Table of Content

Unit 1: Introduction to MongoDB

1.1 What is MongoDB?

MongoDB is a **NoSQL**, **document-oriented** database. It stores data in **JSON-like** documents called **BSON** (Binary JSON), making it more flexible than traditional relational databases (RDBMS).

1.2 Difference Between MongoDB and RDBMS

Feature	MongoDB (NoSQL)	RDBMS (MySQL, PostgreSQL)
Data Model	Document-oriented (BSON)	Table-based (rows and columns)
Schema	Dynamic (schema-less)	Fixed (predefined schema)
Joins	Limited joins (embedded docs or refs)	Supports JOIN operations
Scalability	Horizontal (via sharding)	Vertical scaling (limited by hardware)
Transactions	Supported (multi-doc since v4.0)	Supported natively

1.3 Why Use MongoDB?

- Flexible schema design
- Developer-friendly (JSON-style)
- High performance and scalability
- Ideal for modern web apps (Node.js, MERN stack)

1.4 MongoDB Architecture

MongoDB stores data in this structure:

MongoDB Server
└── Database
  └── Collection
    └── Document (key-value pairs)

Key Terms:

- **Database**: A container for collections.
- **Collection**: A group of documents (equivalent to tables).
- **Document**: Actual data, stored in BSON format.

Example Document:

```
{
  "name": "John Doe",
  "age": 22,
  "isStudent": true,
  "skills": ["MongoDB", "Node.js", "React"]
}
```

1.4 Key Concepts to Remember

Term	Description
NoSQL	Non-relational databases (flexible schema)
BSON	Binary JSON – stores MongoDB documents internally
Document	Primary unit of data in MongoDB (like a row, but flexible)
Collection	A group of related documents (like a table)
Schema-less	No fixed structure for fields in documents

1.5 Summary

- MongoDB is a schema-less, NoSQL document database.
- It uses BSON (Binary JSON) for storing documents.
- It's ideal for modern, scalable applications (especially in the MERN stack).
- Data is stored in collections and documents instead of rows and tables.

Unit 2: MongoDB Atlas Setup

2.1 Why MongoDB Atlas?

MongoDB Atlas is a **cloud-based database platform** offered by MongoDB Inc. It helps you:

- Host MongoDB without local installation.
- Easily scale, back up, and manage data.
- Share access with collaborators from anywhere.
- Avoid managing server infrastructure.

Step 1: Create a MongoDB Atlas Account

1. Go to: https://www.mongodb.com/cloud/atlas/register
2. Fill in the details or use Google/GitHub to sign up.
3. After login, click **"Build a Database"**.

Step 2: Create a Free Shared Cluster

1. Choose:
   - **Cloud Provider**: AWS / GCP / Azure
   - **Region**: Choose nearest to your location (e.g., Mumbai for India)
2. Cluster Tier:
   Choose the free option → M0 Sandbox (Always Free)
3. Name your cluster (e.g., Cluster0) and click **Create Cluster**.

Step 3: Configure Security

1. Add a Database User

- Username: student
- Password: strongpassword123 (use your own safe password)

Save this information securely.

2. Add IP Address to Access List

- Choose: **Allow access from anywhere**
- This will set: 0.0.0.0/0

In real-world production apps, **use your current IP only** for better security.

Step 4: Connect to the Cluster

After the cluster is ready:

1. Click **"Connect"** → Choose **"Connect using MongoDB Compass"**.
2. Copy the **Connection String** like:

```
mongodb+srv://student:strongpassword123@cluster0.mongodb.net/
```

Step 5: Install and Use MongoDB Compass

Install MongoDB Compass:

Download from: https://www.mongodb.com/try/download/compass

Install and open Compass.

Connect:

1. Paste the connection URI into Compass.
2. Click **Connect**.

You will see your cluster and databases visually!

2.2 Bonus: Modify Connection URI

To connect via terminal (mongosh), use a modified URI like:

```
mongosh "mongodb+srv://student:<password>@cluster0.mongodb.net/myDatabase"
```

Replace <password> and myDatabase as needed.

2.3 Summary Table

Step	Action
Create Atlas Account	Sign up at mongodb.com
Build Free Cluster	M0 cluster (AWS/GCP/Azure)
Add DB User	Create username + password
Whitelist IP	Use 0.0.0.0/0 for demo purposes
Get Connection String	Use in Compass or shell
Install Compass	GUI client to connect and view documents
Create DB/Collection	Add data manually via Compass

Unit 3 MongoDB Set Up

3.1 MongoDB installation in Windows

Step 1: Download MongoDB Community Server

1. Visit the official MongoDB download page:
   ☞ https://www.mongodb.com/try/download/community
2. Choose the following options:
   - **Version**: Latest Stable (e.g., 6.x)
   - **Platform**: Windows
   - **Package**: .msi (Windows Installer)
3. Click **Download** and wait for the file to finish.

Step 2: Run the MongoDB Installer

1. **Double-click** the .msi file you downloaded.
2. In the **Setup Wizard**, click **Next**.
3. **Choose Setup Type**:
   - Select **Complete** (recommended for beginners).
4. **Service Configuration**:
   - Keep default settings:
     - **Install MongoDB as a Service**
     - Run service as **Network Service User**
5. Optional: You can also install **MongoDB Compass** (GUI tool) from the same wizard.
6. Click **Install** and allow the installer to finish.

Step 3: Verify MongoDB Installation

After installation, MongoDB binaries are usually located at:

```
C:\Program Files\MongoDB\Server\<version>\bin
```

To make the command-line tools accessible from any folder, **add this bin path to the System Environment Variables**:

Add to PATH:

1. Open Windows Search → type “Environment Variables”.
2. Click “Edit the system environment variables”.
3. Click **Environment Variables** button.
4. Under “System variables”, select Path, then click **Edit**.
5. Click **New**, then add the path:

```
C:\Program Files\MongoDB\Server\6.0\bin
```

Adjust version number (6.0) based on what you installed.

6. Click OK → OK → OK to save and exit.

Step 4: Test Installation (via Terminal)

Open **Command Prompt (cmd)** and type:

```
mongod --version
```

If installed correctly, you should see version info like:

```
db version v6.0.6
```

To test MongoDB server is running:

```
mongod
```

This starts the server. You will see logs like:

Waiting for connections on port 27017

Leave this window open while MongoDB is running.

Step 5: Open a New Terminal and Use mongosh

MongoDB now uses mongosh (Mongo Shell) for interacting with the database.

Open a new command prompt window and run:

```
mongosh
```

You'll see:

```
Current Mongosh Log ID: ...
Using MongoDB: 6.0.x
test>
```

You're now connected to the MongoDB shell and ready to start using it!

Default Data Storage Location

When MongoDB runs, it stores data at:

```
C:\data\db
```

If this folder doesn't exist, create it manually:

```
mkdir C:\data\db
```

MongoDB won't start unless this folder exists.

Summary

Step	Description
Download Installer	From MongoDB official website
Install MongoDB	Complete setup with default service
Add to PATH	So you can run commands from anywhere in terminal
Start MongoDB Server	mongod
Connect with Shell	mongosh
Test Insert & Query	Use basic CRUD in MongoDB shell

3.2 Mongoosh

Goal: Set up MongoDB on your local system and interact with it using the **MongoDB Shell (mongosh)**.

What is mongosh?

- mongosh is the **official MongoDB shell**, used to interact with your MongoDB database from the command line.
- It supports:
  - CRUD operations
  - Admin tasks
  - Writing JavaScript in shell
  - Connecting to local or remote MongoDB instances

Step-by-Step Installation (Windows)

If you've already installed MongoDB (from Unit 1), skip to Step 5.

Step 1: Download MongoDB Community Edition

- Go to: https://www.mongodb.com/try/download/community
- Choose:
  - **Platform**: Windows
  - **Package**: .msi
- Click **Download** and install it.

Step 2: Install MongoDB

1. Launch the .msi file.
2. Select **Complete Setup**.
3. Check the box to install **MongoDB as a service**.
4. (Optional) Choose to install **MongoDB Compass**.
5. Complete the installation.

Step 3: Set MongoDB in PATH

To use MongoDB from any terminal window:

1. Go to:
   `C:\Program Files\MongoDB\Server\<version>\bin`
2. Copy the path and add it to **System Environment Variables**:
   - Search: "Edit the system environment variables"
   - Go to: Environment Variables → Path → Edit → New → Paste the path
   - Click OK and apply changes

Step 4: Create MongoDB Data Directory

MongoDB stores its data in a folder. You need to create it:

```
mkdir C:\data\db
```

This is the default directory for MongoDB.

Step 5: Start MongoDB Server

1. Open **Command Prompt** and run:

```
mongod
```

If successful, you'll see logs saying:

Waiting for connections on port 27017

Your database server is now running.

Keep this terminal open! It's running your database server.

Step 6: Open a New Terminal and Launch mongosh

Now, open **another Command Prompt** and run:

```
mongosh
```

This connects to your local MongoDB server.

Basic Commands to Practice

Once inside mongosh, try the following:

1. Create/Select a Database

```
use bookstore
```

If the database doesn't exist, MongoDB creates it when you first insert data.

2. Insert a Document

```
db.books.insertOne({
  title: "Clean Code",
  author: "Robert Martin",
  pages: 464
})
```

3. View Documents

```
db.books.find()
```

Useful mongosh Commands

Command	Description
show dbs	List all databases
use <dbName>	Switch to a database
show collections	List collections in the current DB
db.collection.insertOne()	Insert a single document
db.collection.find()	View all documents
db.collection.drop()	Delete a collection
exit	Exit the shell

Summary Table

Task	Command / Action
Start MongoDB Server	mongod
Open MongoDB Shell	mongosh
Switch DB	use myDB
Insert Data	db.myCollection.insertOne({})
View Data	db.myCollection.find()
Stop Server	Close the terminal where mongod is running

Unit 4: Creating Databases & Collections

4.1 Databases & Collections

Goal: Learn how to **create databases and collections** in MongoDB using the **MongoDB shell (mongosh)**, and understand naming conventions and how collections are created dynamically.

Key Concepts

- **Database**: A container for collections.
- **Collection**: A group of MongoDB documents, like a table in RDBMS.
- **Document**: A single record (in JSON/BSON format).
- MongoDB is **schema-less**, meaning:
  - You don't define tables and columns ahead of time.
  - Fields can vary from document to document in the same collection.

MongoDB Architecture Recap

MongoDB → Database → Collection → Document

4.2 Creating a Database

In MongoDB, **you switch to a database first**. If it doesn't exist, it will be created when you store data in it.

use studentDB

This switches to the database studentDB.
Nothing is created yet — it exists **only in memory** until data is inserted.

4.3 Creating a Collection

Option A: Implicit Creation (Recommended)

MongoDB automatically creates a collection when you insert your first document.

```
db.students.insertOne({
  name: "Ravi",
  department: "Mechanical",
  year: 2
})
```

→ This automatically creates the students collection inside studentDB.

Option B: Explicit Creation (Rarely used)

```
db.createCollection("faculty")
```

This manually creates a collection.

Verifying the Database & Collections

```
show dbs                // Lists all databases
show collections        // Lists all collections in the current DB
```

Practice Example

```
use collegeDB

db.courses.insertMany([
  { name: "Data Structures", credits: 4 },
  { name: "Operating Systems", credits: 3 }
])

db.courses.find()
```

Output:

```
[
  { _id: ObjectId("..."), name: "Data Structures", credits: 4 },
  { _id: ObjectId("..."), name: "Operating Systems", credits: 3 }
]
```

Collection Naming Conventions

- Use **lowercase** letters
- Use **plural** nouns (e.g., students, orders)
- Avoid special characters or spaces
- Use camelCase or underscores (e.g., userProfiles, order_items)

Good: products, courseList, user_accounts

✗ Bad: 123data, My Table, Product$Info

4.4 Summary Table

Action	Command
Switch to a DB	use dbName
Insert Document	db.collection.insertOne({})
Create Collection	db.createCollection("name")
List Collections	show collections
List Databases	show dbs
View Collection Documents	db.collection.find()

Unit 5: Basic CRUD (Insert)

5.1 CRUD Overview

CRUD stands for:

- **C**reate → insertOne(), insertMany()
- **R**ead → find(), findOne()
- **U**pdate → updateOne(), updateMany()
- **D**elete → deleteOne(), deleteMany()

This unit focuses on the **Create** operations.

5.2 MongoDB Document Structure

A **document** in MongoDB is a JSON-like structure (actually stored as BSON) with key-value pairs.

Example:

```
{
  "name": "Anjali",
  "email": "anjali@example.com",
  "age": 21
}
```

Note: MongoDB automatically adds an _id field to each document if not provided.

1. insertOne() – Insert a Single Document

Syntax:

```
db.collection.insertOne({ key1: value1, key2: value2 })
```

Example:

```
use studentDB

db.students.insertOne({
  name: "Karan",
  department: "ECE",
  year: 3
})
```

Output:

```
{
  acknowledged: true,
  insertedId: ObjectId("...")
}
```

2. insertMany() – Insert Multiple Documents

Syntax:

```
db.collection.insertMany([
  { key1: value1, key2: value2 },
  { key1: value1, key2: value2 },
])
```

Example:

```
db.students.insertMany([
  { name: "Deepa", department: "CSE", year: 2 },
  { name: "Arun", department: "MECH", year: 4 }
])
```

Output:

```
{
  acknowledged: true,
  insertedIds: {
    "0": ObjectId("..."),
    "1": ObjectId("...")
  }
}
```

5.3 Nested Documents

MongoDB supports **nested documents** (documents inside documents):

```
db.students.insertOne({
  name: "Nisha",
  contact: {
    email: "nisha@gmail.com",
    phone: "9876543210"
  },
  address: {
    city: "Chennai",
    pincode: 600001
  }
})
```

This structure is flexible and doesn't need a fixed schema.

5.4 Data Types in MongoDB Documents

Type	Example
String	"name": "Ravi"
Number	"age": 22
Boolean	"isActive": true
Array	"skills": ["Node", "MongoDB"]
Object	"profile": { "email": "x@y.com" }
Date	"joined": new Date()

5.5 Practice Task

1. Use/Create a new database:

use libraryDB

2. Insert a single book:

```
db.books.insertOne({
  title: "MongoDB for Beginners",
  author: "Jane Doe",
  year: 2023,
  price: 499
})
```

3. Insert multiple books:

```
db.books.insertMany([
  {
    title: "Learn Node.js",
    author: "John Smith",
    year: 2021,
    price: 699
  },
  {
    title: "JavaScript Essentials",
    author: "Anita Sharma",
    year: 2022,
    price: 399
  }
])
```

4. View inserted documents:

```
db.books.find()
```

5.6 Common Errors to Avoid

Problem	Fix
Forgetting {} around insert data	Always use object format: { key: value }
Typo in collection name	Check spelling – MongoDB will create a new one if mistyped
Using invalid key names	Avoid $, . in field names unless required

5.7 Summary Table

Task	Command Example
Insert one document	db.students.insertOne({ name: "Raj", age: 20 })
Insert many documents	db.students.insertMany([{...}, {...}])
View all documents	db.students.find()
Use nested objects	{ name: "A", contact: { phone: "123" } }

Unit 6: Basic Read/Query Operations

6.1 Key Concepts

- **find()** → Returns a cursor to all matching documents (can return many).
- **findOne()** → Returns the first matching document only.
- **Query Filters** → Allow you to search by fields or conditions.
- **Comparison Operators** → $gt, $lt, $eq, $ne, $in, etc.

1. find() – Retrieve All or Filtered Documents

Syntax:

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

◆ Example: Get All Documents

```
db.students.find()
```

Retrieves all documents in the students collection.

2. findOne() – Retrieve First Match Only

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

Returns the first document where department is "CSE".

3. Query by Field Match

```
db.students.find({ year: 3 })
```

Get all students in 3rd year.

4. Using Comparison Operators

Operator	Meaning	Example
$eq	Equal to	{ year: { $eq: 2 } }
$ne	Not equal to	{ department: { $ne: "ECE" } }
$gt	Greater than	{ year: { $gt: 2 } }
$lt	Less than	{ year: { $lt: 4 } }
$gte	Greater or Equal	{ year: { $gte: 3 } }
$lte	Less or Equal	{ year: { $lte: 2 } }
$in	In array	{ department: { $in: ["CSE", "IT"] } }
$nin	Not in array	{ year: { $nin: [1, 2] } }

5. Project Specific Fields (using projection)

```
db.students.find({ year: 3 }, { name: 1, department: 1, _id: 0 })
```

Returns only the name and department fields for year 3 students (excludes _id).

6. Querying Nested Fields

```
db.students.find({ "contact.phone": "9876543210" })
```

Match nested field values.

7. Query with Multiple Conditions (Logical AND)

```
db.students.find({ department: "CSE", year: 2 })
```

Default behavior is logical **AND**.

8. Logical Operators

Operator	Usage Example
$or	{ $or: [{ year: 2 }, { year: 3 }] }
$and	{ $and: [{ department: "CSE" }, { year: 3 }] }
$not	{ year: { $not: { $gt: 3 } } }
$nor	{ $nor: [{ year: 1 }, { department: "EEE" }] }

6.2 Example Data

```
db.students.insertMany([
  { name: "Aarav", year: 1, department: "CSE" },
  { name: "Bhavana", year: 3, department: "ECE" },
  { name: "Chirag", year: 2, department: "MECH" },
  { name: "Disha", year: 4, department: "CSE" }
])
```

Practice Queries

1. Get all CSE students:

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

2. Get students in year 3 or 4:

```
db.students.find({ year: { $in: [3, 4] } })
```

3. Get students not in MECH department:

```
db.students.find({ department: { $ne: "MECH" } })
```

4. Get only names and years of students:

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

6.3 Summary Table

Task	Example Query
Find all	db.students.find()
Find with filter	db.students.find({ year: 3 })
Use comparison operators	db.students.find({ year: { $gt: 2 } })
Logical OR	db.students.find({ $or: [{ year: 2 }, ...] })
Project specific fields	{ name: 1, _id: 0 }
Nested fields	"contact.phone": "..."

Unit 7: Basic CRUD – Update in MongoDB

7.1 What is an Update?

An **update** modifies fields in existing documents within a collection.

MongoDB provides two main update methods:

- updateOne() – Updates the **first matching** document.
- updateMany() – Updates **all matching** documents.

1 . Syntax of updateOne()

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

Example:

```
db.students.updateOne(
  { name: "Aarav" },
  { $set: { year: 2 } }
)
```

This updates the year field to 2 for the first student named **Aarav**.

2. Syntax of updateMany()

```
db.collection.updateMany(
  { <filter> },
  { <update operation> }
)
```

Example:

```
db.students.updateMany(
  { department: "CSE" },
  { $inc: { year: 1 } }
)
```

This **increments** the year by 1 for all students in the CSE department.

3. Common Update Operators

Operator	Purpose	Example
$set	Set a new value to a field	{ $set: { age: 21 } }
$inc	Increment or decrement a number	{ $inc: { marks: 5 } }
$unset	Remove a field from a document	{ $unset: { mobile: "" } }
$rename	Rename a field	{ $rename: { fullName: "name" }}

7.2 Example Data

You can insert the following data to try updates:

```
db.students.insertMany([
  { name: "Aarav", year: 1, department: "CSE" },
  { name: "Bhavana", year: 3, department: "ECE" },
  { name: "Chirag", year: 2, department: "MECH" },
  { name: "Disha", year: 4, department: "CSE" },
  { name: "Elina", year: 1, department: "EEE" }
])
```

Real-World Examples

Set a New Field

```
db.students.updateOne(
  { name: "Disha" },
  { $set: { passed: true } }
)
```

Increase Year by 1 for All CSE Students

```
db.students.updateMany(
  { department: "CSE" },
  { $inc: { year: 1 } }
)
```

Remove a Field

```
db.students.updateOne(
  { name: "Elina" },
  { $unset: { department: "" } }
)
```

Rename a Field

```
db.students.updateMany(
  {},
  { $rename: { year: "semester" } }
)
```

7.3 Advance Update

1. Upsert (Update or Insert)

When updating a document, if it doesn’t exist, MongoDB can **insert it instead** using the upsert: true option.

Syntax:

```
db.students.updateOne(
  { name: "Farhan" },
  { $set: { year: 1, department: "CSE" } },
  { upsert: true }
)
```

If "Farhan" doesn't exist, it will be created.

Use Case: Useful for idempotent operations like syncing data.

2. Replace a Document (replaceOne)

Replaces the **entire document**, not just a field.

Syntax:

```
db.students.replaceOne(
  { name: "Disha" },
  { name: "Disha", year: 2, department: "ECE" }
)
```

Removes all other fields in the old document.

Use Case: When updating an outdated document schema entirely.

3. Return Updated Document (findOneAndUpdate)

To return the **document after update**, you can use this method.

Syntax:

```
db.students.findOneAndUpdate(
  { name: "Aarav" },
  { $set: { department: "IT" } },
  { returnDocument: "after" } // or returnNewDocument: true in some drivers
)
```

Returns the document **after** the update is applied.

4. Multi-condition Updates

Use **compound filters** for more precise updates.

Example:

```
db.students.updateMany(
  { year: { $lt: 3 }, department: "CSE" },
  { $inc: { year: 1 } }
)
```

Promotes only CSE students in years 1 and 2.

5. Use writeConcern (Optional Advanced)

For production apps, especially with replicas, mention that updates can be made **reliable** using write concern.

Example:

```
db.students.updateOne(
  { name: "Aarav" },
  { $set: { year: 4 } },
  { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```

7.3 Common Mistakes

Mistake	Fix
Using update without $set	Always wrap new data in $set, $inc, etc.
Forgetting to use a filter	Always specify {} or a condition to avoid errors.
Confusing updateOne with updateMany	Use updateOne for a single update.

7.4 Summary Table

Method	Use Case
updateOne()	Update first matching document
updateMany()	Update multiple documents
$set	Set or overwrite values
$inc	Add/subtract numeric fields
$unset	Delete a field from document
$rename	Rename field in document

Unit 8: Basic CRUD: delete in MongoDB

8.1 MongoDB Delete Operations

MongoDB supports deleting one or multiple documents using:

1. deleteOne()
2. deleteMany()

1. deleteOne() – Delete a Single Document

Deletes the **first** document that matches the filter condition.

Syntax:

```
db.collection.deleteOne({ <filter> })
```

Example:

```
db.students.deleteOne({ name: "Chirag" })
```

Removes only one document where name is "Chirag".

2. deleteMany() – Delete Multiple Documents

Deletes **all documents** that match the filter.

Syntax:

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

Example:

```
db.students.deleteMany({ department: "EEE" })
```

Removes all students in the EEE department.

3. Delete with Multiple Conditions

You can use logical or comparison operators in your filter.

Example:

```
db.students.deleteMany({
  year: { $lt: 2 },
  department: "CSE"
})
```

Deletes all CSE students in year 1.

4. Preview Before Deleting (Recommended)

Always preview your delete query using find():

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

🔍 **Why?** Helps prevent accidental data loss.

5. Delete by _id

Deleting documents by their unique _id is often the safest.

Example:

```
db.students.deleteOne({ _id: ObjectId("664fa8e12a2...") })
```

You must use ObjectId() for _id values.

6. Use writeConcern (Optional for Production)

Ensure the delete is acknowledged by a majority of replica set members.

Example:

```
db.students.deleteMany(
  { year: 1 },
  { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```

8.2 Sample Data to Practice

```
db.students.insertMany([
  { name: "Aarav", year: 1, department: "CSE" },
  { name: "Bhavana", year: 3, department: "ECE" },
  { name: "Chirag", year: 2, department: "MECH" },
  { name: "Disha", year: 4, department: "CSE" },
  { name: "Elina", year: 1, department: "EEE" }
])
```

Practice Tasks

1. Delete one student named "Chirag":

```
db.students.deleteOne({ name: "Chirag" })
```

2. Delete all students in the "ECE" department:

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

3. Delete students in year 1 from "EEE":

```
db.students.deleteMany({ year: 1, department: "EEE" })
```

4. Safely delete using _id:

```
db.students.findOne({ name: "Aarav" }) // Get the _id
db.students.deleteOne({ _id: ObjectId("...") })
```

8.3 Advanced MongoDB Delete Concepts

1. Bulk Write Operations with Delete

Perform multiple delete, insert, update operations in a single call — improves performance.

```
db.students.bulkWrite([
  { deleteOne: { filter: { name: "John" } } },
  { deleteMany: { filter: { department: "EEE" } } }
])
```

Efficient for batch processing or scheduled jobs.

2. Soft Deletes (Recommended for Real Apps)

Instead of physically deleting documents, **mark them as deleted**.

Why? Enables undo, audit logs, and prevents accidental data loss.

Example:

```
db.students.updateOne(
  { name: "Aarav" },
  { $set: { deleted: true, deletedAt: new Date() } }
)
```

To query only active students:

```
db.students.find({ deleted: { $ne: true } })
```

Reversible, auditable, and production-safe.

3. Delete with Collation (Case Insensitive Delete)

By default, deletes are **case-sensitive**. You can use collation to ignore case.

```
db.students.deleteOne(
  { name: "aarav" },
  { collation: { locale: "en", strength: 2 } }
)
```

Deletes "Aarav" or "aarav" – case-insensitive.

4. Using Projection to Return Fields Before Delete

Use findOneAndDelete() to return the deleted document:

```
db.students.findOneAndDelete(
  { name: "Disha" },
  { projection: { _id: 0, name: 1, year: 1 } }
)
```

Useful if you want to log deleted data.

When to Use What?

Feature	Use Case
Soft Delete	Undo delete, audit trail
Bulk Delete	Batch processing or cleanup tasks
Collation	Case-insensitive delete matching
findOneAndDelete	Return deleted data for logging/backup

8.4 Summary Table

Method	Purpose	Deletes
deleteOne()	Deletes first match	Only one document
deleteMany()	Deletes all matches	Multiple documents
find()	Preview your filter result	Non-destructive
_id	Most accurate deletion filter	One unique document

Unit 9: JSON and BSON

9.1 Overview

MongoDB internally uses **BSON**, but developers typically interact with it using **JSON**. While they look similar, they differ in **structure, data types, and purpose**.

9.2. What is JSON?

JSON (JavaScript Object Notation) is a lightweight data-interchange format.

- Easy to read and write
- Text-based
- Used widely in APIs, web apps, and configurations

Example:

```
{
  "name": "Aarav",
  "age": 22,
  "skills": ["JavaScript", "React"]
}
```

9.3 What is BSON?

BSON (Binary JSON) is a binary-encoded serialization of JSON-like documents.

- Used internally by MongoDB
- Supports more data types than JSON
- Faster to encode/decode and store efficiently

Meaning: JSON is for communication. BSON is for **internal storage** and **transport**.

9.4 JSON vs BSON: Key Differences

Feature	JSON	BSON
Format	Text	Binary
Used In	Frontend, APIs, config files	MongoDB internal engine
Data Types	Limited (string, number, bool)	Extended (Date, ObjectId, BinData)
Readability	Human-readable	Machine-readable only
Size	Lightweight	Slightly larger (stores metadata)
Speed	Slower to parse	Fast parsing (binary optimized)

Supports Comments	✗	✗ (same as JSON)

9.5 Extra Data Types in BSON

Data Type	Description	Example
ObjectId	Unique ID MongoDB assigns to docs	"_id": ObjectId("656ef...")
Date	Stores date and time	"createdAt": ISODate("2025-06-13T10:00:00Z")
Binary	Stores binary data like files	"file": BinData(...)
Decimal128	High-precision decimal type	"price": NumberDecimal("19.99")

9.6 BSON Example (raw view in Compass or mongosh)

```
{
  "_id": ObjectId("665af093e3d1c2a0be1c1234"),
  "name": "Priya",
  "joinedAt": ISODate("2024-05-12T10:23:00Z"),
  "salary": NumberDecimal("100000.50")
}
```

This **won't work** in plain JSON — these types are BSON-specific.

9.7 Hands-on Exercise

1. Insert a document with extended BSON types:

```
db.employees.insertOne({
  name: "Rahul",
  joinedAt: new Date(),
  skills: ["MongoDB", "Node.js"],
  salary: NumberDecimal("65000.75")
})
```

2. View it in **MongoDB Compass** → notice how types like Date and Decimal are visualized.
3. Try exporting as JSON → compare with Compass/BSON view.

9.8 Why BSON in MongoDB?

- BSON supports more types → better data modeling
- Efficient for parsing/querying
- Auto-generates _id (ObjectId)
- Works well with **binary storage and fast transport**

"BSON Supports More Types" → Better Data Modeling?

JSON (used in frontend and APIs) supports a **limited set of data types**:

- Strings
- Numbers
- Booleans

- Arrays
- Objects (nested key-value pairs)
- null

But when you're storing **real-world data** in a database, you often need **more precision, uniqueness, or structure**, which JSON can't express.

That's where **BSON** comes in.

Extra BSON Types in MongoDB

Here are the **most useful BSON data types** that make MongoDB more powerful and efficient for **real-life application data modeling**:

1. ObjectId – Unique Document ID

Use Case: Every document needs a unique identifier.

Why it's useful:

- It encodes timestamp, machine ID, process ID, and counter — **unique and sortable**
- Automatically generated if you don't provide _id

Example:

```
{
  "_id": ObjectId("666b2c123abc456789d01234"),
  "name": "Raj"
}
```

When to use: Use it for every document's _id field (default).

2. Date / ISODate – Precise Timestamp

Use Case: Storing timestamps for created/updated events.

Why it's useful:

- Allows date queries like $gt, $lt, sorting, TTL expiration
- Used in audit logs, bookings, user logins, etc.

Example:

```
{
  "username": "sita",
  "createdAt": ISODate("2025-06-13T09:00:00Z"),
  "lastLogin": new Date()
}
```

When to use: Whenever you track time — created/updated date, appointment time, session expiry, etc.

3. Decimal128 – High-precision Numbers

Use Case: Financial data like prices, billing, interest rates

Why it's useful:

- JSON can lose precision with floating point
- Decimal128 stores **exact** values with high precision

Example:

```
{
  "product": "Laptop",
  "price": NumberDecimal("79999.99")
}
```

When to use: Billing, salaries, invoice amounts, currency conversion, interest rates, etc.

4. Binary Data (BinData) – Store files or media

Use Case: You want to store image data, encrypted files, or PDFs directly in the DB.

Why it's useful:

- BSON can handle binary streams like images, audio, etc.
- Used in GridFS or when integrating with external file systems

Example:

```
{
  "userId": ObjectId("..."),
  "profilePicture": BinData(0, "base64_encoded_image_data_here")
}
```

When to use: Profile pictures, document uploads, storing blobs like QR codes.

5. Timestamps (BSON Timestamp type) – Event logging

Use Case: High-volume write operations like logging

Why it's useful:

- BSON Timestamp is a 64-bit value including time and increment counter
- Useful in **replication** and **event logs**

When to use: Internal use by MongoDB, rarely needed manually.

Real-Life Examples of Better Data Modeling

E-Commerce Order Schema

```
{
  _id: ObjectId("..."),
  userId: ObjectId("..."),
  items: [
    { productId: ObjectId("..."), quantity: 2, price: NumberDecimal("2499.99") }
```

```
  ],
  total: NumberDecimal("4999.98"),
  placedAt: new Date(),
  status: "Processing"
}
```

Benefits of BSON:

- ObjectId for unique references
- NumberDecimal for financial accuracy
- Date for timestamped queries

Employee Payroll System

```
{
  name: "Arjun",
  salary: NumberDecimal("65000.50"),
  doj: ISODate("2022-11-01T09:30:00Z"),
  bonusEligible: true
}
```

Why BSON types matter:

- JSON can't handle precise salaries (65000.50 may become 65000.5)
- ISODate enables date-based filters like "employees joined in 2022"

Invoice Archiving (with PDF Binary)

```
{
  invoiceId: "INV-2043",
  customer: "Meena",
  totalAmount: NumberDecimal("1399.99"),
  date: new Date(),
  pdf: BinData(0, "base64_string_of_pdf_data")
}
```

BSON allows storing a PDF binary directly — which JSON can't do.

When to Use JSON vs BSON?

Situation	Use JSON	Use BSON
Web APIs / Frontend apps	✓	✗
Inside MongoDB database	✗	✓
Need for dates, precision, IDs	✗ JSON lacks precision	BSON supports it
Need to store files or binary	✗ Not possible	BinData supported

9.9 Summary

Topic	JSON	BSON
Format	Text	Binary
MongoDB Role	Input/output layer	Internal storage + transport

Data Types	Basic only	Rich types (ObjectId, Date, etc.)
Use Case	Frontend, REST APIs	Backend database engine

Unit 10: Mongo Compass – Visual Interface for MongoDB

What is MongoDB Compass?

MongoDB Compass is a **desktop GUI tool** provided by MongoDB to interact with your database **visually** instead of through the terminal (mongosh).

It allows you to:

- View documents in collections
- Run queries without writing full shell syntax
- Insert, edit, delete documents
- Analyze schema and indexes
- Optimize performance

10.1 How to Install MongoDB Compass

Download:

Go to the official MongoDB Compass page

1. Choose your OS (Windows / Mac / Linux)
2. Download and run the installer
3. Launch Compass

No need for command-line — just install and run.

10.2 Connecting to MongoDB

When you open Compass, you'll see a connection screen asking for a **connection string** (MongoDB URI).

1. **Atlas Cloud Cluster**

If using MongoDB Atlas, use:

```
mongodb+srv://<username>:<password>@cluster0.xxxxxx.mongodb.net/test
```

- Replace <username>, <password>, and cluster name
- Paste it into Compass and click **Connect**

2. **Local MongoDB**

If running MongoDB locally:

```
mongodb://localhost:27017
```

10.3 Key Sections in Compass

1. **Databases Tab**

- Lists all databases in your MongoDB instance
- You can **create** or **drop** a database from here

2. **Collections**

- Inside each DB are **collections** (like tables in SQL)
- You can:
  - View documents
  - Create new documents
  - Run queries
  - View indexes
  - Analyze schema

10.4 Viewing & Editing Documents

Browse Documents:

- Click on a collection
- You'll see all documents listed in a table-like structure
- Expand nested fields and arrays

Insert a Document:

Click Insert Document → input your JSON:

```
{
 "name": "Anita",
  "age": 23,
  "department": "CSE"
}
```

Edit Document:

- Hover on a document → click Edit
- Modify fields and save

Delete Document:

- Hover on a document → click Delete

10.4 Querying Data in Compass

You don't need full shell syntax — just use **JSON-based filters**.

Examples:

Find all students in CSE:

```
{ "department": "CSE" }
```

Find students older than 21:

```
{ "age": { "$gt": 21 } }
```

Combine filters (AND):

```
{ "age": { "$gt": 21 }, "department": "CSE" }
```

Use OR:

```
{ "$or": [ { "age": { "$gt": 25 } }, { "name": "Anita" } ] }
```

10.5 Schema Tab

MongoDB Compass can analyze your collection's schema to:

- Show field types and frequencies
- Detect anomalies (e.g., missing fields, mixed types)
- Recommend indexes for performance

🎯 Great for students to **understand how documents are structured**.

10.6 Performance Tab (Advanced)

- Shows query execution time
- Query Planner insights
- Index usage stats

10.7 MongoDB Compass – Mini Assignment

Scenario: College Student Records

You are working on a project to manage student records for a college. Use **MongoDB Compass** to create a college database and perform various operations on a students collection.

Step-by-Step Tasks

1. Create Database and Collection

- Open MongoDB Compass
- Connect to your local MongoDB or Atlas URI
- Click Create Database
    - **Database name**: college

- **Collection name**: students

2. Insert Student Records

Insert the following **5 documents** into the students collection using **Insert Document** button:

```
{
  "name": "Aryan",
  "age": 21,
  "department": "CSE",
  "subjects": ["DSA", "OS", "DBMS"]
}
{
  "name": "Divya",
  "age": 22,
  "department": "ECE",
  "subjects": ["EMT", "DSP", "VLSI"]
}
{
  "name": "Rahul",
  "age": 20,
  "department": "MECH",
  "subjects": ["Thermo", "FEM", "CAD"]
}
{
  "name": "Sneha",
  "age": 23,
  "department": "CSE",
  "subjects": ["CN", "ML", "AI"]
}
{
  "name": "Pranav",
  "age": 19,
  "department": "EEE",
  "subjects": ["Circuits", "Machines", "Control"]
}
```

3. Read (Query) the Data

Using the **filter bar** in Compass, try the following queries:

1. Get all students from the **CSE** department

```
{ "department": "CSE" }
```

2. Find students **older than 20**

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

3. Students enrolled in **ML** subject

```
{ "subjects": "ML" }
```

4. Students in **CSE OR ECE**

```
{ "$or": [ { "department": "CSE" }, { "department": "ECE" } ] }
```

4. Update Documents

1. Update Rahul's age to **21**
   - Find by { "name": "Rahul" }
   - Click Edit, change "age": 21
2. Add a new field to Sneha:

```
"year": "Final"
```

3. Rename Pranav's department from EEE to Electrical

5. Delete Documents

1. Delete Aryan's record from the collection
   - Find by { "name": "Aryan" }
   - Click Delete
2. Delete all students below age 21
   - Use this filter:

```
{ "age": { "$lt": 21 } }
```

   - Delete matching results manually or with bulk delete option

6. Bonus Tasks (Optional)

- Add an **index** on the department field
- View the **schema tab** and note data types and field stats
- Export the current documents to a .json file

10.8 Summary

Feature	Description
GUI Interface	No need to use shell – visual interaction
Browse & Query Data	Use JSON queries to filter and view results
Insert/Edit/Delete	One-click document manipulation
Schema Analysis	Auto-detect data structure and field statistics
Indexing	Create and test indexes visually

Unit 11: Schema-less Design Concepts in MongoDB

11.1 Overview

MongoDB is a **NoSQL document database** that follows a **schema-less design** philosophy. This means:

- You don't have to predefine the structure (schema) of documents.
- Documents in the same collection **can have different fields and data types**.
- You gain **flexibility** but must manage **consistency** manually in application logic.

11.2 Why Schema-less?

Benefits:

- **Agile development**: Add/remove fields as features evolve.
- **Faster iteration**: No need to perform migrations like in SQL.
- **Handles unstructured data**: Logs, IoT, and user-generated content.
- **Ideal for dynamic or user-defined data**.

Trade-offs:

- Harder to enforce data consistency.
- Complex validation logic may shift to the app/backend.
- Indexing performance can suffer with too much schema variability.

11.3 Schema-less vs Structured (RDBMS) – Quick Comparison

Feature	MongoDB (Schema-less)	SQL Databases (Structured)
Flexibility	High	Low
Enforced structure	No (optional via schema)	Yes
Data normalization	Not required	Required
Ideal for	Evolving data, JSON APIs	Fixed schema business apps

11.4 Flat Document and Nested Document

Example 1: Flat Document

```
{
  "name": "Alice",
  "age": 24,
  "department": "CSE",
  "year": "Final"
}
```

- All fields are **top-level**
- Easy to index and query
- **Suited for simple data structures**

Example 2: Nested Document

```
{
  "name": "Bob",
  "age": 22,
  "department": "ECE",
  "address": {
    "city": "Bangalore",
    "pincode": 560001
  },
  "courses": [
    { "name": "VLSI", "marks": 85 },
    { "name": "DSP", "marks": 90 }
  ]
}
```

- address is a **nested object**
- courses is an **array of embedded objects**
- Reflects real-world hierarchical data

Flat vs Nested: When to Use?

Use Case	Flat	Nested/Embedded
Simple user profiles	✓	✗
Complex sub-documents (e.g., orders with items)	✗	✓
Optimized read performance	✓ (for shallow reads)	✓ (if embedding avoids extra queries)
Many-to-many relationships	✗ (embedding not ideal)	✗ Use referencing instead

11.5 Real-World Examples

School App – Students

Flat:

```
{
  "name": "Sneha",
  "roll": 45,
  "class": "10A",
  "maths_marks": 80,
  "science_marks": 85
}
```

Nested:

```
{
  "name": "Sneha",
  "roll": 45,
  "class": "10A",
  "marks": {
    "maths": 80,
    "science": 85
  }
}
```

Nested version is cleaner and groups related fields.

11.6 Validation in Schema-less MongoDB

What is Validation?

MongoDB is schema-less, but that doesn't mean it's **structure-less**.
You can define **rules or constraints** on what kind of documents can be inserted into a collection.

This process is called **Schema Validation**. It helps ensure:

- Required fields are present
- Fields are of correct type
- Nested structures conform to expected format

Why Validation in MongoDB?

Even though MongoDB allows flexible documents, **data consistency** is important when:

- Your app expects certain fields to always exist
- You want to prevent garbage or incorrect data
- You collaborate with teams and want to maintain a common data structure

Where Is Validation Applied?

Validation is defined **at the collection level** using **$jsonSchema**, which enforces structure using rules similar to Mongoose or JSON Schema standards.

You apply it using:

- db.createCollection() with validation
- collMod to modify existing collections

Syntax: $jsonSchema Example

```
db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "age", "courses"],
      properties: {
        name: { bsonType: "string" },
        age: { bsonType: "int", minimum: 1 },
        courses: {
          bsonType: "array",
          items: {
            bsonType: "object",
            required: ["title", "marks"],
            properties: {
              title: { bsonType: "string" },
              marks: { bsonType: "int", minimum: 0, maximum: 100 }
            }
          }
        }
      }
    }
  }
})
```

What this does:

- Ensures every student has name, age, and courses
- courses is an array of objects, each with title (string) and marks (int 0–100)

Validation Levels

You can choose **how strictly** MongoDB applies the rules.

1. Validation Level

Level	Description
strict	Only valid documents are inserted/updated (default)
moderate	Only newly inserted fields are validated; existing docs not validated

validationLevel: "strict" // or "moderate"

2. Validation Action

Action	Effect
error	Reject invalid data (default)
warn	Log warning to console, but allow invalid data

validationAction: "error" // or "warn"

Examples

Inserting Valid Document

```
db.students.insertOne({
  name: "Ravi",
  age: 20,
  courses: [
    { title: "Math", marks: 90 },
    { title: "Physics", marks: 85 }
  ]
})
```

Passes validation.

Inserting Invalid Document

```
db.students.insertOne({
  name: "Deepa",
  age: "twenty-one",   // Wrong type
  courses: []
})
```

Error: age must be an int

Updating Validation for Existing Collection

If a collection already exists:

```
db.runCommand({
  collMod: "students",
  validator: {
    $jsonSchema: { /* schema here */ }
  },
  validationLevel: "strict",
  validationAction: "error"
})
```

Real-World Use Case

Employee Collection Example

```
db.createCollection("employees", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["empId", "name", "salary"],
      properties: {
        empId: { bsonType: "int" },
        name: { bsonType: "string" },
        salary: { bsonType: "double", minimum: 0 },
        department: { bsonType: "string" },
        isPermanent: { bsonType: "bool" }
```

```
        }
      }
    }
})
```

Validating Nested Objects & Arrays

MongoDB supports **deep nested validation**.

Example: Order Document

```
{
  "orderId": 123,
  "items": [
    { "productId": "P01", "qty": 2, "price": 50.0 },
    { "productId": "P02", "qty": 1, "price": 100.0 }
  ]
}
```

Schema for this can enforce:

- items must be an array
- Each item must have productId (string), qty (int), price (double)

Tools That Support Schema Validation

Tool	How It Helps
MongoDB Compass	UI to define JSON schema validator
Mongoose (Node.js)	Define schema + validation using models
Atlas UI	Set schema from cloud console

Summary

- MongoDB is schema-less but allows validation for consistency
- Use $jsonSchema for defining field types and constraints
- Choose validation level and action (strict/error, moderate/warn)
- Validation supports deep nested objects and arrays
- Integrates well with Mongoose and Compass

Unit 12: Embedding vs Referencing in MongoDB

12.1 What is Data Modelling?

Data modelling in MongoDB means deciding **how to structure your documents** for efficient reading, writing, and scaling.

Unlike relational databases (RDBMS), MongoDB gives you two main ways to represent relationships:

- **Embedding** (denormalization)
- **Referencing** (normalization)

12.2 Understanding Relationships

Real-World Relationships

Type	Example
1:1	User → Profile
1:N	Blog Post → Comments
N:N	Students ↔ Courses

MongoDB supports all of these using **embedded documents or references**.

Embedding (Denormalization)

Definition:

Embedding means **storing related data in the same document**.

Use When:

- Data is **tightly coupled**
- You **always fetch** related data together
- You want **fewer queries**

Example: Blog Post with Embedded Comments

```
{
  _id: ObjectId("post1"),
  title: "MongoDB Tips",
  author: "Ravi",
  comments: [
    { user: "Amit", text: "Nice post!", date: ISODate() },
    { user: "Sneha", text: "Helpful!", date: ISODate() }
  ]
}
```

Fast reads
Great for displaying entire post + comments

❌ If comments grow large, the document size limit (16MB) becomes a problem
❌ Hard to update individual comments

Referencing (Normalization)

Definition:

Referencing means **storing related data in separate documents**, connected via IDs.

Use When:

- Data is **accessed independently**
- Frequent **updates** to sub-data
- You want to **avoid duplication**

Example: Post and Comments in Separate Collections

```
// Post
{
  _id: ObjectId("post1"),
  title: "MongoDB Tips",
  author: "Ravi"
}

// Comment
{
  _id: ObjectId("comment1"),
  postId: ObjectId("post1"),
  user: "Amit",
  text: "Nice post!"
}
```

Scales well
Better when sub-items are large or accessed separately
❌ Requires manual joins using $lookup

When to Embed vs Reference

Criteria	Embed	Reference
Access pattern	Always together	Access separately
Data size	Small & bounded	Growing or unbounded
Read performance	Faster (1 query)	Slower (requires join)
Write frequency	Rare writes to subdocs	Frequent writes/updates
Document growth concern	No	Yes

12.3 Types of Relationships

One-to-One (1:1)

Example: User → Profile

Embed:

```
{
  _id: ObjectId("user1"),
  name: "Asha",
  profile: {
    age: 28,
    bio: "Developer"
  }
}
```

Reference:

```
// User
{ _id: ObjectId("user1"), name: "Asha" }

// Profile
{ _id: ObjectId("profile1"), userId: ObjectId("user1"), age: 28, bio: "Developer" }
```

➡️ Prefer **embedding** unless profile becomes large or shared.

One-to-Many (1:N)

Example: Product → Reviews

Embed:

```
{
  _id: "product1",
  name: "Laptop",
  reviews: [
    { user: "Ram", rating: 5 },
    { user: "Sita", rating: 4 }
  ]
}
```

Reference:

```
// Product
{ _id: "product1", name: "Laptop" }

// Review
{ _id: "review1", productId: "product1", user: "Ram", rating: 5 }
```

➡️ Use **reference** if there are many reviews or frequent writes.

Many-to-Many (N:N)

Example: Students ↔ Courses

Reference with Cross-Referencing

```
// Student
{
  _id: "stu1",
  name: "Manoj",
  courses: ["course1", "course2"]
}

// Course
{
  _id: "course1",
  title: "Web Dev",
  enrolledStudents: ["stu1", "stu2"]
}
```

➡️ Always use **references** for N:N relationships.

Example Using $lookup (Manual Join)

Join posts and comments:

```
db.posts.aggregate([
  {
    $lookup: {
      from: "comments",
      localField: "_id",
      foreignField: "postId",
      as: "comments"
    }
  }
])
```

Combines two collections like SQL JOIN

12.4 Real-World Modeling Scenarios

Scenario	Suggestion
User → Settings (1:1)	Embed
Order → Items (1:N)	Embed (if bounded)
Product → Reviews (1:N)	Reference (large list)
Student ↔ Courses (N:N)	Reference both ways
Blog Post → Comments	Embed if small, else ref

12.5 Summary

- MongoDB allows **embedding** and **referencing** to model relationships
- Use **embedding** when related data is small, static, and always needed
- Use **referencing** when data is large, dynamic, or shared
- Understand relationship types (1:1, 1:N, N:N) and apply accordingly
- $lookup enables joining referenced documents in aggregation

Unit 13: Indexing Basics in MongoDB

13.1 What is an Index?

An **index** in MongoDB is like an index in a book—it helps you find information **faster** without scanning everything.

By default, MongoDB creates an index on _id for every document. You can create **additional indexes** on fields you query frequently.

13.2 Why Use Indexes?

Without indexes, MongoDB does a **collection scan**: checks every document one by one — which is **slow** for large datasets.

With indexes:

- **Read queries** become faster
- Improves **sorting** and **filtering**
- Supports **uniqueness** constraints

13.3 Types of Indexes

1. Single Field Index

Used on one field (e.g., name, age)

```
db.users.createIndex({ name: 1 }) // Ascending index
```

1 means ascending, -1 means descending

Query example:

```
db.users.find({ name: "Ravi" })
```

With an index on name, this runs much faster.

2. Compound Index

Index on **multiple fields**.

```
db.users.createIndex({ age: 1, city: 1 })
```

Speeds up queries like:

```
db.users.find({ age: 30, city: "Delhi" })
```

Order matters:

- The above index supports queries on { age } or { age, city }
- But not just { city } — unless age is also used

3. Multikey Index

Used when the indexed field is an **array**.

```
// Sample document
{
  name: "Raj",
  skills: ["Node", "MongoDB", "React"]
}

// Create multikey index
db.users.createIndex({ skills: 1 })
```

Allows queries like:

```
db.users.find({ skills: "MongoDB" })
```

MongoDB automatically creates **multikey indexes** when you index an array field.

13.4 Viewing Indexes

```
db.users.getIndexes()
```

Shows all indexes on a collection.

13.5 Dropping an Index

```
db.users.dropIndex("name_1")
```

Or drop all:

```
db.users.dropIndexes()
```

13.6 Performance Tip – explain()

Use explain() to see if a query uses an index:

```
db.users.find({ name: "Amit" }).explain("executionStats")
```

Look for:

- indexName
- nReturned
- totalDocsExamined (lower is better)

13.7 Practical Example

```
db.products.insertMany([
  { name: "Laptop", category: "Electronics", price: 60000 },
  { name: "Phone", category: "Electronics", price: 30000 },
  { name: "Shoes", category: "Footwear", price: 4000 },
])
```

```
// Index on category
db.products.createIndex({ category: 1 })

// Compound index on category and price
db.products.createIndex({ category: 1, price: -1 })

// Query using index
db.products.find({ category: "Electronics" }).sort({ price: -1 })
```

13.8 Indexing Trade-Offs

Benefit	Cost
Fast read/query	Slower write performance
Efficient sorting/filtering	Takes disk space
Supports uniqueness	Indexes must be maintained

13.9 Check how many documents were scanned or compared

Syntax:

```
db.collection.find(query).explain("executionStats")
```

Example:

```
db.users.find({ name: "Ravi" }).explain("executionStats")
```

Key Fields in the Output:

- **nReturned**: Number of documents returned by the query
- **totalDocsExamined**: Number of documents scanned
- **totalKeysExamined**: Number of index entries scanned

How to Interpret:

Field	Meaning
totalDocsExamined	Total number of documents MongoDB looked at (with or without index)
totalKeysExamined	Number of index entries scanned (if index used)
nReturned	Number of results returned to the client

Sample Output (Simplified):

```
{
  "executionStats": {
    "nReturned": 1,
    "totalKeysExamined": 1,
    "totalDocsExamined": 1,
    "executionTimeMillis": 0,
    ...
  }
}
```

Red Flag:

If totalDocsExamined is **much higher** than nReturned, your query **did not use an index** efficiently (or at all).

Use this insight to:

- Add or improve indexes
- Restructure queries

13.9 Summary

- Indexes boost read performance
- Use createIndex() for single, compound, multikey indexes
- Use getIndexes() to list and dropIndex() to remove
- Use explain() to check performance
- Don't over-index: every index has a cost

Unit 14: Advanced Index Types in MongoDB

14.1 Specialized Indexes

MongoDB supports **specialized indexes** tailored for search, performance, and data lifecycle needs.

1. TTL (Time-To-Live) Index

Use Case:

Automatically delete documents after a specific time — perfect for **logs, sessions, temporary data**, etc.

How It Works:

- You create a special index on a field of type Date
- MongoDB checks and deletes expired documents in the background

Example:

```
// Insert a session with createdAt timestamp
db.sessions.insertOne({
  user: "john",
  createdAt: new Date()
})

// Create TTL index to expire docs after 60 seconds
db.sessions.createIndex(
  { createdAt: 1 },
  { expireAfterSeconds: 60 }
)
```

Note: TTL indexes can only be used on **single fields** containing Date values.

2. Text Index

Use Case:

Full-text search across one or more string fields. Ideal for blogs, products, search features.

How It Works:

- Supports **tokenization**, **case-insensitive** search
- You can search for keywords using $text

Example:

```
// Sample collection
db.articles.insertMany([
  { title: "MongoDB Tutorial", content: "Learn indexing and aggregation" },
  { title: "NodeJS Guide", content: "Working with Express and MongoDB" }
])

// Create text index on multiple fields
db.articles.createIndex({ title: "text", content: "text" })

// Search using $text
db.articles.find({ $text: { $search: "MongoDB" } })
```

Only one **text index per collection** allowed, but it can include multiple fields.

3. Hashed Index

Use Case:

Used in **sharding** (horizontal scaling). Distributes documents **evenly** across shards.

How It Works:

- Stores a **hash** of the field's value (e.g., _id)
- Not useful for range queries (like $gt, $lt), but great for equality

Example:

```
db.users.createIndex({ _id: "hashed" })
```

4. Geospatial Indexes

Use Case:

Index geographic coordinates for apps like **maps, delivery services, location filters**

Supported Types:

- **2d**: for flat (planar) coordinates
- **2dsphere**: for spherical Earth-based coordinates (real-world GPS)

Example (2dsphere):

```
db.places.insertOne({
  name: "Pizza Point",
  location: {
    type: "Point",
    coordinates: [77.5946, 12.9716] // [longitude, latitude]
  }
})

// Create 2dsphere index
```

```
db.places.createIndex({ location: "2dsphere" })

// Find places near a point
db.places.find({
  location: {
    $near: {
      $geometry: {
        type: "Point",
        coordinates: [77.5946, 12.9716]
      },
      $maxDistance: 5000 // meters
    }
  }
})
```

Coordinates should be in [longitude, latitude] format.

5.Wildcard Index

Use Case:

Useful when field names are dynamic or when schema is not fixed (common in schema-less apps).

How It Works:

- Creates an index on **all fields** or fields that match a **wildcard path**

Example:

```
// Sample document with unknown structure
db.logs.insertOne({ meta: { user: "Amit", ip: "192.168.1.1" } })

// Wildcard index on all `meta` fields
db.logs.createIndex({ "meta.$**": 1 })

// Now queries on meta.user or meta.ip are fast
```

14.2 Index Management Utilities

View all indexes:

```
db.collection.getIndexes()
```

Remove a specific index:

```
db.collection.dropIndex("index_name")
```

Drop all:

```
db.collection.dropIndexes()
```

14.3 Summary

Index Type	Use Case	Notes
TTL	Auto-delete after time	Works only on Date fields
Text	Full-text search	1 text index per collection
Hashed	Sharding, uniform distribution	Not for range queries
Geospatial	Location-based filtering	Use 2dsphere for GPS
Wildcard	Unknown or dynamic fields	Great for flexible schemas

Unit 15: Aggregation Framework – Part 1

15.1 What is Aggregation?

Aggregation is the process of transforming data by processing multiple documents and returning computed results — similar to GROUP BY, SUM, AVG in SQL.

Aggregation Pipeline Basics

Aggregation works as a **pipeline** — documents pass through multiple **stages**, and each stage performs an operation.

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

Use the **aggregate()** method with a **pipeline**, which is an array of stages.

```
db.orders.aggregate([
  { $match: { status: "delivered" } },
  { $group: { _id: "$userId", totalSpent: { $sum: "$amount" } } }
])
```

Aggregation Stages

Category	Stage	Remarks
Filtering	$match	Used early to reduce input size and focus on relevant docs
Field shaping	$project, $addFields	Reshape data, compute new fields
Aggregation	$group	Group and accumulate values (sum, count, avg)
Array handling	$unwind	Needed before deeper aggregation when arrays are involved
Pagination & Sorting	$sort, $skip, $limit	Coming in Part 2
Analytics	$count, $bucket, $facet	Coming in Part 2
Joins	$lookup	Will be introduced after core aggregation
Advanced	$merge, $replaceRoot, etc.	Reserved for bonus/advanced learners

15.2 Aggregation Framework – Part 1

1. $match – Filter Documents

Purpose:

Filter input documents by conditions (like find() but used inside a pipeline).

Syntax:

```
{ $match: { field: value, status: "active" } }
```

Example:

```
db.users.aggregate([
  { $match: { age: { $gte: 25 } } }
])
```

Result: Only users aged 25 or above are passed to the next stage.

2. $project – Select or Reshape Fields

Purpose:

Include, exclude, or reshape fields.

Syntax:

```
{ $project: { name: 1, email: 1, password: 0 } }
```

1 to include, 0 to exclude

Example:

```
db.users.aggregate([
  { $project: { name: 1, age: 1, isAdult: { $gte: ["$age", 18] } } }
])
```

Adds a derived field isAdult.

3. $addFields – Add or Modify Fields

Purpose:

Adds or updates fields **without removing existing ones**.

Syntax:

```
{ $addFields: { fullName: { $concat: ["$firstName", " ", "$lastName"] } } }
```

Example:

```
db.products.aggregate([
  { $addFields: { discountedPrice: { $multiply: ["$price", 0.9] } } }
])
```

Adds a discountedPrice field keeping all others intact.

4. $group – Aggregate Documents

Purpose:

Group by _id and perform accumulations like $sum, $avg, $max, etc.

Syntax:

```
{ $group: { _id: "$category", total: { $sum: "$amount" } } }
```

Example:

```
db.orders.aggregate([
  { $group: { _id: "$userId", totalSpent: { $sum: "$amount" }, orders: { $sum: 1 } } }
])
```

Grouped by user, calculates total spending and number of orders.

5. $unwind – Flatten Array Fields

Purpose:

Converts an array field into multiple documents.

Syntax:

```
{ $unwind: "$items" }
```

Example:

```
db.orders.aggregate([
  { $unwind: "$items" },
  { $group: { _id: "$items.productId", totalQty: { $sum: "$items.quantity" } } }
])
```

One document per array element, ideal for order line-items.

15.3 Aggregation Framework – Part 2

1. $sort – Sorting Documents

Purpose:

Sort the output of documents in **ascending (1)** or **descending (-1)** order.

Syntax:

```
{ $sort: { fieldName: 1 or -1 } }
```

Example:

```
db.orders.aggregate([
  { $sort: { amount: -1 } } // highest to lowest
])
```

Use this before $limit for top-N queries

2. $limit – Limit Results

Purpose:

Limits number of documents in the pipeline result.

Syntax:

```
{ $limit: 5 }
```

Example:

```
db.orders.aggregate([
  { $match: { status: "delivered" } },
  { $sort: { amount: -1 } },
  { $limit: 3 }
])
```

Get top 3 highest-value delivered orders

3. $skip – Skip Documents

Purpose:

Skip a number of documents — useful for pagination.

Syntax:

```
{ $skip: 10 }
```

Example:

```
// For page 3, with 5 per page:
db.orders.aggregate([
  { $sort: { createdAt: -1 } },
  { $skip: 10 },
  { $limit: 5 }
])
```

Implement skip-limit based pagination

4. $count – Count Documents

Purpose:

Count documents **after filtering**.

Syntax:

```
{ $count: "fieldName" }
```

Example:

```
db.orders.aggregate([
  { $match: { status: "cancelled" } },
  { $count: "cancelledOrders" }
])
```

Returns a single document with the count

5. $bucket – Group Into Ranges

Purpose:

Group values into **user-defined ranges** (like marks into grades).

Syntax:

```
{
  $bucket: {
    groupBy: "$amount",
    boundaries: [0, 500, 1000, 1500],
    default: "Others",
    output: {
      orderCount: { $sum: 1 },
      orders: { $push: "$_id" }
    }
  }
}
```

Use for creating dynamic price slabs, score ranges, etc.

6. $facet – Multiple Pipelines in Parallel

Purpose:

Run multiple aggregations in parallel and combine results into one document.

Example:

```
db.orders.aggregate([
  {
    $facet: {
      "Top Orders": [
        { $sort: { amount: -1 } },
        { $limit: 2 }
      ],
      "Status Breakdown": [
        { $group: { _id: "$status", count: { $sum: 1 } } }
      ]
    }
  }
])
```

Great for dashboard-style queries (summary + details together)

7. $lookup – Joins Across Collections

Purpose:

Join documents from another collection (like JOIN in SQL).

Example:

Join orders with users collection on userId.

```
db.orders.aggregate([
  {
    $lookup: {
      from: "users",
      localField: "userId",
      foreignField: "_id",
      as: "userDetails"
    }
  }
])
```

Result will have a userDetails array containing the matched user(s)

15.4 Practice Problems

Database: Banking Database, the collection as follows:

1. **branches**
2. **customers**
3. **accounts**
4. **transactions**
5. **loans**

1. branches

```
[
 { "_id": 1, "branch_name": "Chennai Main" },
 { "_id": 2, "branch_name": "Bangalore Central" },
 { "_id": 3, "branch_name": "Mumbai West" }
]
```

2. customers

```
[
 { "_id": 101, "name": "Aarav", "email": "aarav@example.com", "phone": "9876543210", "branch_id": 1 },
 { "_id": 102, "name": "Meera", "email": "meera@example.com", "phone": "9123456780", "branch_id": 1 },
 { "_id": 103, "name": "Vikram", "email": "vikram@example.com", "phone": "9998887777", "branch_id": 2 },
 { "_id": 104, "name": "Riya", "email": "riya@example.com", "phone": "7776665555", "branch_id": 2 },
 { "_id": 105, "name": "Kabir", "email": "kabir@example.com", "phone": "8887776666", "branch_id": 3 }
]
```

3. accounts

```
[
 { "_id": "A1", "customer_id": 101, "branch_id": 1, "account_type": "Savings", "balance": 35000 },
 { "_id": "A2", "customer_id": 102, "branch_id": 1, "account_type": "Current", "balance": 50000 },
 { "_id": "A3", "customer_id": 103, "branch_id": 2, "account_type": "Savings", "balance": 42000 },
 { "_id": "A4", "customer_id": 104, "branch_id": 2, "account_type": "Current", "balance": 62000 },
 { "_id": "A5", "customer_id": 105, "branch_id": 3, "account_type": "Savings", "balance": 70000 }
]
```

4. transactions

```
[
 { "_id": "T1", "account_id": "A1", "amount": 5000, "transaction_type": "debit", "timestamp": ISODate("2023-01-05T10:00:00Z") },
 { "_id": "T2", "account_id": "A2", "amount": 15000, "transaction_type": "credit", "timestamp": ISODate("2023-01-20T12:30:00Z") },
 { "_id": "T3", "account_id": "A3", "amount": 2000, "transaction_type": "debit", "timestamp": ISODate("2023-01-25T09:00:00Z") },
 { "_id": "T4", "account_id": "A4", "amount": 8000, "transaction_type": "credit", "timestamp": ISODate("2023-02-02T14:00:00Z") },
```

{ "_id": "T5", "account_id": "A5", "amount": 9000, "transaction_type": "debit", "timestamp": ISODate("2023-01-10T08:00:00Z") }

]

5. loans

[

{ "_id": "L1", "customer_id": 101, "amount": 100000, "loan_type": "Personal" },

{ "_id": "L2", "customer_id": 102, "amount": 150000, "loan_type": "Home" },

{ "_id": "L3", "customer_id": 103, "amount": 120000, "loan_type": "Car" },

{ "_id": "L4", "customer_id": 104, "amount": 90000, "loan_type": "Personal" }

]

Problem Statements:

1. Find the total number of accounts in each branch

```
db.accounts.aggregate([
 {
  $group: {
   _id: "$branch_id",  // Group by branch_id
   total_accounts: { $sum: 1 }  // Count accounts
  }
 },
 {
  $lookup: {
   from: "branches",  // Join with branches collection to get branch details
   localField: "_id",  // Match branch_id from accounts
   foreignField: "_id",  // Match _id from branches
   as: "branch_details"
  }
 },
 {
  $unwind: "$branch_details"  // Flatten the array from lookup
 },
 {
  $project: {
   branch_name: "$branch_details.branch_name",
   total_accounts: 1  // Only include total_accounts and branch_name in the result
  }
 }
])
```

2. Find the average balance of all accounts per branch

```
db.accounts.aggregate([
 {
  $group: {
   _id: "$branch_id",  // Group by branch_id
   average_balance: { $avg: "$balance" }  // Calculate average balance
  }
 },
 {
  $lookup: {
   from: "branches",  // Join with branches collection to get branch details
   localField: "_id",
   foreignField: "_id",
   as: "branch_details"
  }
 },
 {
  $unwind: "$branch_details"
 },
 {
  $project: {
   branch_name: "$branch_details.branch_name",
   average_balance: 1
  }
 }
])
```

3. Find the total transaction amount for each account

```
db.transactions.aggregate([
 {
  $group: {
   _id: "$account_id",  // Group by account_id
   total_transaction_amount: { $sum: "$amount" }  // Sum of all transactions per account
  }
 },
 {
  $lookup: {
   from: "accounts",  // Join with accounts to get more account details
   localField: "_id",
   foreignField: "_id",
   as: "account_details"
  }
 },
```

```
 {
  $unwind: "$account_details"
 },
 {
  $project: {
   account_type: "$account_details.account_type",
   total_transaction_amount: 1
  }
 }
])
```

4. Find the total loan amount for each customer

```
db.loans.aggregate([
 {
  $group: {
   _id: "$customer_id",  // Group by customer_id
   total_loan_amount: { $sum: "$amount" }  // Sum of all loan amounts per customer
  }
 },
 {
  $lookup: {
   from: "customers",  // Join with customers collection to get customer details
   localField: "_id",
   foreignField: "_id",
   as: "customer_details"
  }
 },
 {
  $unwind: "$customer_details"
 },
 {
  $project: {
   customer_name: "$customer_details.name",
   total_loan_amount: 1
  }
 }
])
```

5. Find all transactions within a specific date range

```
db.transactions.aggregate([
 {
  $match: {
   timestamp: {
    $gte: ISODate("2023-01-01T00:00:00Z"),
```

```
      $lte: ISODate("2023-02-01T23:59:59Z")
    } // Filter transactions by date range
  }
 },
 {
  $lookup: {
    from: "accounts",  // Join with accounts collection
    localField: "account_id",
    foreignField: "_id",
    as: "account_details"
  }
 },
 {
  $unwind: "$account_details"
 },
 {
  $project: {
    account_type: "$account_details.account_type",
    amount: 1,
    transaction_type: 1,
    timestamp: 1
  }
 }
])
```

6. Find the total number of customers per branch

```
db.customers.aggregate([
 {
  $group: {
    _id: "$branch_id",  // Group by branch_id
    total_customers: { $sum: 1 }  // Count customers
  }
 },
 {
  $lookup: {
    from: "branches",  // Join with branches collection
    localField: "_id",
    foreignField: "_id",
    as: "branch_details"
  }
 },
 {
  $unwind: "$branch_details"
```

```
  },
  {
   $project: {
    branch_name: "$branch_details.branch_name",
    total_customers: 1
   }
  }
])
```

7. Find the highest balance among all accounts

```
db.accounts.aggregate([
  {
   $sort: { balance: -1 }  // Sort accounts by balance in descending order
  },
  {
   $limit: 1  // Limit the result to the top account
  },
  {
   $lookup: {
    from: "customers",  // Join with customers collection to get customer details
    localField: "customer_id",
    foreignField: "_id",
    as: "customer_details"
   }
  },
  {
   $unwind: "$customer_details"
  },
  {
   $project: {
    customer_name: "$customer_details.name",
    balance: 1
   }
  }
])
```

8. Customer with the highest balance per branch

```
db.accounts.aggregate([
  {
   $sort: { balance: -1 }  // Sort accounts by balance in descending order
  },
  {
   $group: {
```

```
    _id: "$branch_id",  // Group by branch_id
    highest_balance_account: { $first: "$$ROOT" }  // Select the account with the highest balance in each
group
   }
  },
  {
   $lookup: {
    from: "customers",  // Join with customers collection to get customer details
    localField: "highest_balance_account.customer_id",
    foreignField: "_id",
    as: "customer_details"
   }
  },
  {
   $unwind: "$customer_details"  // Flatten the customer details array
  },
  {
   $lookup: {
    from: "branches",  // Join with branches collection to get branch details
    localField: "_id",
    foreignField: "_id",
    as: "branch_details"
   }
  },
  {
   $unwind: "$branch_details"
  },
  {
   $project: {
    branch_name: "$branch_details.branch_name",  // Show the branch name
    customer_name: "$customer_details.name",    // Show the customer name
    email: "$customer_details.email",          // Show the customer's email
    phone: "$customer_details.phone",          // Show the customer's phone number
    highest_balance: "$highest_balance_account.balance"  // Show the highest balance
   }
  }
])
```

15.5 Summary

Stage	Use Case
$sort	Ordering results
$limit	Top-N queries
$skip	Pagination
$count	Total after filter

$bucket	Range grouping
$facet	Dashboard-style multi-stats
$lookup	Join across collections

Unit 16: Transactions & ACID in MongoDB

16.1 What is a Transaction?

A **transaction** in MongoDB allows multiple read/write operations to be executed as **a single atomic unit** — either **all succeed or none do**.

Think of it like transferring money between accounts — you **withdraw** from one and **deposit** into another. Both must happen, or neither should.

16.2 What is ACID?

ACID is a set of **guarantees** that database transactions must provide to ensure reliability:

Property	Meaning
A - Atomicity	All operations succeed or none are applied
C - Consistency	Transforms the DB from one valid state to another
I - Isolation	Concurrent transactions don't interfere with each other
D - Durability	Once committed, changes remain even in case of system failure

16.3 How to Use Transactions in MongoDB (multi-document)

MongoDB supports **multi-document transactions** in:

- **Replica sets** (since MongoDB 4.0)
- **Sharded clusters** (since MongoDB 4.2)

Use Cases:

- Money transfers between accounts
- Order creation + inventory update
- Loan approval + customer profile update

Basic Transaction Example

Use this pattern for session-based transactions in Node.js (Mongoose/MongoDB Native):

Mongo Shell Example (mongosh):

```
const session = db.getMongo().startSession();

session.startTransaction();

try {
  const accountsColl = session.getDatabase("bank").accounts;
  const transactionsColl = session.getDatabase("bank").transactions;

  // 1. Debit sender
  accountsColl.updateOne(
    { _id: "A1" },
    { $inc: { balance: -5000 } }
  );

  // 2. Credit receiver
  accountsColl.updateOne(
    { _id: "A2" },
    { $inc: { balance: 5000 } }
  );

  // 3. Log transaction
  transactionsColl.insertOne({
    from: "A1",
    to: "A2",
    amount: 5000,
    date: new Date()
  });

  session.commitTransaction();
  print("Transaction committed!");
} catch (error) {
  session.abortTransaction();
  print("❌ Transaction aborted due to error:", error);
} finally {
  session.endSession();
}
```

Key Notes:

- Use **session.startTransaction()** and **session.commitTransaction()**.
- Any failure during the transaction must call **session.abortTransaction()**.
- All collections used **must be in the same replica set or sharded environment**.

16.4 Limitations

Limitation	Details
Performance	Transactions are slower than single ops — use wisely
Writes only	Cannot mix read & write in some drivers (check driver versions)
Memory cost	Large transactions consume more RAM/locks
Avoid where possible	Prefer atomic single-doc updates if schema allows

16.5 Summary Table

Concept	MongoDB Feature
Atomicity	Transactions & Write Concerns
Isolation	Snapshot isolation by default
Consistency	Ensured through schema and constraints
Durability	Journaling + Replication