 3 to 6 (Index here means 0-based, like arrays: first doc = index 0)

Command : db.students.find().skip(3).limit(4)

☞ **Explanation:**

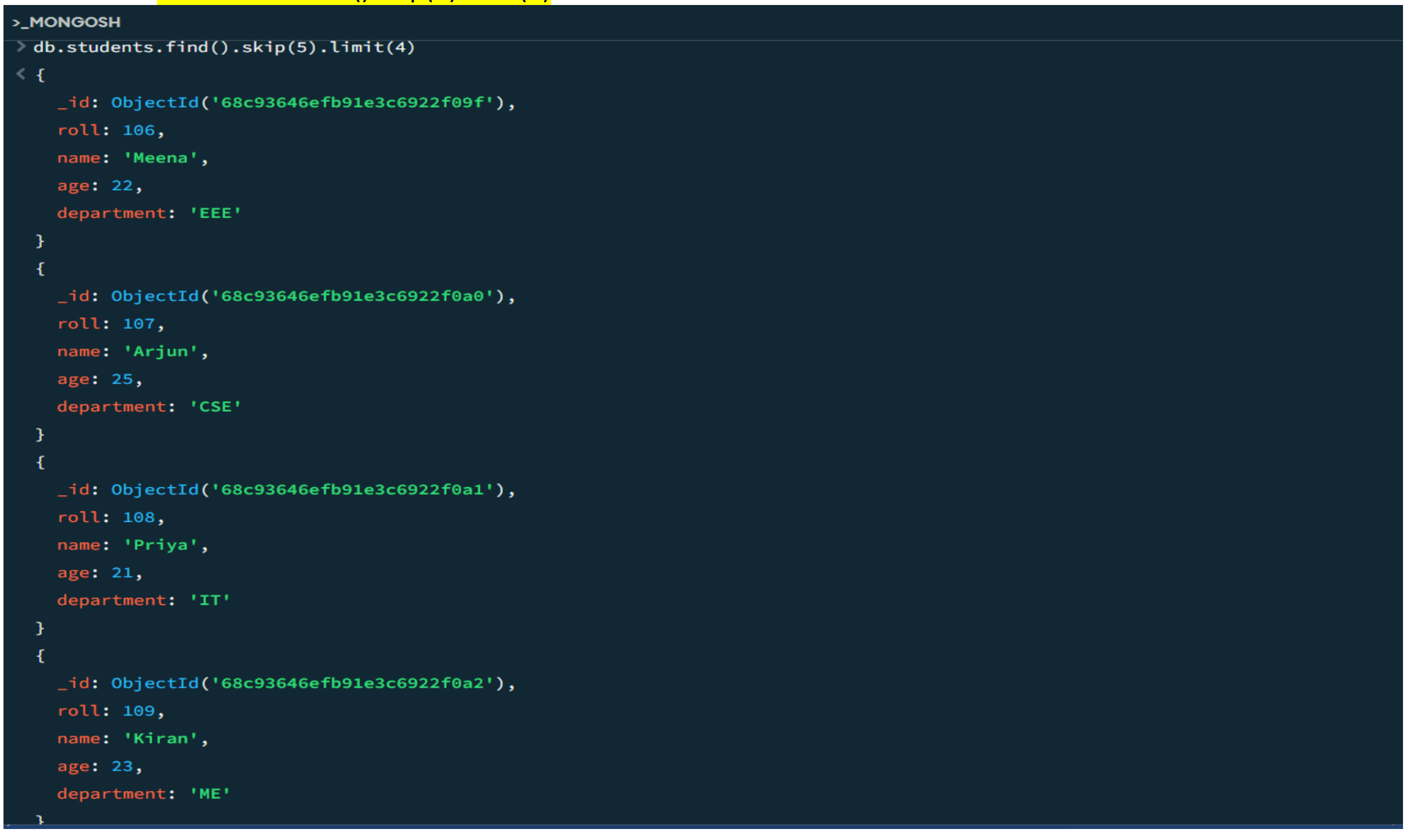
- `.skip(3)` → skips first 3 documents (0,1,2)
- `.limit(4)` → takes next 4 documents (indexes 3,4,5,6)
- 

```
>_MONGOSH
> db.students.find().skip(3).limit(4)
< {
    _id: ObjectId('68c93646efb91e3c6922f09d'),
    roll: 104,
    name: 'Sneha',
    age: 20,
    department: 'IT'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09e'),
    roll: 105,
    name: 'Vikas',
    age: 24,
    department: 'ECE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09f'),
    roll: 106,
    name: 'Meena',
    age: 22,
    department: 'EEE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a0'),
    roll: 107,
    name: 'Arjun',
    age: 25,
    department: 'CSE'
  }
```

Note : MongoDB returns results in **insertion order** (natural order).

Query: **Records from index** 5 to 8

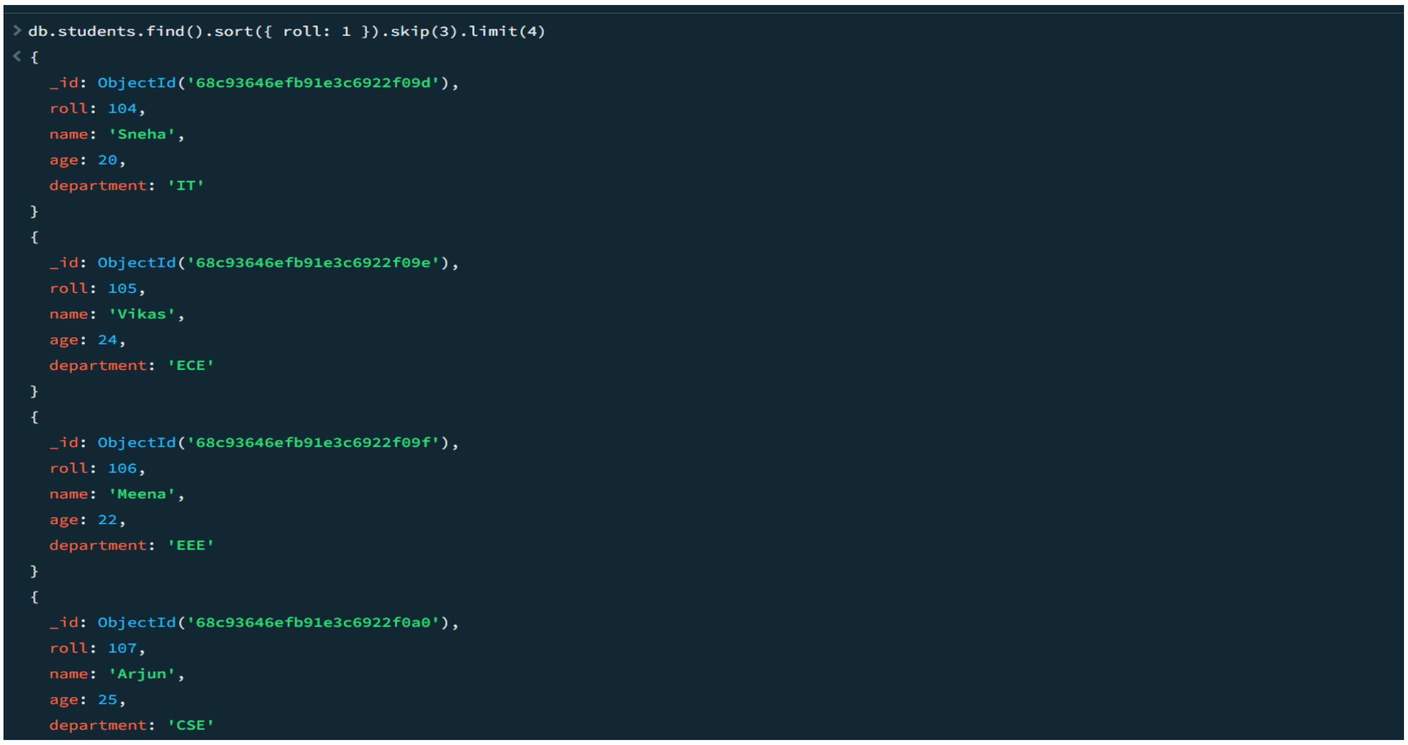
Command : db.students.find().skip(5).limit(4)



```
>_MONGOSH
> db.students.find().skip(5).limit(4)
< {
    _id: ObjectId('68c93646efb91e3c6922f09f'),
    roll: 106,
    name: 'Meena',
    age: 22,
    department: 'EEE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a0'),
    roll: 107,
    name: 'Arjun',
    age: 25,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a1'),
    roll: 108,
    name: 'Priya',
    age: 21,
    department: 'IT'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a2'),
    roll: 109,
    name: 'Kiran',
    age: 23,
    department: 'ME'
  }
```

Query-3: If you want consistent results, always combine with **sort**:

Command: db.students.find().sort({ roll: 1 }).skip(3).limit(4)



```
> db.students.find().sort({ roll: 1 }).skip(3).limit(4)
< {
    _id: ObjectId('68c93646efb91e3c6922f09d'),
    roll: 104,
    name: 'Sneha',
    age: 20,
    department: 'IT'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09e'),
    roll: 105,
    name: 'Vikas',
    age: 24,
    department: 'ECE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f09f'),
    roll: 106,
    name: 'Meena',
    age: 22,
    department: 'EEE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a0'),
    roll: 107,
    name: 'Arjun',
    age: 25,
    department: 'CSE'
```

H. Indexing

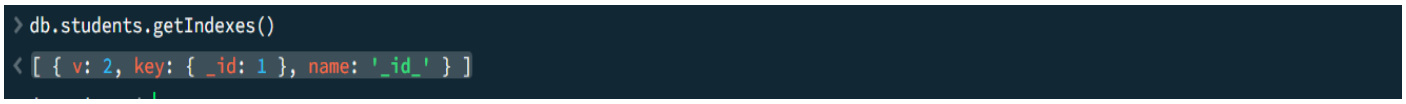
1. What are Indexes?

- An **index** is like the index of a book 🕮 – it helps MongoDB **find data faster**.
- Without indexes, MongoDB must scan **all documents** (slow).
- With indexes, it can quickly jump to the right documents.

2. Default Index (`_id`): Every MongoDB collection automatically creates an index on the **`_id` field.**

- This ensures **uniqueness** (no two documents can have the same `_id`).
- That's why you see `{ key: { _id: 1 }, name: "_id_" }`.

3. Understanding Output



```
> db.students.getIndexes()
< [ { v: 2, key: { _id: 1 }, name: '_id_' } ]
```

- `v: 2` → Index version (internal, you can ignore this).
- `key: { _id: 1 }` → Indexed field.
  - `_id: 1` means ascending order index on `_id`.
- `name: "_id_"` → Name of the index (system gives `_id_`).

So this simply means:
☞ **Your collection has only one index, on `_id` (ascending).**

4. Creating Your Own Index

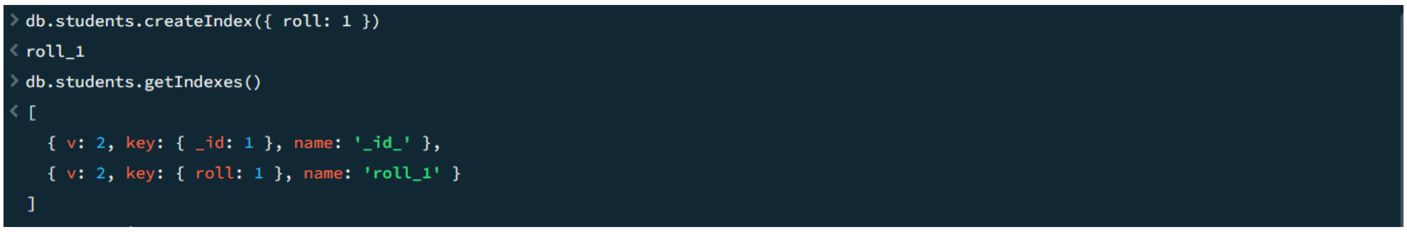
Example: You want to search students by `roll` quickly.

Command: db.students.createIndex({ roll: 1 })

Now check again:

Command: db.students.getIndexes()

You'll see something like:



```
> db.students.createIndex({ roll: 1 })
< roll_1
> db.students.getIndexes()
< [
    { v: 2, key: { _id: 1 }, name: '_id_' },
    { v: 2, key: { roll: 1 }, name: 'roll_1' }
  ]
```

Summary:

- By default, only `_id` is indexed.
- You can create indexes on other fields (`roll`, `email`, `age`, etc.) for **fast queries**.
- Use `getIndexes()` to see all.

I. Aggregation Pipeline (Powerful Analytics)

What is Aggregation?

Aggregation in MongoDB is a **powerful framework** used to process, transform, and analyze data stored in collections.
It works like a **data pipeline**: documents flow through multiple stages, and each stage transforms the documents before passing them to the next stage.

Think of it like:
Raw Data → (pipeline stages) → Processed Results (like reports, charts, analytics)

Why Use Aggregation?

- To **filter, group, and sort** documents.
- To **perform calculations** (sum, average, min, max, count).
- To **reshape documents** (project only selected fields).
- To do **complex analytics** directly inside MongoDB (instead of fetching all data into your app).

Syntax:

```
        db.collection.aggregate([
          { stage1 },
          { stage2 },
          { stage3 }
])
```

Important Aggregation Stages

Stage	Purpose	Example
`$match`	Filters documents (like `find()`)	`{ $match: { department: "CSE" } }`
`$group`	Groups documents and applies aggregate functions	`{ $group: { _id: "$department", total: { $sum: 1 } } }`
`$project`	Selects/reshapes fields (like SELECT in SQL)	`{ $project: { name: 1, age: 1 } }`
`$sort`	Sorts results	`{ $sort: { age: -1 } }`
`$limit`	Limits the number of results	`{ $limit: 5 }`
`$skip`	Skips first N documents	`{ $skip: 3 }`
`$lookup`	Joins another collection (like SQL JOIN)	`{ $lookup: { from: "orders", localField: "id", foreignField: "custId", as: "orders" } }`
`$unwind`	Breaks an array into multiple documents	`{ $unwind: "$subjects" }`

Collection: `students`

```
[
  { "_id": "68c93641efb91e3c6922f09a", "roll": 101, "name": "Rakesh", "age": 22, "department": "CSE" },
  { "_id": "68c93641efb91e3c6922f09b", "roll": 102, "name": "Anita", "age": 21, "department": "ECE" },
  { "_id": "68c93641efb91e3c6922f09c", "roll": 103, "name": "Rahul", "age": 23, "department": "CSE" },
  { "_id": "68c93646efb91e3c6922f09d", "roll": 104, "name": "Sneha", "age": 20, "department": "IT" },
  { "_id": "68c93646efb91e3c6922f09e", "roll": 105, "name": "Vikas", "age": 24, "department": "ECE" },
  { "_id": "68c93646efb91e3c6922f09f", "roll": 106, "name": "Meena", "age": 22, "department": "EEE" },
  { "_id": "68c93646efb91e3c6922f0a0", "roll": 107, "name": "Arjun", "age": 25, "department": "CSE" },
  { "_id": "68c93646efb91e3c6922f0a1", "roll": 108, "name": "Priya", "age": 21, "department": "IT" },
  { "_id": "68c93646efb91e3c6922f0a2", "roll": 109, "name": "Kiran", "age": 23, "department": "ME" },
  { "_id": "68c93646efb91e3c6922f0a3", "roll": 110, "name": "Divya", "age": 22, "department": "CSE" },
  { "_id": "68c93646efb91e3c6922f0a4", "roll": 111, "name": "Manoj", "age": 20, "department": "ECE" },
  { "_id": "68c93646efb91e3c6922f0a5", "roll": 112, "name": "Ritu", "age": 24, "department": "EEE" },
{ "_id": "68c93646efb91e3c6922f0a6"),  "roll": 113,  "name": "Amit",  "age": 21,  "department": "ME"},
{  "_id": ("68cae19ed35117b6f496eee"),  "roll": 101,  "name": "Rakesh",  "deptId": 1},
{  "_id": ("68cae19ed35117b6f496eef"),  "roll": 102,  "name": "Anita",  "deptId": 2},
{  "_id": ("68cae19ed35117b6f496ef0"),  "roll": 103,  "name": "Rahul",  "deptId": 1}
]
```

Use above collection for writing Queries on Aggregation

1. Count documents:

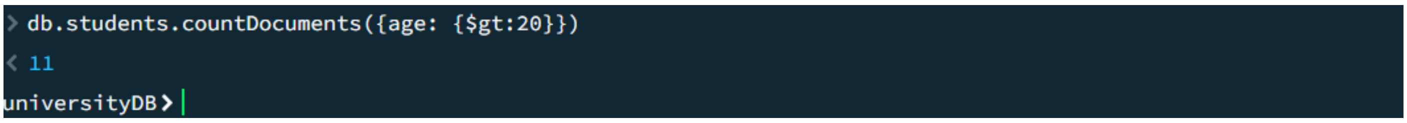
MongoDB provides two main ways **to count documents:**

i. countDocuments() (direct method, not aggregation)

- Used outside aggregation.
- Example:

Command: db.students.countDocuments({ age: { $gt: 20 } })

☞ **Counts users where age > 20.**



```
> db.students.countDocuments({age: {$gt:20}})
< 11
universityDB>
```

ii. $count (Aggregation Stage)

- Used **inside an aggregation pipeline**.
- Returns the number of documents that pass through the pipeline.

Syntax:

```
db.collection.aggregate([
  { $match: <condition> },
  { $count: "<fieldName>" }
])
```

Syntax Explanation

db.collection.aggregate([...])

- Runs an **aggregation pipeline** on the collection.
- Inside `[ ... ]` you define **stages** that will be executed in order.
- Each stage transforms the data before passing it to the next stage.

{$match: <condition>}

- This is an **optional stage** (you may or may not use it).
- Filters the documents based on the given condition.
- Works like `WHERE` in SQL.

Example: { $match: { age: { $gt: 25 } } }

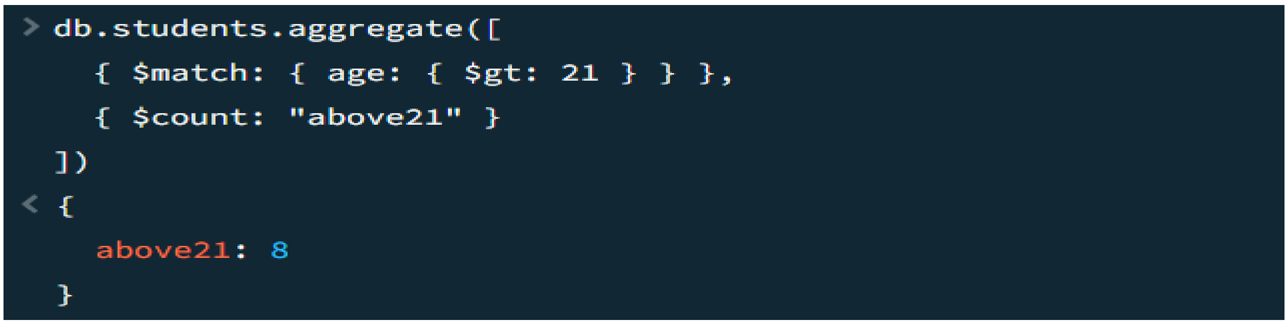
{$count: "<fieldName>"}

- `$count` is a **stage** that counts how many documents reach this point in the pipeline.
- `"<fieldName>"` is the **name of the output field** that will hold the count result.
- This is **mandatory** → you must provide a name.
- The `$count` stage **needs a name** for the resulting field.
- That name can be **anything you choose** (it is just an alias).

Example: using `"above21"`

```
db.students.aggregate([
  { $match: { age: { $gt: 21 } } },
  { $count: "above21" }
])
```

Output:

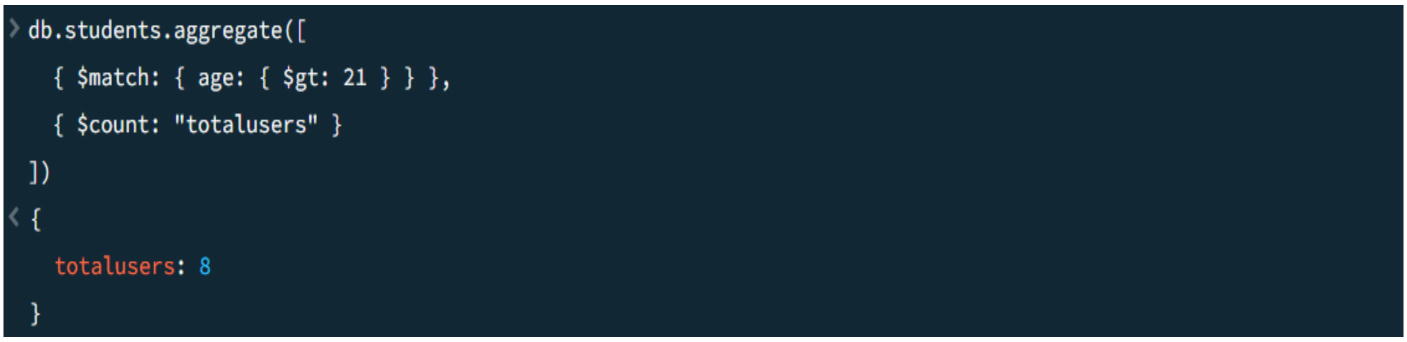


```
> db.students.aggregate([
    { $match: { age: { $gt: 21 } } },
    { $count: "above21" }
  ])
< {
    above21: 8
  }
```

Example 2: Using "totalUsers"

```
db.students.aggregate([
  { $match: { age: { $gt: 21 } } },
  { $count: "totalusers" }
])
```

Output:

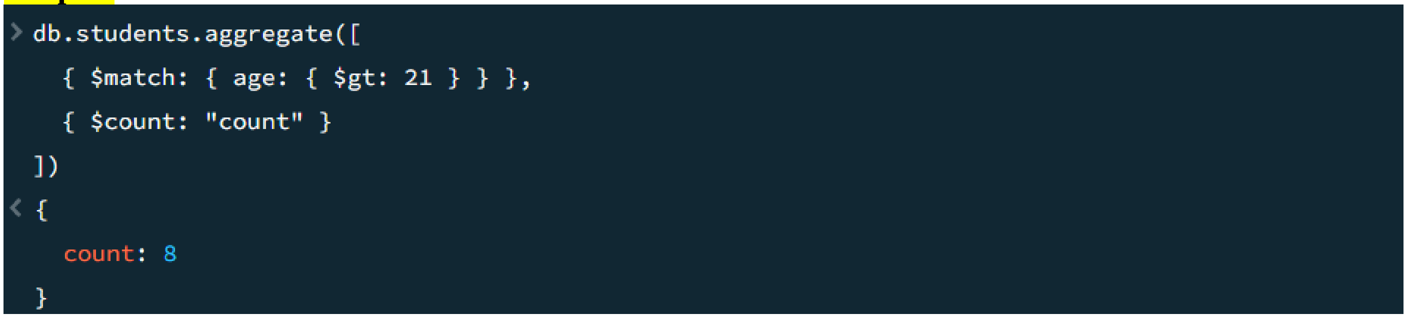


```
> db.students.aggregate([
    { $match: { age: { $gt: 21 } } },
    { $count: "totalusers" }
  ])
< {
    totalusers: 8
  }
```

Example 3: Using "count"

```
db.students.aggregate([
  { $match: { age: { $gt: 21 } } },
  { $count: "count" }
])
```

Output:



```
> db.students.aggregate([
    { $match: { age: { $gt: 21 } } },
    { $count: "count" }
  ])
< {
    count: 8
  }
```

Conclusion:

- The word after `$count` is **not fixed**.
- You **must give some name**, but it can be anything: `"above25"`, `"total"`, `"count"`, `"result"`, etc.

Difference: `countDocuments() vs $count`

Feature	`countDocuments()`	`$count` (Aggregation)
Usage	Direct method on collection	Inside aggregation pipeline
Flexibility	Only counts	Can combine with `$match`, `$group`, `$sort`
Example	`db.users.countDocuments({ age: 20 })`	`db.users.aggregate([{ $match: { age: 20 } }, { $count: "total" }])`

2. Group: The **aggregation framework** processes documents and returns computed results.
It uses stages like `$match`, `$group`, `$sort`, etc.

➢ To **group by age**, we use `$group`.

Syntax:

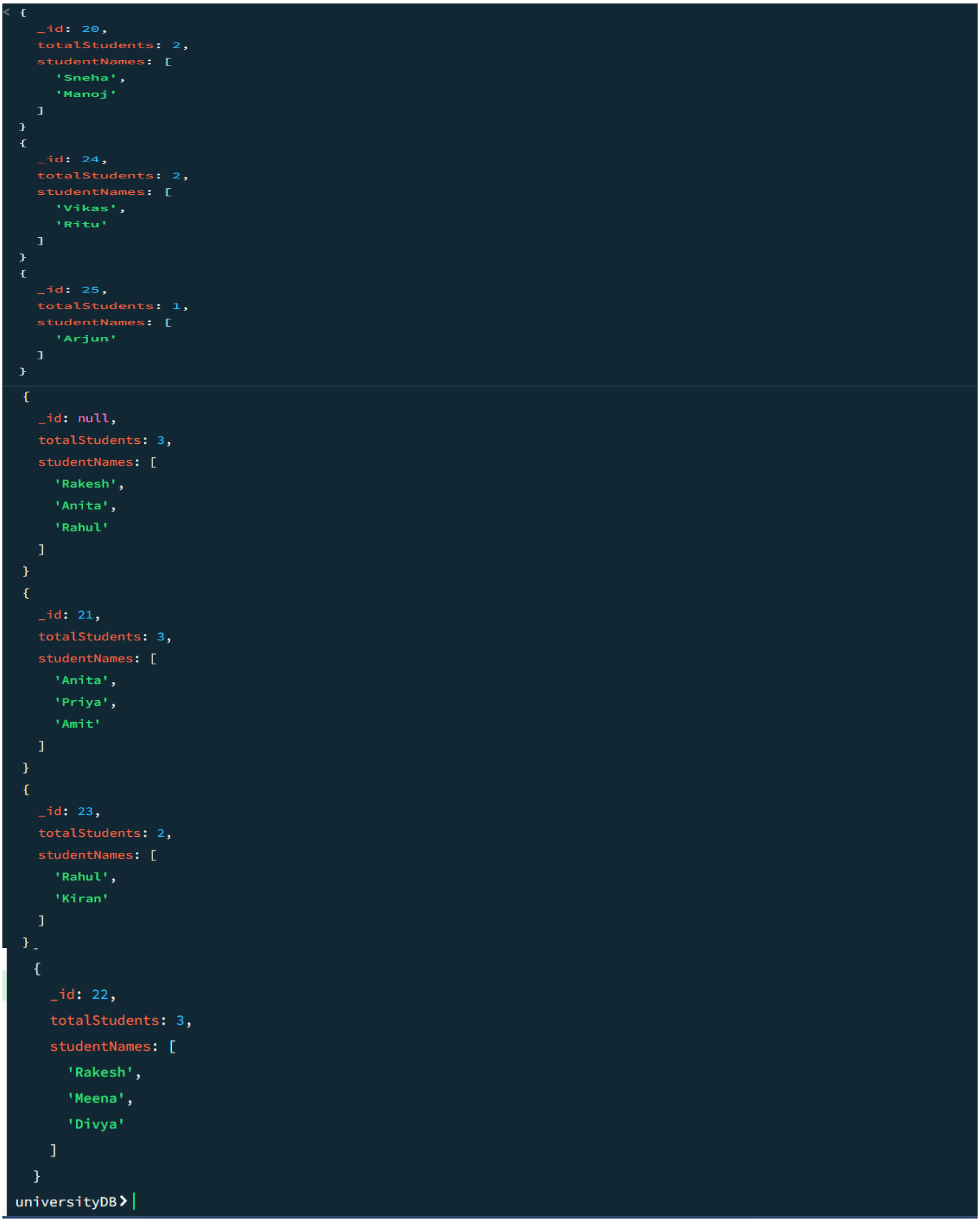
```
db.collection.aggregate([
  {
    $group: {
      _id: "$age",              // group by age field
      count: { $sum: 1 },       // count how many per age
      names: { $push: "$name" } // optional: collect names of that age
    }
  }
])
```

Example-1: Write a MongoDB aggregation query to group students by age, count how many students belong to each age, and list the names of students for each age.

Answer:

```
db.students.aggregate([
  {
    $group: {
      _id: "$age",                    // Grouping by age
      totalStudents: { $sum: 1 },     // Counting students in each group
      studentNames: { $push: "$name" } // Collecting student names
    }
  }
])
```

Output:



```
< {
    _id: 20,
    totalStudents: 2,
    studentNames: [
      'Sneha',
      'Manoj'
    ]
  }
  {
    _id: 24,
    totalStudents: 2,
    studentNames: [
      'Vikas',
      'Ritu'
    ]
  }
  {
    _id: 25,
    totalStudents: 1,
    studentNames: [
      'Arjun'
    ]
  }
```

```
  {
    _id: null,
    totalStudents: 3,
    studentNames: [
      'Rakesh',
      'Anita',
      'Rahul'
    ]
  }
  {
    _id: 21,
    totalStudents: 3,
    studentNames: [
      'Anita',
      'Priya',
      'Amit'
    ]
  }
  {
    _id: 23,
    totalStudents: 2,
    studentNames: [
      'Rahul',
      'Kiran'
    ]
  }
```

```
  {
    _id: 22,
    totalStudents: 3,
    studentNames: [
      'Rakesh',
      'Meena',
      'Divya'
    ]
  }
universityDB>
```

Explanation of above Query:

1. `db.students.aggregate([...])`

- We are running an **aggregation pipeline** on the `students` collection.
- Inside the pipeline, we define stages (here we use only one stage: `$group`).

2. $group

- `$group` is used to **group documents** by a specific field.
- For every unique value of the grouping field, MongoDB creates a **group (bucket)**.

3. _id: "$age"

- `_id` is the **group key**.
- Here we set it to `"$age"`, meaning **group documents by age**.
- So all students with the same `age` will go into the same group.

4. totalStudents: { $sum: 1 }

- `$sum: 1` counts how many documents are in each group.
- This gives us the **total number of students** for that age.
- `$sum` is an **accumulator** in MongoDB aggregation.
- Normally, `$sum` adds up values from documents.
- When we write `1`, it means:"**Add 1 for every document in this group.**"
- So, `$sum: 1` is basically a **counter** → it counts how many documents fall into that group.

5. studentNames: { $push: "$name" }

- `$push` collects values into an array.
- Here, it takes each student's `name` and pushes it into an array for that age group.
- So we get a **list of names** for every age.

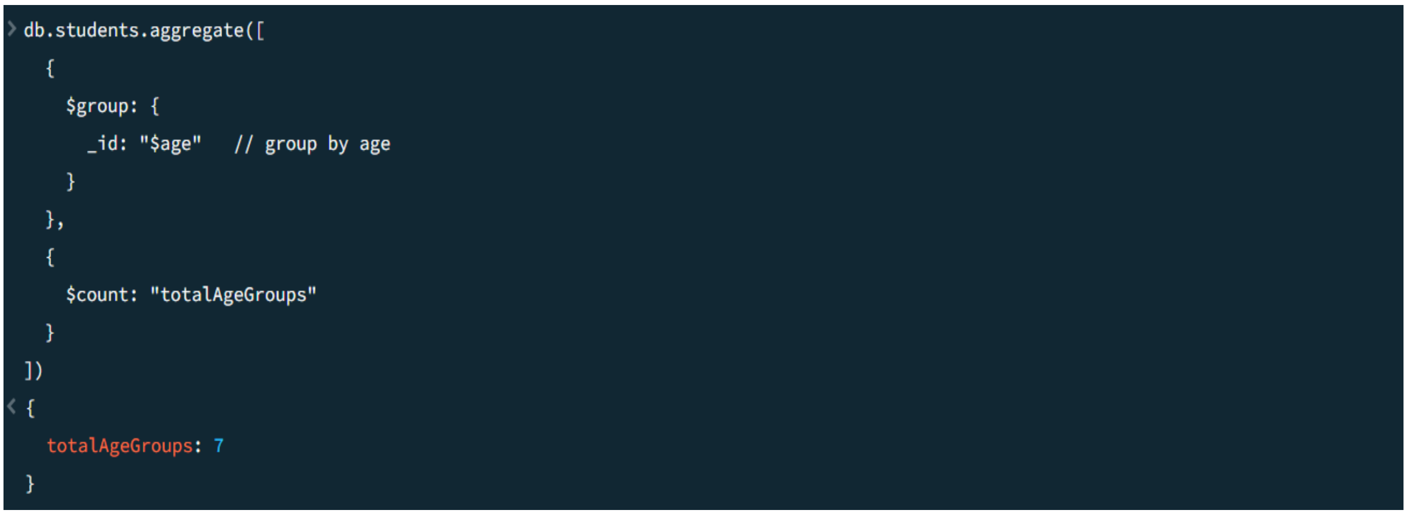
Example-2: If you want **only the count of groups** (not names), **Using `$count` after `$group`**

Answer:

```
db.students.aggregate ([
  {
    $group: {
      _id: "$age"   // group by age
    }
  },
  {
    $count: "totalAgeGroups"
  }
])
```

☞ Above Query gives the **number of different age groups**, not the count of students.

Output:



```
> db.students.aggregate([
    {
      $group: {
        _id: "$age"    // group by age
      }
    },
    {
      $count: "totalAgeGroups"
    }
  ])
< {
    totalAgeGroups: 7
  }
```

3. Match :

- `$match` is an **aggregation stage** that **filters documents** (like `WHERE` in SQL).
- It passes only the documents that match the given condition(s) to the next stage in the pipeline.
- It uses the **same query syntax** as `find()`.

Syntax:

```
db.collection.aggregate([
  {
    $match: { condition }
  }
])
```

Examples:

i. Match by field value:

Question: Write an aggregation query to find all students in the **CSE department**.

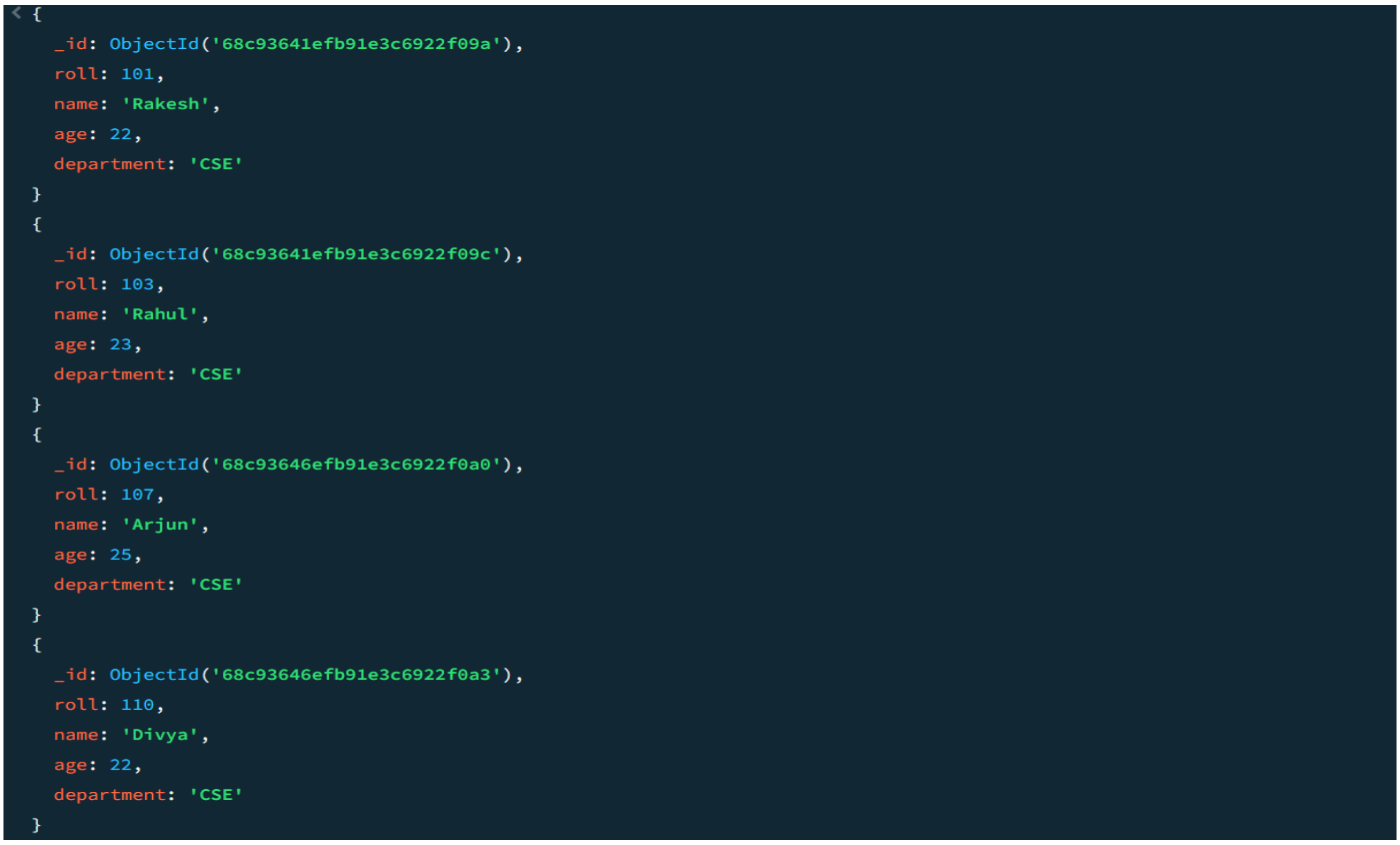
Query:

```
db.students.aggregate([
  { $match: { department: "CSE" } }

])
```

☞ **Filters documents where `department = "CSE"`.**

Output:


```
< {
    _id: ObjectId('68c93641efb91e3c6922f09a'),
    roll: 101,
    name: 'Rakesh',
    age: 22,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93641efb91e3c6922f09c'),
    roll: 103,
    name: 'Rahul',
    age: 23,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a0'),
    roll: 107,
    name: 'Arjun',
    age: 25,
    department: 'CSE'
  }
  {
    _id: ObjectId('68c93646efb91e3c6922f0a3'),
    roll: 110,
    name: 'Divya',
    age: 22,
    department: 'CSE'
  }
```

Created Student collection with 15 documents to UniversityDB: students.json file

```
db.students.insertMany ([
 {
   studentRollNumber: 101,
   name: "Ravi Kumar",
   department: "CSE",
   age: 20,
   mobileNumber: "9876543210",
   email: "ravi.kumar@example.com",
   subjects: { java: 85, c: 78, cpp: 80, python: 90, daa: 88 },
   totalMarks: 421
 },
 {
   studentRollNumber: 102,
   name: "Anjali Sharma",
   department: "ECE",
   age: 21,
   mobileNumber: "9876501234",
   email: "anjali.sharma@example.com",
   subjects: { java: 75, c: 82, cpp: 70, python: 88, daa: 79 },
   totalMarks: 394
 },
 {
   studentRollNumber: 103,
   name: "Kiran Reddy",
   department: "IT",
   age: 22,
   mobileNumber: "9123456789",
   email: "kiran.reddy@example.com",
   subjects: { java: 65, c: 70, cpp: 68, python: 72, daa: 75 },
   totalMarks: 350
 },
 {
   studentRollNumber: 104,
   name: "Meena Gupta",
   department: "CSE",
   age: 19,
   mobileNumber: "9876123456",
   email: "meena.gupta@example.com",
   subjects: { java: 95, c: 88, cpp: 92, python: 90, daa: 93 },
   totalMarks: 458
 },
 {
   studentRollNumber: 105,
   name: "Arjun Patel",
   department: "CSE",
   age: 20,
   mobileNumber: "9876549876",
   email: "arjun.patel@example.com",
   subjects: { java: 78, c: 82, cpp: 85, python: 80, daa: 84 },
   totalMarks: 409
 },
 {
   studentRollNumber: 106,
   name: "Sneha Iyer",
   department: "ECE",
   age: 21,
   mobileNumber: "9765432109",
   email: "sneha.iyer@example.com",
   subjects: { java: 88, c: 85, cpp: 90, python: 86, daa: 87 },
   totalMarks: 436
 },
 {
   studentRollNumber: 107,
   name: "Vikram Singh",
   department: "MECH",
   age: 23,
   mobileNumber: "9898989898",
   email: "vikram.singh@example.com",
   subjects: { java: 60, c: 65, cpp: 70, python: 68, daa: 72 },
   totalMarks: 335
 },
 {
   studentRollNumber: 108,
   name: "Pooja Nair",
   department: "IT",
   age: 20,
   mobileNumber: "9988776655",
   email: "pooja.nair@example.com",
   subjects: { java: 85, c: 89, cpp: 92, python: 94, daa: 90 },
   totalMarks: 450
 },
 {
   studentRollNumber: 109,
   name: "Rahul Verma",
   department: "CSE",
   age: 22,
   mobileNumber: "9123987654",
   email: "rahul.verma@example.com",
   subjects: { java: 70, c: 75, cpp: 80, python: 78, daa: 74 },
   totalMarks: 377
 },
 {
   studentRollNumber: 110,
   name: "Divya Menon",
   department: "ECE",
   age: 21,
   mobileNumber: "9876001122",
   email: "divya.menon@example.com",
   subjects: { java: 92, c: 90, cpp: 95, python: 93, daa: 91 },
   totalMarks: 461
 },
 {
   studentRollNumber: 111,
   name: "Amit Joshi",
   department: "IT",
   age: 22,
   mobileNumber: "9811223344",
   email: "amit.joshi@example.com",
   subjects: { java: 77, c: 73, cpp: 75, python: 80, daa: 78 },
   totalMarks: 383
 },
 {
   studentRollNumber: 112,
   name: "Neha Rani",
   department: "CSE",
   age: 20,
   mobileNumber: "9090909090",
   email: "neha.rani@example.com",
   subjects: { java: 89, c: 85, cpp: 87, python: 90, daa: 88 },
     totalMarks: 439
   },
   {
     studentRollNumber: 113,
     name: "Sandeep Yadav",
     department: "MECH",
     age: 23,
     mobileNumber: "9123001122",
     email: "sandeep.yadav@example.com",
     subjects: { java: 68, c: 65, cpp: 70, python: 72, daa: 69 },
     totalMarks: 344
   },
   {
     studentRollNumber: 114,
     name: "Kavya Rao",
     department: "IT",
     age: 21,
     mobileNumber: "9876554433",
     email: "kavya.rao@example.com",
     subjects: { java: 91, c: 87, cpp: 89, python: 92, daa: 90 },
     totalMarks: 449
   },
   {
     studentRollNumber: 115,
     name: "Manoj Das",
     department: "CSE",
     age: 22,
     mobileNumber: "9765123456",
     email: "manoj.das@example.com",
     subjects: { java: 82, c: 78, cpp: 80, python: 85, daa: 83 },
     totalMarks: 408
   }
 ])
```

ii. Match with comparison operator

Question: Find all students with **total marks greater than 400**.

Query:

```
db.students.aggregate([
  { $match: { totalMarks: { $gt: 400 } } }
])
```

☞ Filters only documents where `totalMarks > 400`.

Output:

```
[
  { studentRollNumber: 101, name: "Ravi Kumar", totalMarks: 421 },
  { studentRollNumber: 104, name: "Meena Gupta", totalMarks: 458 },
  { studentRollNumber: 105, name: "Arjun Patel", totalMarks: 409 },
  { studentRollNumber: 106, name: "Sneha Iyer", totalMarks: 436 },
  { studentRollNumber: 108, name: "Pooja Nair", totalMarks: 450 },
  { studentRollNumber: 110, name: "Divya Menon", totalMarks: 461 },
  { studentRollNumber: 112, name: "Neha Rani", totalMarks: 439 },
  { studentRollNumber: 114, name: "Kavya Rao", totalMarks: 449 },
  { studentRollNumber: 115, name: "Manoj Das", totalMarks: 408 }
]
```

Question: Find all students with **Java marks greater than 80**.

Query:

```
db.students.aggregate([
  { $match: { "subjects.java": { $gt: 80 } } }
])
```

Output (students with Java > 80):

```
[
  { studentRollNumber: 101, name: "Ravi Kumar", subjects: { java: 85, ... } },
  { studentRollNumber: 104, name: "Meena Gupta", subjects: { java: 95, ... } },
  { studentRollNumber: 106, name: "Sneha Iyer", subjects: { java: 88, ... } },
  { studentRollNumber: 108, name: "Pooja Nair", subjects: { java: 85, ... } },
  { studentRollNumber: 110, name: "Divya Menon", subjects: { java: 92, ... } },
  { studentRollNumber: 112, name: "Neha Rani", subjects: { java: 89, ... } },
….]
```

iii. Match with multiple conditions (AND)

Question: Find students in the **CSE department** with **totalMarks above 75**.

Query:

```
db.students.aggregate([
  { $match: { department: "CSE", totalMarks: { $gt: 75 } } }
])
```

Explanation:

- Filters students where `department = "CSE"` AND `totalMarks > 75`.
- Since all our CSE students have totalMarks above 75, all will appear.

Output:

```
[
  { studentRollNumber: 101, name: "Ravi Kumar", totalMarks: 421 },
  { studentRollNumber: 104, name: "Meena Gupta", totalMarks: 458 },
  { studentRollNumber: 105, name: "Arjun Patel", totalMarks: 409 },
  { studentRollNumber: 109, name: "Rahul Verma", totalMarks: 377 },
  { studentRollNumber: 112, name: "Neha Rani", totalMarks: 439 },
  { studentRollNumber: 115, name: "Manoj Das", totalMarks: 408 }
]
```

iv. Match with OR condition

Question: Find students who are either in **CSE** or **ECE**.

Query:

```
db.students.aggregate([
  { $match: { $or: [ { department: "CSE" }, { department: "ECE" } ] } }
])
```

Explanation:

- `$or` includes students belonging to either CSE OR ECE.

Output:

```
[
  { studentRollNumber: 101, name: "Ravi Kumar", department: "CSE" },
  { studentRollNumber: 104, name: "Meena Gupta", department: "CSE" },
  { studentRollNumber: 105, name: "Arjun Patel", department: "CSE" },
  { studentRollNumber: 109, name: "Rahul Verma", department: "CSE" },
  { studentRollNumber: 112, name: "Neha Rani", department: "CSE" },
  { studentRollNumber: 115, name: "Manoj Das", department: "CSE" },
  { studentRollNumber: 102, name: "Anjali Sharma", department: "ECE" },
  { studentRollNumber: 106, name: "Sneha Iyer", department: "ECE" },
  { studentRollNumber: 110, name: "Divya Menon", department: "ECE" }
]
```

✅ **9 students total** (6 from CSE + 3 from ECE).

v. Using $match before $group

Question: Group students by age, but only include students with **totalMarks above 70**.

Query:

```
db.students.aggregate([
  { $match: { totalMarks: { $gt: 70 } } },   // filter first
  { $group: { _id: "$age", total: { $sum: 1 } } }
])
```

Explanation:

- All students in our dataset have `totalMarks > 70`.
- So effectively, it groups all students by their `age`.

Output:

```
[
  { _id: 19, total: 1 },   // Meena Gupta
  { _id: 20, total: 4 },   // Ravi, Arjun, Pooja, Neha
  { _id: 21, total: 4 },   // Anjali, Sneha, Divya, Kavya
  { _id: 22, total: 4 },   // Kiran, Rahul, Amit, Manoj
  { _id: 23, total: 2 }    // Vikram, Sandeep
]
```

✅ So we grouped **15 students into 5 different age groups**.

4. Sort: $sort is a **pipeline stage** used to sort documents. And It takes a **sort key** and a direction:
1 → ascending order and -1 → descending order

Syntax:

```
{ $sort: { fieldName: 1 } }   // ascending
{ $sort: { fieldName: -1 } }  // descending
```

Examples on the students collection:

i. **Sort students by total marks in descending order**

Question: Find all students sorted by their **totalMarks** in **descending order**.

Query:

```
db.students.aggregate([
  { $sort: { totalMarks: -1 } }
])
```

Output (top 3 shown for brevity):

```
[
  { "name": "Sneha", "totalMarks": 448, "age": 26, "department": "CSE" },
  { "name": "Ramesh", "totalMarks": 433, "age": 24, "department": "CSE" },
  { "name": "Priya",  "totalMarks": 427, "age": 22, "department": "CSE" },
  ...
]
```

ii. **Sort students by age in ascending order**

Question: List all students sorted by **age** from youngest to oldest.

Query:

```
db.students.aggregate([
  { $sort: { age: 1 } }
])
```

Output (top 3 shown):

```
[
  { "name": "Priya", "age": 22, "totalMarks": 427 },
  { "name": "Vikram", "age": 22, "totalMarks": 402 },
  { "name": "Deepa", "age": 23, "totalMarks": 408 },
  ... ]
```

iii. **Sort by multiple fields (age asc, then totalMarks desc)**

Question: Sort students by **age** (ascending). If two students have the same age, sort them by **totalMarks** (descending).

Query:

```
db.students.aggregate([
  { $sort: { age: 1, totalMarks: -1 } }
])
```

Output (sample):

```
[
  { "name": "Priya", "age": 22, "totalMarks": 427 },
  { "name": "Vikram", "age": 22, "totalMarks": 402 },
  { "name": "Deepa", "age": 23, "totalMarks": 408 },
  { "name": "Meena", "age": 23, "totalMarks": 389 },
  ...
]
```

iv. **Using $match and $sort**

Question: Find all students whose **age is greater than 23**, and display them in **ascending order of age**.

Query:

```
db.students.aggregate([
  { $match: { age: { $gt: 23 } } },  // filter students with age > 23
  { $sort: { age: 1 } }              // sort ascending by age
])
```

Output:

```
[
  {
    "studentRollNumber": 101,
    "name": "Ramesh",
    "department": "CSE",
    "age": 24,
    "mobileNumber": "9876543210",
    "email": "ramesh@example.com",
    "subjects": { "java": 85, "c": 78, "cpp": 90, "python": 88, "DAA": 92 },
    "totalMarks": 433
  },
  {
    "studentRollNumber": 106,
    "name": "Anitha",
    "department": "CSE",
    "age": 24,
    "mobileNumber": "8765432109",
    "email": "anitha@example.com",
    "subjects": { "java": 81, "c": 73, "cpp": 88, "python": 79, "DAA": 92 },
    "totalMarks": 413
  },
  {
    "studentRollNumber": 102,
    "name": "Suresh",
    "department": "ECE",
    "age": 25,
    "mobileNumber": "9876501234",
    "email": "suresh@example.com",
    "subjects": { "java": 75, "c": 80, "cpp": 85, "python": 70, "DAA": 88 },
    "totalMarks": 398
  },
  {
    "studentRollNumber": 110,
    "name": "Sneha",
    "department": "CSE",
    "age": 26,
    "mobileNumber": "8765498765",
    "email": "sneha@example.com",
    "subjects": { "java": 89, "c": 85, "cpp": 90, "python": 91, "DAA": 93 },
    "totalMarks": 448
  },
  {
    "studentRollNumber": 111,
    "name": "Akash",
    "department": "ECE",
    "age": 27,
    "mobileNumber": "7896543210",
    "email": "akash@example.com",
    "subjects": { "java": 76, "c": 70, "cpp": 79, "python": 74, "DAA": 80 },
    "totalMarks": 379
  }
]
```

5. Limit: **`$limit` is used to** restrict the number of documents **that pass through the pipeline. and Very useful when we want** top N results**.**

Syntax: { $limit: <number> }

Here `<number>` is the maximum number of documents to return.

Examples on students collection

i. Get first 5 students from the collection

Question: Show only the first 5 students.

Query:

```
        db.students.aggregate([
          { $limit: 5 }
        ])
```

Output (first 5 docs in the collection order):

```
[
  { "name": "Ramesh", "age": 24, "department": "CSE", "totalMarks": 433 },
  { "name": "Suresh", "age": 25, "department": "ECE", "totalMarks": 398 },
  { "name": "Priya", "age": 22, "department": "CSE", "totalMarks": 427 },
  { "name": "Vikram", "age": 22, "department": "EEE", "totalMarks": 402 },
  { "name": "Deepa", "age": 23, "department": "IT", "totalMarks": 408 }
]
```

ii. Get Top 3 students **by totalMarks**

Question: Find the top 3 students who scored the highest marks.

Query:

```
         db.students.aggregate([
           { $sort: { totalMarks: -1 } },   // highest marks first
           { $limit: 3 }                    // pick top 3
])
```

Output:

```
[
  { "name": "Sneha", "age": 26, "department": "CSE", "totalMarks": 448 },
  { "name": "Ramesh", "age": 24, "department": "CSE", "totalMarks": 433 },
  { "name": "Priya", "age": 22, "department": "CSE", "totalMarks": 427 }
]
```

iii. Combine $match + $sort + $limit

Question: Find the **top 2 scorers** in the **CSE department**.

Query:

```
db.students.aggregate([
  { $match: { department: "CSE" } },   // only CSE
  { $sort: { totalMarks: -1 } },       // sort by marks desc
  { $limit: 2 }                        // take top 2
])
```

Output:

```
[
  { "name": "Sneha", "age": 26, "department": "CSE", "totalMarks": 448 },
  { "name": "Ramesh", "age": 24, "department": "CSE", "totalMarks": 433 }
]
```

Explanation:

- `$limit` stops the pipeline after reaching the specified number of documents.
- Often used after `$sort` to get **top N results**.
- Can be combined with `$match` for filtered results.

6. Project: `$project` is used to **choose which fields** to include, exclude, or add new computed fields in the result. It reshapes documents. And Works like SQL's **SELECT column list**

Syntax:

```
{
  $project: {
    field1: 1,    // include
    field2: 1,    // include
    field3: 0,    // exclude
    newField: <expression>   // computed field
  }
}
```

☞ `1` → include field

☞ `0` → exclude field

☞ You can also create new fields using expressions (like `$sum`, `$concat`, `$toUpper`, etc.).

Examples on students collection

i. Include only name and department

Question: Show only the student **name** and **department**.

Query:

```
db.students.aggregate([
  { $project: { name: 1, department: 1 } }
])
```

Output:

```
[
  { "name": "Ramesh", "department": "CSE" },
  { "name": "Suresh", "department": "ECE" },
  { "name": "Priya", "department": "CSE" },
  ...
]
```

ii. Exclude email and mobile number

Question: Show all fields **except email and mobileNumber**.

Query:

```
db.students.aggregate([
  { $project: { email: 0, mobileNumber: 0 } }
])
```

Output (sample):

```
[
  { "name": "Ramesh", "age": 24, "department": "CSE", "totalMarks": 433 },
  { "name": "Suresh", "age": 25, "department": "ECE", "totalMarks": 398 },
  ...
]
```

iii. Add a computed field – percentage

Question: Show student name, department, totalMarks, and add a new field `percentage` (out of 500).

Query:

```
db.students.aggregate([
  {
    $project: {
      name: 1,
      department: 1,
      totalMarks: 1,
      percentage: { $multiply: [ { $divide: ["$totalMarks", 500] }, 100 ] }
    }
  }
])
```

Output (sample):

```
[
  { "name": "Ramesh", "department": "CSE", "totalMarks": 433, "percentage": 86.6 },
  { "name": "Suresh", "department": "ECE", "totalMarks": 398, "percentage": 79.6 },
  { "name": "Priya",  "department": "CSE", "totalMarks": 427, "percentage": 85.4 },
  ...
]
```

iv. Rename a field

Question: Show student **name** but rename it as **studentName**.

Query:

```
db.students.aggregate([
  { $project: { studentName: "$name", department: 1, totalMarks: 1 } }
])
```

Output (sample):

```
[
  { "studentName": "Ramesh", "department": "CSE", "totalMarks": 433 },
  { "studentName": "Suresh", "department": "ECE", "totalMarks": 398 },
  ...
]
```

✅ **Explanation:**

- `$project` lets you **control the shape** of documents in the pipeline.
- You can:
  - Select specific fields (`1` include / `0` exclude)
  - Compute new fields (percentage, rank, etc.)
  - Rename fields for better readability
  -

Question: Find the top 3 students based on total marks. Show only their name, department, and percentage.

Query:

```
db.students.aggregate([
  { $sort: { totalMarks: -1 } },          // Step 1: sort by marks (highest first)
  { $limit: 3 },                          // Step 2: keep only top 3
  {
    $project: {                           // Step 3: reshape output
      name: 1,
      department: 1,
    percentage: {
      $multiply: [ { $divide: ["$totalMarks", 500] }, 100 ]
    }
   }
 }])
```

Output (sample):

```
[
 { "name": "Sneha", "department": "CSE", "percentage": 89.6 },
 { "name": "Ramesh", "department": "CSE", "percentage": 86.6 },
 { "name": "Priya",  "department": "CSE", "percentage": 85.4 }
]
```

Explanation of pipeline:

1. **`$sort`** → Sort students by totalMarks descending.
2. **`$limit`** → Restrict result to top 3 students.
3. **`$project`** → Show only `name`, `department`, and compute `percentage`.

7. addFields

- `$addFields` **adds new fields** or **updates existing fields** in the documents.
- Unlike `$project`, it **does not remove other fields** by default; it keeps all existing fields.
- Useful for **computed fields**, **renaming fields**, or **modifying data** in the pipeline.

Syntax:

```
{
  $addFields: {
    newField1: <expression>,  // create a new field
    existingField: <expression>  // modify an existing field
  }
}
```

Examples on students Collection

i. Add a new field percentage

Question: **Add a `percentage` field to each student (totalMarks out of 500).**

Query:

```
      db.students.aggregate([
        {
          $addFields: {
            percentage: { $multiply: [ { $divide: ["$totalMarks", 500] }, 100 ] }
          }
        }
])
```

Output (sample):

```
[
  { "name": "Ramesh", "totalMarks": 433, "percentage": 86.6, "department": "CSE", ...
},
  { "name": "Suresh", "totalMarks": 398, "percentage": 79.6, "department": "ECE", ...
},
  { "name": "Priya",  "totalMarks": 427, "percentage": 85.4, "department": "CSE", ... }
]
```

✓ Adds a new computed field `percentage` while keeping all other fields.

ii. Modify an existing field

Question: Increase `totalMarks` by 10 for all students.

Query:
```
db.students.aggregate([
  {
    $addFields: {
      totalMarks: { $add: ["$totalMarks", 10] }
    }
  }
])
```

Output (sample):
```
[
  { "name": "Ramesh", "totalMarks": 443, "department": "CSE", ... },
  { "name": "Suresh", "totalMarks": 408, "department": "ECE", ... },
  ...
]
```
✓ Updates the `totalMarks` field directly without removing other fields.

iii. Add multiple fields at once

Question: Add `percentage` and a new field `passed` (true if totalMarks >= 200).

Query:
```
db.students.aggregate([
  {
    $addFields: {
      percentage: { $multiply: [ { $divide: ["$totalMarks", 500] }, 100 ] },
      passed: { $gte: ["$totalMarks", 200] }
    }
  }
])
```
Output (sample):
```
[
  { "name": "Ramesh", "totalMarks": 433, "percentage": 86.6, "passed": true, ... },
  { "name": "Suresh", "totalMarks": 398, "percentage": 79.6, "passed": true, ... },
  ...
]
```
✓ Adds multiple fields in one stage and keeps all existing fields.

Difference Between `$addFields` and `$project`

Feature	`$project`	`$addFields`
Keeps existing fields	No (only selected fields)	Yes (all fields are kept)
Adds new fields	Yes	Yes
Modifies fields	Yes (must include in project)	Yes
Removes fields by default	Yes (not included fields are removed)	No

Question: Find the **top 3 students** based on **totalMarks**, add a new field percentage (out of 500), and show all existing fields along with the new field.

Query:

db.students.aggregate([

```
        // Step 1: Add a computed field 'percentage'
        {
          $addFields: {
            percentage: { $multiply: [ { $divide: ["$totalMarks", 500] }, 100 ] }
          }
        },
        // Step 2: Sort by totalMarks descending
        { $sort: { totalMarks: -1 } },

        // Step 3: Limit to top 3 students

        { $limit: 3 }

      ])
```

Output (sample):

```
 [
   {
     "studentRollNumber": 110,
     "name": "Sneha",
     "department": "CSE",
     "age": 26,
     "mobileNumber": "8765498765",
     "email": "sneha@example.com",
     "subjects": { "java": 89, "c": 85, "cpp": 90, "python": 91, "DAA": 93 },
     "totalMarks": 448,
     "percentage": 89.6
   },
   {
     "studentRollNumber": 101,
     "name": "Ramesh",
     "department": "CSE",
     "age": 24,
     "mobileNumber": "9876543210",
     "email": "ramesh@example.com",
     "subjects": { "java": 85, "c": 78, "cpp": 90, "python": 88, "DAA": 92 },
     "totalMarks": 433,
     "percentage": 86.6
   },
   {
     "studentRollNumber": 103,
     "name": "Priya",
     "department": "CSE",
     "age": 22,
     "mobileNumber": "9876512340",
     "email": "priya@example.com",
     "subjects": { "java": 81, "c": 79, "cpp": 88, "python": 85, "DAA": 94 },
     "totalMarks": 427,
     "percentage": 85.4
   }
 ]
```

8. Lookup:

- `$lookup` is used to **join data from another collection** into the current collection.
- It's similar to SQL's `LEFT JOIN`.
- Allows combining documents from **two collections** based on a matching field.

Syntax (basic):

```
{
  $lookup: {
    from: "<otherCollection>",        // Collection to join
    localField: "<fieldInCurrent>",   // Field in current collection
    foreignField: "<fieldInOther>",   // Field in other collection
    as: "<newArrayField>"             // Output array field with joined documents
  }
}
```

- `from` → the collection you want to join
- `localField` → field in the current collection
- `foreignField` → field in the other collection
- `as` → new field (array) to hold joined documents

Example with Students and Departments

Suppose we have **two collections**:

1. `students`

```
[
  { "_id": 1, "name": "Ramesh", "departmentId": 101, "totalMarks": 433 },
  { "_id": 2, "name": "Suresh", "departmentId": 102, "totalMarks": 398 },
  { "_id": 3, "name": "Priya",  "departmentId": 101, "totalMarks": 427 }
]
```

2. `departments`

```
[
  { "deptId": 101, "deptName": "CSE" },
  { "deptId": 102, "deptName": "ECE" }
]
```

Question: Join `students` with `departments` to get student name and department name.

Query

```
      db.students.aggregate([
        {
          $lookup: {
            from: "departments",        // other collection
            localField: "departmentId", // field in students
            foreignField: "deptId",     // field in departments
            as: "departmentInfo"         // output array
          }
        }
      ])
```

Output

```
[
  {
    "_id": 1,
    "name": "Ramesh",
    "departmentId": 101,
    "totalMarks": 433,
    "departmentInfo": [
      { "deptId": 101, "deptName": "CSE" }
    ]
  },
  {
    "_id": 2,
    "name": "Suresh",
    "departmentId": 102,
    "totalMarks": 398,
    "departmentInfo": [
      { "deptId": 102, "deptName": "ECE" }
    ]
  },
  {
    "_id": 3,
    "name": "Priya",
    "departmentId": 101,
    "totalMarks": 427,
    "departmentInfo": [
      { "deptId": 101, "deptName": "CSE" }
    ]
  }
]
```

Explanation

1. `$lookup` joins `students` with `departments` using `departmentId` → `deptId`.
2. The matching document(s) from `departments` are stored as an **array** in `departmentInfo`.
3. If no match exists, the array will be empty.

Optional: Flatten the array with `$unwind`:

`Query`: If you want **direct fields instead of an array**:

```
      db.students.aggregate([
        {
          $lookup: {
            from: "departments",
            localField: "departmentId",
            foreignField: "deptId",
            as: "departmentInfo"
          }
        },
        { $unwind: "$departmentInfo" },
        { $project: { name: 1, totalMarks: 1, department: "$departmentInfo.deptName" }
      }
      ])
```

Output:

```
[
  { "name": "Ramesh", "totalMarks": 433, "department": "CSE" },
  { "name": "Suresh", "totalMarks": 398, "department": "ECE" },
  { "name": "Priya",  "totalMarks": 427, "department": "CSE" }
]
```

Now you get a **flat structure** instead of an array.

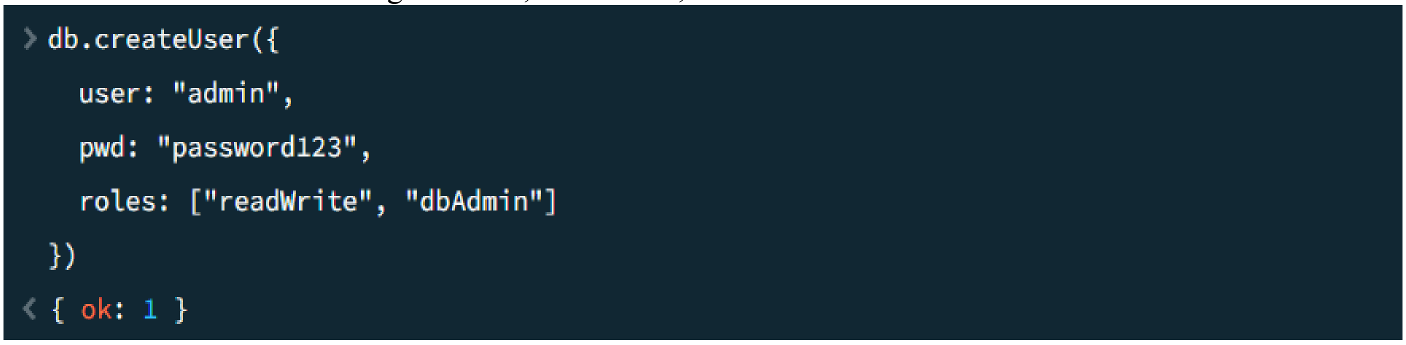
J. Admin Commands:

Create database user:

```
db.createUser({
  user: "admin",
  pwd: "password123",
  roles: ["readWrite", "dbAdmin"]
})
```

Creates a new DB user named `admin` with password `password123`.

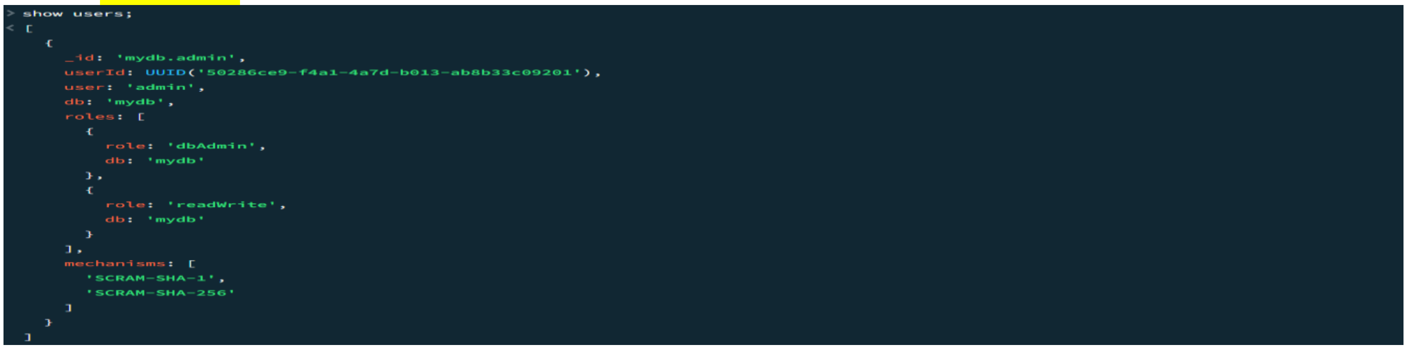
- **readWrite** → Can read & write data.
- **dbAdmin** → Can manage indexes, collections, etc.



```
> db.createUser({
    user: "admin",
    pwd: "password123",
    roles: ["readWrite", "dbAdmin"]
  })
< { ok: 1 }
```

Show users: Shows all database users (for current DB).

Command: show users



```
> show users;
< [
    {
      _id: 'mydb.admin',
      userId: UUID('50286ce9-f4a1-4a7d-b013-ab8b33c09201'),
      user: 'admin',
      db: 'mydb',
      roles: [
        {
          role: 'dbAdmin',
          db: 'mydb'
        },
        {
          role: 'readWrite',
          db: 'mydb'
        }
      ],
      mechanisms: [
        'SCRAM-SHA-1',
        'SCRAM-SHA-256'
      ]
    }
  ]
```

summary:

- `db.users` → refers to your **data collection** named `users`.
- `show users` → shows the **database users (accounts)** that have access to the DB.

Integration with Express.js (Node.js)

MongoDB is often used with **Node.js** + **Express** to build APIs.

☞ Example: Simple Express API with Mongoose ODM

```
const express = require("express");
const mongoose = require("mongoose");

const app = express();
app.use(express.json());

// Connect to DB
mongoose.connect("mongodb://localhost:27017/mydb");

// Define User Schema
const User = mongoose.model("User", new mongoose.Schema({
                                                          name: String,
                                                          age: Number
                                                        }));
```

```
// CRUD APIs

// CREATE
app.post("/users", async (req, res) => {
  const user = new User(req.body);
  await user.save();
  res.send(user);
});

// READ
app.get("/users", async (req, res) => {
  const users = await User.find();
  res.json(users);
});


// UPDATE
app.put("/users/:id", async (req, res) => {
  const user = await User.findByIdAndUpdate(req.params.id, req.body, { new: true });
  res.send(user);
});

// DELETE
app.delete("/users/:id", async (req, res) => {
  await User.findByIdAndDelete(req.params.id);
  res.send({ message: "User deleted" });
});

app.listen(3000, () => console.log("Server running on http://localhost:3000"));
```

Let me show you what the **outputs will look like** when you call each endpoint.

1. CREATE (POST /users)

☞ Request (via Postman / curl):

```
POST http://localhost:3000/users
Content-Type: application/json

{
  "name": "Rakesh",
  "age": 25
}
```

☞ Output (Response):

```
{
  "_id": "650faeab43e9f23c8d9f1234",   // auto-generated by MongoDB
  "name": "Rakesh",
  "age": 25,
  "__v": 0
}
```

2. READ (GET /users)

☞ Request:

```
GET http://localhost:3000/users
```

☞ Output (Response):

```
[
  {
    "_id": "650faeab43e9f23c8d9f1234",
    "name": "Rakesh",
    "age": 25,
    "__v": 0
  },
  {
    "_id": "650faf0043e9f23c8d9f5678",
    "name": "Anita",
    "age": 22,
    "__v": 0
  }
]
```

(You will see all users stored in the DB.)

3. UPDATE (PUT /users/:id)

☞ Request:

```
PUT http://localhost:3000/users/650faeab43e9f23c8d9f1234
Content-Type: application/json

{
  "age": 26
}
```

☞ Output (Response):

```
{
  "_id": "650faeab43e9f23c8d9f1234",
  "name": "Rakesh",
  "age": 26,
  "__v": 0
}
```

(The `age` field is updated.)

4. DELETE (DELETE /users/:id)

☞ Request:

```
DELETE http://localhost:3000/users/650faeab43e9f23c8d9f1234
```

☞ Output (Response):

```
{
  "message": "User deleted"
}
```

✓ So basically:

- **POST** → creates a user and returns the new user object.
- **GET** → returns all users in an array.
- **PUT** → updates a user and returns the updated document.
- **DELETE** → removes the user and returns a confirmation message.

Importing and Exporting MongoDB Documents

- ➢ Importing and exporting in MongoDB allows you to **move data between the database and external files**.
- ➢ Exporting saves data from a collection into **JSON or CSV files**, useful for **backup, sharing, or migration**.
- ➢ Importing loads data from **JSON or CSV files** into a collection, helping with **restoring backups or populating new databases**.
- ➢ mongoexport is used for exporting, and mongoimport is used for importing.

➢ You can choose specific fields, formats, and control whether the first CSV row is treated as headers. These utilities make it easy to manage, transfer, and maintain data across different environments.

1. Exporting Documents (Backup / Sharing)

- Exporting means **taking data from a MongoDB collection and saving it into an external file** (like .json or .csv).
- This is useful when you want to:
  - Backup your database.
  - Share data with others.
  - Move data between environments (e.g., dev → prod).
- MongoDB provides a utility called **mongoexport**.
- Exported files can later be imported back using `mongoimport`.
- This ensures data can be **safely transferred or restored** when needed.

Syntax: ***mongoexport --db <database_name> --collection <collection_name> --out <filename.json>***

Example: Export all documents from students collection in school database:

Command (JSON Export): mongoexport --db school --collection students --out students.json

☞ Exports data into a JSON file named students.json.

Sample Output (students.json):

```
{ "_id": { "$oid": "651f8d0a1a2c3b4d5e6f7890" }, "name": "Ravi", "age": 21, "grade": "A" }

{ "_id": { "$oid": "651f8d0a1a2c3b4d5e6f7891" }, "name": "Anita", "age": 23, "grade": "B" }

{ "_id": { "$oid": "651f8d0a1a2c3b4d5e6f7892" }, "name": "Vikram", "age": 20, "grade": "A" }
```

2. Importing Documents (Restore / Load Data)

- Importing is the process of **loading external JSON or CSV data into a MongoDB collection**.
- It is commonly used to **restore backups or populate new collections**.
- `mongoimport` reads the data from files and inserts it into the specified collection.
- You can import **JSON arrays or CSV files**, and specify field names.
- This helps in **migrating data or sharing datasets** across databases.
- It ensures the data in your collection matches the imported external data.
- MongoDB provides a utility called **mongoimport**.

Syntax:
mongoimport --db <database_name> --collection <collection_name> --file <filename.json> --jsonArray

Example: Import data into students collection in school database:

Command (JSON Import):
mongoimport --db school --collection students --file students.json --jsonArray

☞ Imports all documents from students.json into MongoDB.

Effect in MongoDB Collection:

```
{ "_id": ObjectId("651f8d0a1a2c3b4d5e6f7890"), "name": "Ravi", "age": 21, "grade": "A" }

{ "_id": ObjectId("651f8d0a1a2c3b4d5e6f7891"), "name": "Anita", "age": 23, "grade": "B" }

{ "_id": ObjectId("651f8d0a1a2c3b4d5e6f7892"), "name": "Vikram", "age": 20, "grade": "A" }
```

3. Exporting in CSV Format

- You can export MongoDB collections to **CSV files** to focus on specific fields.
- This is useful for **reports, Excel analysis, or sharing only relevant data**.
- mongoexport allows selection of fields and file type (--type=csv).
- CSV export helps **simplify large datasets** for easier handling in other tools.
- You can later import CSV files back into MongoDB using mongoimport.
- It provides a simple way to **interact with non-JSON systems**.

Command (CSV Export):

```
mongoexport --db school --collection students --type=csv --fields name,age,grade --out students.csv
```

☞ Exports only name, age, grade fields into a CSV file.

Sample Output (students.csv):

```
name,age,grade

Ravi,21,A

Anita,23,B
Vikram,20,A
```

4. Importing from CSV

- MongoDB allows importing **CSV files into a collection** using mongoimport.
- The --headerline option uses the **first row as field names**, mapping CSV columns to document fields.
- It is useful for **loading structured tabular data** from Excel or other sources.
- You can populate new collections or update existing ones with CSV data.
- This feature simplifies **integration of MongoDB with other data tools**.
- It is ideal for **data migration, initial project setup, or bulk imports**.

Command (CSV Import):

```
mongoimport --db school --collection students --type=csv --headerline --file students.csv
```

☞ --headerline tells MongoDB to use the **first row of the CSV file as field names**.

Effect in MongoDB Collection:

```
{ "_id": ObjectId("6520abc1234def5678901234"), "name": "Ravi", "age": 21, "grade": "A" }

{ "_id": ObjectId("6520abc1234def5678901235"), "name": "Anita", "age": 23, "grade": "B" }

{ "_id": ObjectId("6520abc1234def5678901236"), "name": "Vikram", "age": 20, "grade": "A" }
```

Summary Table

Operation	Tool	Example Command
Export JSON	mongoexport	mongoexport --db school --collection students --out students.json
Import JSON	mongoimport	mongoimport --db school --collection students --file students.json --jsonArray
Export CSV	mongoexport	mongoexport --db school --collection students --type=csv --fields name,age,grade --out students.csv
Import CSV	mongoimport	mongoimport --db school --collection students --type=csv --headerline --file students.csv

SAMPLE QUERIES: **collection: students.json**

```
[{
  "_id": {
    "$oid": "68d25c2688d2e395158ea17f"
  },
  "studentRollNumber": 101,
  "name": "Ravi Kumar",
  "department": "CSE",
  "age": 20,
  "mobileNumber": "9876543210",
  "email": "ravi.kumar@example.com",
  "subjects": {
    "java": 85,
    "c": 78,
    "cpp": 80,
    "python": 90,
    "daa": 88
  },
  "totalMarks": 421
},
{
  "_id": {
    "$oid": "68d25c2688d2e395158ea180"
  },
  "studentRollNumber": 102,
  "name": "Anjali Sharma",
  "department": "ECE",
  "age": 21,
  "mobileNumber": "9876501234",
  "email": "anjali.sharma@example.com",
  "subjects": {
    "java": 75,
    "c": 82,
    "cpp": 70,
    "python": 88,
    "daa": 79
  },
  "totalMarks": 394
},
{
  "_id": {
    "$oid": "68d25c2688d2e395158ea181"
  },
  "studentRollNumber": 103,
  "name": "Kiran Reddy",
  "department": "IT",
  "age": 22,
  "mobileNumber": "9123456789",
  "email": "kiran.reddy@example.com",
  "subjects": {
    "java": 65,
    "c": 70,
    "cpp": 68,
    "python": 72,
    "daa": 75
  },
  "totalMarks": 350
},
{
  "_id": {
    "$oid": "68d25c2688d2e395158ea182"
  },
  "studentRollNumber": 104,
  "name": "Meena Gupta",
  "department": "CSE",
  "age": 19,
  "mobileNumber": "9876123456",
  "email": "meena.gupta@example.com",
  "subjects": {
    "java": 95,
    "c": 88,
    "cpp": 92,
    "python": 90,
    "daa": 93
  },
  "totalMarks": 458
},
{
  "_id": {
    "$oid": "68d25c2688d2e395158ea183"
  },
  "studentRollNumber": 105,
  "name": "Arjun Patel",
  "department": "CSE",
  "age": 20,
  "mobileNumber": "9876549876",
  "email": "arjun.patel@example.com",
  "subjects": {
    "java": 78,
    "c": 82,
    "cpp": 85,
    "python": 80,
    "daa": 84
  },
  "totalMarks": 409
},
{
  "_id": {
    "$oid": "68d25c2688d2e395158ea184"
  },
  "studentRollNumber": 106,
  "name": "Sneha Iyer",
  "department": "ECE",
  "age": 21,
  "mobileNumber": "9765432109",
  "email": "sneha.iyer@example.com",
  "subjects": {
    "java": 88,
    "c": 85,
    "cpp": 90,
    "python": 86,
    "daa": 87
  },
  "totalMarks": 436
},
{
  "_id": {
    "$oid": "68d25c2688d2e395158ea185"
  },
  "studentRollNumber": 107,
  "name": "Vikram Singh",
  "department": "MECH",
  "age": 23,
  "mobileNumber": "9898989898",
  "email": "vikram.singh@example.com",
  "subjects": {
    "java": 60,
    "c": 65,
    "cpp": 70,
    "python": 68,
    "daa": 72
  },
  "totalMarks": 335
 },
 {
  "_id": {
    "$oid": "68d25c2688d2e395158ea186"
  },
  "studentRollNumber": 108,
  "name": "Pooja Nair",
  "department": "IT",
  "age": 20,
  "mobileNumber": "9988776655",
  "email": "pooja.nair@example.com",
  "subjects": {
    "java": 85,
    "c": 89,
    "cpp": 92,
    "python": 94,
    "daa": 90
  },
  "totalMarks": 450
 },
 {
  "_id": {
    "$oid": "68d25c2688d2e395158ea187"
  },
  "studentRollNumber": 109,
  "name": "Rahul Verma",
  "department": "CSE",
  "age": 22,
  "mobileNumber": "9123987654",
  "email": "rahul.verma@example.com",
  "subjects": {
    "java": 70,
    "c": 75,
    "cpp": 80,
    "python": 78,
    "daa": 74
  },
  "totalMarks": 377
 },
 {
  "_id": {
    "$oid": "68d25c2688d2e395158ea188"
  },
  "studentRollNumber": 110,
  "name": "Divya Menon",
  "department": "ECE",
  "age": 21,
  "mobileNumber": "9876001122",
  "email": "divya.menon@example.com",
  "subjects": {
    "java": 92,
    "c": 90,
    "cpp": 95,
    "python": 93,
    "daa": 91
  },
  "totalMarks": 461
 },
 {
  "_id": {
    "$oid": "68d25c2688d2e395158ea189"
  },
  "studentRollNumber": 111,
  "name": "Amit Joshi",
  "department": "IT",
  "age": 22,
  "mobileNumber": "9811223344",
  "email": "amit.joshi@example.com",
  "subjects": {
    "java": 77,
    "c": 73,
    "cpp": 75,
    "python": 80,
    "daa": 78
  },
  "totalMarks": 383
 },
 {
  "_id": {
    "$oid": "68d25c2688d2e395158ea18a"
  },
  "studentRollNumber": 112,
  "name": "Neha Rani",
  "department": "CSE",
  "age": 20,
  "mobileNumber": "9090909090",
  "email": "neha.rani@example.com",
  "subjects": {
    "java": 89,
    "c": 85,
    "cpp": 87,
    "python": 90,
    "daa": 88
  },
  "totalMarks": 439
 },
 {
  "_id": {
    "$oid": "68d25c2688d2e395158ea18b"
  },
 "studentRollNumber": 113,
 "name": "Sandeep Yadav",
 "department": "MECH",
 "age": 23,
 "mobileNumber": "9123001122",
 "email": "sandeep.yadav@example.com",
 "subjects": {
  "java": 68,
  "c": 65,
  "cpp": 70,
  "python": 72,
  "daa": 69
 },
 "totalMarks": 344
},
{
 "_id": {
  "$oid": "68d25c2688d2e395158ea18c"
 },
 "studentRollNumber": 114,
 "name": "Kavya Rao",
 "department": "IT",
 "age": 21,
 "mobileNumber": "9876554433",
 "email": "kavya.rao@example.com",
 "subjects": {
  "java": 91,
  "c": 87,
  "cpp": 89,
  "python": 92,
  "daa": 90
 },
 "totalMarks": 449
},
{
 "_id": {
  "$oid": "68d25c2688d2e395158ea18d"
 },
 "studentRollNumber": 115,
 "name": "Manoj Das",
 "department": "CSE",
 "age": 22,
 "mobileNumber": "9765123456",
 "email": "manoj.das@example.com",
 "subjects": {
  "java": 82,
  "c": 78,
  "cpp": 80,
  "python": 85,
  "daa": 83
 },
 "totalMarks": 408
```

```
}]
```

Q1. Write a query to Find all students in the CSE department.

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

Output (names only for readability):

```
---------------------------------------
[ "Ravi Kumar", "Meena Gupta", "Arjun Patel", "Rahul Verma", "Neha Rani", "Manoj Das" ]
```

Q2.Write a query to Show only name and totalMarks of students whose age is greater than 21.

```
  printjson(db.students.find({ age: { $gt: 21 } }, { name: 1, totalMarks: 1, _id: 0 }))
```

✅Output:

```
-------
[
  { name: "Kiran Reddy", totalMarks: 350 },
  { name: "Vikram Singh", totalMarks: 335 },
  { name: "Rahul Verma", totalMarks: 377 },
  { name: "Amit Joshi", totalMarks: 383 },
  { name: "Sandeep Yadav", totalMarks: 344 },
  { name: "Manoj Das", totalMarks: 408 }
]
```

Q3.Write a query to Find students whose totalMarks are between 400 and 450.

```
    printjson(db.students.find({ totalMarks: { $gte: 400, $lte: 450 } }))
```

✅**Output (names only):**

```
------------------------------
[ "Ravi Kumar", "Arjun Patel", "Sneha Iyer", "Pooja Nair", "Neha Rani", "Kavya Rao", "Manoj Das" ]
```

Q4.Write a query to Find students with java marks greater than 85.

```
printjson(db.students.find({ "subjects.java": { $gt: 85 } }))
```

✅**Output (names only):**

```
[ "Meena Gupta", "Sneha Iyer", "Pooja Nair", "Divya Menon", "Neha Rani", "Kavya Rao" ]
```

Q5.Write a query to Display name, department, and python marks of all IT students.

```
printjson(db.students.find({ department: "IT" }, { name: 1, department: 1, "subjects.python": 1, _id: 0 }))
```

✅**Output:**

```
[
  { name: "Kiran Reddy", department: "IT", subjects: { python: 72 } },
  { name: "Pooja Nair", department: "IT", subjects: { python: 94 } },
  { name: "Amit Joshi", department: "IT", subjects: { python: 80 } },
  { name: "Kavya Rao", department: "IT", subjects: { python: 92 } }
]
```

Q6.Write a query to Find top 3 students with the highest totalMarks.

```
printjson(db.students.find().sort({ totalMarks: -1 }).limit(3))
```

✅**Output:**

```
[
  { name: "Divya Menon", totalMarks: 461 },
  { name: "Meena Gupta", totalMarks: 458 },
  { name: "Pooja Nair", totalMarks: 450 }
]
```

Q7.Write a query to List bottom 5 students based on totalMarks.

```
printjson(db.students.find().sort({ totalMarks: 1 }).limit(5))
```

✅**Output:**

```
[
  { name: "Vikram Singh", totalMarks: 335 },
  { name: "Sandeep Yadav", totalMarks: 344 },
  { name: "Kiran Reddy", totalMarks: 350 },
  { name: "Amit Joshi", totalMarks: 383 },
  { name: "Rahul Verma", totalMarks: 377 }
]
```

Q8.Write a query to Sort students by department (asc) and within department by totalMarks (desc).

```
 printjson(db.students.find().sort({ department: 1, totalMarks: -1 }))
```

✅**Output (names sorted):**

```
CSE → [ "Divya Menon(461)", "Meena Gupta(458)", "Neha Rani(439)", "Ravi Kumar(421)", "Arjun Patel(409)", "Manoj Das(408)", "Rahul Verma(377)" ]
ECE → [ "Sneha Iyer(436)", "Anjali Sharma(394)" ]
IT → [ "Pooja Nair(450)", "Kavya Rao(449)", "Amit Joshi(383)", "Kiran Reddy(350)" ]
MECH → [ "Sandeep Yadav(344)", "Vikram Singh(335)" ]
```

Q9.Write a query to Count total number of students in collection.

```
printjson(db.students.countDocuments())
```

✅**Output:**
15

Q10.Write a query to Count how many students belong to ECE department.

```
printjson(db.students.countDocuments({ department: "ECE" }))
```

✅**Output:**

```
3
```

Q11.Write a query to Count students with more than 400 marks.

```
printjson(db.students.countDocuments({ totalMarks: { $gt: 400 } }))
```

✅**Output:**

```
9
```

Q12.Write a query to Find average totalMarks of all students.

```
printjson(db.students.aggregate([
  { $group: { _id: null, avgMarks: { $avg: "$totalMarks" } } }
]))
```

✅**Output:**

```
{ avgMarks: 404.6 }
```

Q13.Write a query to Find maximum marks scored in python.

```
printjson(db.students.aggregate([
  { $group: { _id: null, maxPython: { $max: "$subjects.python" } } }
]))
```

✅**Output:**

```
{ maxPython: 94 }
```

Q14. Write a query to Find department-wise count of students.

```
printjson(db.students.aggregate([
  { $group: { _id: "$department", count: { $sum: 1 } } }
]))
```

✅**Output:**

```
[
  { _id: "CSE", count: 6 },
  { _id: "ECE", count: 3 },
  { _id: "IT", count: 4 },
  { _id: "MECH", count: 2 }
]
```

Q15.Write a query to Find department-wise average totalMarks.

```
 printjson(db.students.aggregate([
  { $group: { _id: "$department", avgMarks: { $avg: "$totalMarks" } } }
]))
```

✅**Output:**

```
[
  { _id: "CSE", avgMarks: 415.3 },
  { _id: "ECE", avgMarks: 430.3 },
  { _id: "IT", avgMarks: 408 },
  { _id: "MECH", avgMarks: 339.5 }
]
```

Q16.Write a query to Find students having more than 90 in both java and cpp.

```
printjson(db.students.find({ "subjects.java": { $gt: 90 }, "subjects.cpp": { $gt: 90 } }))
```

✅**Output (names):**

```
[ "Meena Gupta", "Pooja Nair", "Divya Menon", "Kavya Rao" ]
```

Q17. Write a query to Find department with highest average totalMarks.

```
printjson(db.students.aggregate([
  { $group: { _id: "$department", avgMarks: { $avg: "$totalMarks" } } },
  { $sort: { avgMarks: -1 } },
  { $limit: 1 }
]))
```

✅**Output:**

```
{ _id: "ECE", avgMarks: 430.3 }
```

Q18. Write a query to Add a new field percentage = (totalMarks/500) * 100.

```
printjson(db.students.aggregate([
  { $addFields: { percentage: { $multiply: [{ $divide: ["$totalMarks", 500] }, 100] } } }
]))
```

✅**Output (first 3 shown):**

```
[
  { name: "Ravi Kumar", totalMarks: 421, percentage: 84.2 },
  { name: "Anjali Sharma", totalMarks: 394, percentage: 78.8 },
  { name: "Kiran Reddy", totalMarks: 350, percentage: 70 }
  ...
]
```

Q19. Write a query to Show only name, department, and newly calculated percentage.

```
 printjson(db.students.aggregate([
  { $project: { name: 1, department: 1, percentage: { $multiply: [{ $divide: ["$totalMarks", 500] }, 100] } }
}
]))
```

✅**Output (first 4):**

```
[
  { name: "Ravi Kumar", department: "CSE", percentage: 84.2 },
  { name: "Anjali Sharma", department: "ECE", percentage: 78.8 },
  { name: "Kiran Reddy", department: "IT", percentage: 70 },
  { name: "Meena Gupta", department: "CSE", percentage: 91.6 }
  ...
]
```

Q20. Write a query to Add field result → Pass if totalMarks ≥ 350 else Fail.

```
printjson(db.students.aggregate([
  { $addFields: { result: { $cond: [{ $gte: ["$totalMarks", 350] }, "Pass", "Fail"] } } }
]))
```

✓Output (name + result only):

```
[
  { name: "Ravi Kumar", result: "Pass" },
  { name: "Anjali Sharma", result: "Pass" },
  { name: "Kiran Reddy", result: "Pass" },
  { name: "Meena Gupta", result: "Pass" },
  { name: "Arjun Patel", result: "Pass" },
  { name: "Sneha Iyer", result: "Pass" },
  { name: "Vikram Singh", result: "Fail" },
  { name: "Pooja Nair", result: "Pass" },
  { name: "Rahul Verma", result: "Pass" },
  { name: "Divya Menon", result: "Pass" },
  { name: "Amit Joshi", result: "Pass" },
  { name: "Neha Rani", result: "Pass" },
  { name: "Sandeep Yadav", result: "Fail" },
  { name: "Kavya Rao", result: "Pass" },
  { name: "Manoj Das", result: "Pass" }
]
```

Q-21.Write a query to find the top 2 students with the highest marks.

```
printjson(db.students.aggregate([
  { $sort: { marks: -1 } },
  { $limit: 2 }
]))
```

✓ Output:

```
[
  { "_id": 5, "name": "John", "marks": 95 },
  { "_id": 2, "name": "Priya", "marks": 92 }
]
```

Q-22.Write a query to find the top 3 students with the lowest marks.

```
printjson(db.students.aggregate([
  { $sort: { marks: 1 } },
  { $limit: 3 }
]))
```

✓ Output:

```
[
  { "_id": 3, "name": "Amit", "marks": 76 },
  { "_id": 1, "name": "Ravi", "marks": 85 },
  { "_id": 4, "name": "Sara", "marks": 89 }
]
```

Q-23.Write a query to Find maximum total marks across all students.

```
printjson(db.students.aggregate([
  {
    $group: {
      _id: null,
      maxMarks: { $max: "$totalMarks" }
    }
  }
]))
```

✅ **Output:**

```
[
  { "_id": "Male", "maxMarks": 95 },
  { "_id": "Female", "maxMarks": 92 }
]
```

Q-24.Write a query to group students by city and count how many students are in each city.

```
 printjson(db.students.aggregate([
  { $group: { _id: "$city", totalStudents: { $sum: 1 } } }
]))
```

✅ **Output:**

```
[
  { "_id": "Hyderabad", "totalStudents": 2 },
  { "_id": "Chennai", "totalStudents": 1 },
  { "_id": "Mumbai", "totalStudents": 1 },
  { "_id": "Delhi", "totalStudents": 1 }
]
```

Q-25.Write a query to calculate the overall class average marks.

```
printjson(
  db.students.aggregate([
    {
      $group: {
        _id: null,
        avgMarks: { $avg: "$totalMarks" }
      }
    }
  ]).toArray()
)
```

✅ **Output:**

```
 [
  {
    _id: null,
    avgMarks: 407.6
  }
]
```

Q-26.Write a query to find students scoring greater than the class average.

```
printjson(
  db.students.aggregate([
    // Step 1: Calculate class average and attach it to all documents
    {
      $group: {
        _id: null,
        avgMarks: { $avg: "$totalMarks" }
      }
    },
    // Step 2: Join this average with all students
    {
      $lookup: {
        from: "students",
        pipeline: [
          {
            $project: { name: 1, department: 1, totalMarks: 1 }
          }
        ],
        as: "students"
      }
    },
    // Step 3: Unwind students
    { $unwind: "$students" },
    // Step 4: Keep only those whose totalMarks > avgMarks
    {
      $match: {
        $expr: { $gt: ["$students.totalMarks", "$avgMarks"] }
      }
    },
    // Step 5: Reshape output
    {
      $project: {
        _id: 0,
        name: "$students.name",
        department: "$students.department",
        totalMarks: "$students.totalMarks",
        avgMarks: 1
      }
    }
  ]).toArray()
)
```

✓ **Output:**

```
 [
  { avgMarks: 407.6,   name: 'Ravi Kumar',    department: 'CSE',    totalMarks: 421  },
  {    avgMarks: 407.6,    name: 'Meena Gupta', department: 'CSE',    totalMarks: 458  },
  {    avgMarks: 407.6,    name: 'Arjun Patel',    department: 'CSE',    totalMarks: 409  },
  {    avgMarks: 407.6,    name: 'Sneha Iyer',    department: 'ECE',    totalMarks: 436  },
  {    avgMarks: 407.6,    name: 'Pooja Nair',    department: 'IT',    totalMarks: 450  },
  {    avgMarks: 407.6,    name: 'Divya Menon',    department: 'ECE',    totalMarks: 461  },
  {    avgMarks: 407.6,    name: 'Neha Rani',    department: 'CSE',    totalMarks: 439  },
  {    avgMarks: 407.6,    name: 'Kavya Rao',    department: 'IT',    totalMarks: 449  },
  {  avgMarks: 407.6,    name: 'Manoj Das',    department: 'CSE',    totalMarks: 408  }]
```

Q-27.Write a query to add a field result as PASS if marks ≥ 80 else FAIL.

```
printjson(db.students.aggregate([
  {
    $addFields: {
      result: { $cond: [{ $gte: ["$marks", 80] }, "PASS", "FAIL"] }
    }
  }
]))
```

✅ **Output:**

```
[
  { "name": "Ravi", "marks": 85, "result": "PASS" },
  { "name": "Priya", "marks": 92, "result": "PASS" },
  { "name": "Amit", "marks": 76, "result": "FAIL" },
  { "name": "Sara", "marks": 89, "result": "PASS" },
  { "name": "John", "marks": 95, "result": "PASS" }
]
```

Q-28.Write a query to project only name and marks of students (exclude _id).

```
printjson(db.students.aggregate([
  { $project: { _id: 0, name: 1, marks: 1 } }
]))
```

✅ **Output:**

```
[
  { "name": "Ravi", "marks": 85 },
  { "name": "Priya", "marks": 92 },
  { "name": "Amit", "marks": 76 },
  { "name": "Sara", "marks": 89 },
  { "name": "John", "marks": 95 }
]
```

Q-29.Write a query to find students whose totalMarks are between 400 and 450.

```
printjson(
  db.students.find(
  { totalMarks: { $gte: 400, $lte: 450 } }
).forEach(printjson)

)
```

✅ **Output:**

```
 {
  _id: ObjectId('68d2461121a2da142e2f9abc'),
  studentRollNumber: 101,
  name: 'Ravi Kumar',
  department: 'CSE',
  age: 20,
  mobileNumber: '9876543210',
  email: 'ravi.kumar@example.com',
  subjects: {
    java: 85,
    c: 78,
    cpp: 80,
    python: 90,
    daa: 88
  },
  totalMarks: 421
}
{
  _id: ObjectId('68d2461121a2da142e2f9ac0'),
  studentRollNumber: 105,
  name: 'Arjun Patel',
  department: 'CSE',
  age: 20,
  mobileNumber: '9876549876',
  email: 'arjun.patel@example.com',
  subjects: {
    java: 78,
    c: 82,
    cpp: 85,
    python: 80,
    daa: 84
  },
  totalMarks: 409
}
{
  _id: ObjectId('68d2461121a2da142e2f9ac1'),
  studentRollNumber: 106,
  name: 'Sneha Iyer',
  department: 'ECE',
  age: 21,
  mobileNumber: '9765432109',
  email: 'sneha.iyer@example.com',
  subjects: {
    java: 88,
    c: 85,
    cpp: 90,
    python: 86,
    daa: 87
  },
  totalMarks: 436
}
{
  _id: ObjectId('68d2461121a2da142e2f9ac3'),
  studentRollNumber: 108,
  name: 'Pooja Nair',
  department: 'IT',
  age: 20,
  mobileNumber: '9988776655',
  email: 'pooja.nair@example.com',
  subjects: {
    java: 85,
    c: 89,
    cpp: 92,
    python: 94,
    daa: 90
  },
  totalMarks: 450
}
{
  _id: ObjectId('68d2461121a2da142e2f9ac7'),
  studentRollNumber: 112,
  name: 'Neha Rani',
  department: 'CSE',
  age: 20,
  mobileNumber: '9090909090',
  email: 'neha.rani@example.com',
  subjects: {
    java: 89,
    c: 85,
    cpp: 87,
    python: 90,
    daa: 88
  },
  totalMarks: 439
}
{
  _id: ObjectId('68d2461121a2da142e2f9ac9'),
  studentRollNumber: 114,
  name: 'Kavya Rao',
  department: 'IT',
  age: 21,
  mobileNumber: '9876554433',
  email: 'kavya.rao@example.com',
  subjects: {
    java: 91,
    c: 87,
    cpp: 89,
    python: 92,
    daa: 90
  },
  totalMarks: 449
}
{
  _id: ObjectId('68d2461121a2da142e2f9aca'),
  studentRollNumber: 115,
  name: 'Manoj Das',
  department: 'CSE',
  age: 22,
  mobileNumber: '9765123456',
  email: 'manoj.das@example.com',
  subjects: {
    java: 82,
    c: 78,
    cpp: 80,
    python: 85,
    daa: 83
  },
  totalMarks: 408
}
```

Q-30.Write a query to sort students first by gender and then by marks descending.

```
printjson(db.students.aggregate([
  { $sort: { gender: 1, marks: -1 } }
]))
```

✅ **Output:**

```
[
  { "name": "Priya", "gender": "Female", "marks": 92 },
  { "name": "Sara", "gender": "Female", "marks": 89 },
  { "name": "John", "gender": "Male", "marks": 95 },
  { "name": "Ravi", "gender": "Male", "marks": 85 },
  { "name": "Amit", "gender": "Male", "marks": 76 }
]
```

Q-31.Write a query to calculate the minimum marks scored in the class.

```
printjson(
  db.students.aggregate([
    {
      $group: {
        _id: null,
        minMarks: { $min: "$totalMarks" }
      }
    }
  ])
)
```

✅ **Output:**

```
[
  {
    "_id": null,
    "minMarks": 335
  }
]
```

Q-32.Write a query to find the student(s) who scored the minimum marks.

```
printjson(
  db.students.aggregate([
    {
      $group: {
        _id: null,
        minMarks: { $min: "$totalMarks" }
      }
    },
    {
      $lookup: {
        from: "students",
        localField: "minMarks",
        foreignField: "totalMarks",
        as: "studentsWithMinMarks"
      }
    },
    {
      $project: {
        _id: 0,
        minMarks: 1,
        studentsWithMinMarks: 1
      }
    }
  ]).toArray()
)
```

Output:

```
[
  {
    minMarks: 335,
    studentsWithMinMarks: [
      {
        _id: ObjectId('68d2461121a2da142e2f9ac2'),
        studentRollNumber: 107,
        name: 'Vikram Singh',
        department: 'MECH',
        age: 23,
        mobileNumber: '9898989898',
        email: 'vikram.singh@example.com',
        subjects: {
          java: 60,
          c: 65,
          cpp: 70,
          python: 68,
          daa: 72
        },
        totalMarks: 335
      }
    ]
  }
]
```

Q-33.Write a query to find the maximum marks scored in each department.

```
printjson(
  db.students.aggregate([
    { $group: { _id: "$department", maxMarks: { $max: "$totalMarks" } } }
  ]).toArray()
)
```

✅ **Output:**

```
[
  { "_id": "CSE", "maxMarks": 458 },
  { "_id": "ECE", "maxMarks": 461 },
  { "_id": "IT", "maxMarks": 450 },
  { "_id": "MECH", "maxMarks": 344 }
]
```

Q-34.Write a query to count the number of students having totalMarks less than 400.

```
printjson(
  db.students.countDocuments({ totalMarks: { $lt: 400 } })
)
```

✅ **Output:**

```
6
```

Q-35.Write a query to display the first 3 students sorted by totalMarks descending.

```
printjson(
  db.students.aggregate([
    { $sort: { totalMarks: -1 } },
    { $limit: 3 },
    { $project: { _id: 0, studentRollNumber: 1, name: 1, totalMarks: 1 } }
  ]).toArray()
)
```

✅ **Output:**

```
[
  { "studentRollNumber": 110, "name": "Divya Menon", "totalMarks": 461 },
  { "studentRollNumber": 104, "name": "Meena Gupta", "totalMarks": 458 },
  { "studentRollNumber": 108, "name": "Pooja Nair", "totalMarks": 450 }
]
```

Q-36.Write a query to Calculate the average marks of students in each department.

```
printjson(
  db.students.aggregate([
    { $group: { _id: "$department", avgMarks: { $avg: "$totalMarks" } } }
  ]).toArray()
)
```

✅ **Output:**

```
 [  {  _id: 'IT',    avgMarks: 408  },  { _id: 'CSE', avgMarks: 418.6666666666667  },
  {  _id: 'ECE', avgMarks: 430.3333333333333 }, { _id: 'MECH',   avgMarks: 339.5  }]
```

Q-37.Write a query to show students with totalMarks between 400 and 450 (inclusive), projecting only name and totalMarks.

```
printjson(
  db.students.aggregate([
    { $match: { totalMarks: { $gte: 400, $lte: 450 } } },
    { $project: { _id: 0, name: 1, totalMarks: 1 } }
  ]).toArray()
)
```

✅ **Output:**

```
[
  { "name": "Arjun Patel", "totalMarks": 409 },
  { "name": "Manoj Das", "totalMarks": 408 },
  { "name": "Ravi Kumar", "totalMarks": 421 },
  { "name": "Neha Rani", "totalMarks": 439 },
  { "name": "Sneha Iyer", "totalMarks": 436 },
  { "name": "Pooja Nair", "totalMarks": 450 }
]
```

Q-38.Write a query to find students whose names start with 'P'.

```
printjson(
  db.students.find({ name: { $regex: /^P/, $options: "i" } }).toArray()
)
```

✓ **Output:**

```
[
  {
    "studentRollNumber": 108,
    "name": "Pooja Nair",
    "department": "IT",
    "age": 20,
    "mobileNumber": "9988776655",
    "email": "pooja.nair@example.com",
    "subjects": { "java": 85, "c": 89, "cpp": 92, "python": 94, "daa": 90 },
    "totalMarks": 450
  }
]
```

Q-39.Write a query to Display only CSE students sorted by totalMarks descending.

```
  printjson(
  db.students.aggregate([
    { $match: { department: "CSE" } },
    { $sort: { totalMarks: -1 } },
    { $project: { _id: 0, name: 1, totalMarks: 1 } }
  ]).toArray()
)
```

✓ **Output:**

```
[
  { "name": "Meena Gupta", "totalMarks": 458 },
  { "name": "Neha Rani", "totalMarks": 439 },
  { "name": "Ravi Kumar", "totalMarks": 421 },
  { "name": "Arjun Patel", "totalMarks": 409 },
  { "name": "Manoj Das", "totalMarks": 408 },
  { "name": "Rahul Verma", "totalMarks": 377 }
]
```

Q-40.Write a query to group students by department and show average, minimum, and maximum marks.

```
printjson(
  db.students.aggregate([
    {
      $group: {
        _id: "$department",
        avgMarks: { $avg: "$totalMarks" },
        minMarks: { $min: "$totalMarks" },
        maxMarks: { $max: "$totalMarks" }
      }
    }
  ]).toArray()
)
```

✅ **Output:**

```
[
  { "_id": "CSE", "avgMarks": 420.8, "minMarks": 377, "maxMarks": 458 },
  { "_id": "ECE", "avgMarks": 430.3, "minMarks": 394, "maxMarks": 461 },
  { "_id": "IT", "avgMarks": 408.0, "minMarks": 350, "maxMarks": 450 },
  { "_id": "MECH", "avgMarks": 339.5, "minMarks": 335, "maxMarks": 344 }
]
```

MongoDB Projection and Joins Queries:

students Collection:

```
[
  { "_id": 1, "name": "Ravi Kumar", "department_id": 101, "age": 20, "marks": 421 },
  { "_id": 2, "name": "Anjali Sharma", "department_id": 102, "age": 21, "marks": 394 },
  { "_id": 3, "name": "Kiran Reddy", "department_id": 103, "age": 22, "marks": 350 }
]
```

departments Collection:

```
[
  { "_id": 101, "dept_name": "CSE", "hod": "Dr. Singh" },
  { "_id": 102, "dept_name": "ECE", "hod": "Dr. Mehta" },
  { "_id": 103, "dept_name": "IT", "hod": "Dr. Rao" }
]
```

Q-1: **Write a query to display only the name and marks of all students.**

Query:

```
db.students.find({}, { _id: 0, name: 1, marks: 1 })
```

Output:

```
[
  { "name": "Ravi Kumar", "marks": 421 },
  { "name": "Anjali Sharma", "marks": 394 },
  { "name": "Kiran Reddy", "marks": 350 }
]
```

Q-2: **Write a query to display student names and ages, excluding marks.**

Query:

```
db.students.find({}, { marks: 0, name: 1, age: 1 })
```

Output:

```
[
  { "name": "Ravi Kumar", "age": 20 },
  { "name": "Anjali Sharma", "age": 21 },
  { "name": "Kiran Reddy", "age": 22 }
]
```

Q-3:**Write a query to display each student along with their department details using a join.**

Query:

```
db.students.aggregate([
  {
    $lookup: {
      from: "departments",
      localField: "department_id",
      foreignField: "_id",
      as: "dept_info"
    }
  }
])
```

Output:

```
[
  {
    "_id": 1,
    "name": "Ravi Kumar",
    "department_id": 101,
    "age": 20,
    "marks": 421,
    "dept_info": [{ "_id": 101, "dept_name": "CSE", "hod": "Dr. Singh" }]
  },
  {
    "_id": 2,
    "name": "Anjali Sharma",
    "department_id": 102,
    "age": 21,
    "marks": 394,
    "dept_info": [{ "_id": 102, "dept_name": "ECE", "hod": "Dr. Mehta" }]
  }
]
```

Q-4:**Write a query to display student names and their department names only.**

Query:

```
db.students.aggregate([
  {
    $lookup: {
      from: "departments",
      localField: "department_id",
      foreignField: "_id",
      as: "dept_info"
    }
  },
  { $project: { _id: 0, name: 1, "dept_info.dept_name": 1 } }
])
```

Output:

```
[
  { "name": "Ravi Kumar", "dept_info": [{ "dept_name": "CSE" }] },
  { "name": "Anjali Sharma", "dept_info": [{ "dept_name": "ECE" }] },
  { "name": "Kiran Reddy", "dept_info": [{ "dept_name": "IT" }] }
]
```

Q-5: **Write a query to display student name, department name, and marks (renaming fields).**

Query:

```
db.students.aggregate([
  {
    $lookup: {
      from: "departments",
      localField: "department_id",
      foreignField: "_id",
      as: "department"
    }
  },
  { $unwind: "$department" },
  { $project: { _id: 0, student_name: "$name", dept_name: "$department.dept_name",
marks: 1 } }
])
```

Output:

```
[
  { "student_name": "Ravi Kumar", "dept_name": "CSE", "marks": 421 },
  { "student_name": "Anjali Sharma", "dept_name": "ECE", "marks": 394 },
  { "student_name": "Kiran Reddy", "dept_name": "IT", "marks": 350 }
]
```

Q-6: **Write a query to display only students who belong to the CSE department along with their marks.**

Query:

```
db.students.aggregate([
  {
    $lookup: {
      from: "departments",
      localField: "department_id",
      foreignField: "_id",
      as: "department"
    }
  },
  { $unwind: "$department" },
  { $match: { "department.dept_name": "CSE" } },
  { $project: { _id: 0, name: 1, "department.dept_name": 1, marks: 1 } }
])
```

Output:

```
[
  { "name": "Ravi Kumar", "department": { "dept_name": "CSE" }, "marks": 421 }
]
```