Day 1 – MongoDB Foundations & Core Administration

1. Introduction to MongoDB Ecosystem (1 hr)

- What is MongoDB?
- Document Database vs Relational Databases
- BSON & JSON structure
- Collections, Databases, and Clusters
- Where MongoDB fits in modern applications

---

2. MongoDB and the Document Model (1 hr)

- Embedded vs Referenced Documents
- Flexible Schema
- Sample document structures
- Lab: Explore sample documents using MongoDB Shell

---

3. MongoDB Data Modelling Introduction (1 hr)

- Data modelling patterns
- One-to-One, One-to-Many, Many-to-Many
- Polymorphic patterns
- Lab: Model a simple E-commerce schema

---

4. The MongoDB Shell (mongosh) (1 hr)

- Basic shell operations
- Connecting to standalone, replica set
- Shell queries and scripting basics
- Lab: Connect and explore collections

---

5. Connecting to a MongoDB Database (30 min)

- Connection string concepts
- Users, roles, authentication
- Connect using mongosh, Compass, and drivers

---

6. MongoDB CRUD Operations – Insert & Find (1 hr)

- insertOne(), insertMany()
- find(), findOne()
- Query filters, projections
- Lab: Perform CRUD queries on a sample dataset

---

7. MongoDB CRUD Operations – Replace & Delete (30 min)

- replaceOne()
- deleteOne(), deleteMany()
- Lab: Data cleanup operations

---

8. Modifying Query Results (1 hr)

- sort(), skip(), limit()
- Aggregation preview
- Query optimization basics
- Lab: Apply query modifiers for performance

End of Day Assignment

- CRUD tasks on sample collections
- Short quiz (MCQ + practical)

1. Introduction to MongoDB Ecosystem (1 hr)

- What is MongoDB?
- Document Database vs Relational Databases
- BSON & JSON structure
- Collections, Databases, and Clusters
- Where MongoDB fits in modern applications

1. Introduction to MongoDB Ecosystem (1 hr)

---

1. INTRODUCTION TO MONGODB ECOSYSTEM

Audience: CSC Global – Bangalore (Tech, Data, Security, Compliance Teams)

A. What is MongoDB? (15 minutes)

Definition

MongoDB is a **NoSQL, document-oriented database** designed for:

- Scalability
- Flexibility
- High availability
- Fast development cycles

Key Characteristics

Feature	Description
NoSQL	Non-relational, schema-flexible
Document Store	Stores data as JSON/BSON
Distributed	Supports sharding & replication
High-Performance	Optimized for read/write workloads
Developer Friendly	Quick prototyping using JSON

Relevance to CSC Global's Business Domains

MongoDB is highly suited for environments involving:

- **Corporate legal entity management**
- **Records, compliance, and filings tracking**
- **Trademark, domain & digital brand protection**
- **Audit logs and large document repositories**
- **Multi-jurisdictional data models**

MongoDB excels where:

- Data structures vary across regions
- Regulatory requirements differ country-to-country
- History, audit, and metadata evolve frequently
- Documents contain hierarchical and nested structures

Why MongoDB for CSC Global

- **Flexible schema** → Different countries have different legal entity formats
- **High ingestion rate** → Bulk corporate filings and compliance data
- **Nested document support** → Corporate hierarchies, ownership chains

- **Scalable workload** → Global client base with large data volumes
- **Audit logging & versioning** → Ideal for compliance lifecycle

---

B. Document Database vs Relational Databases (15 minutes)

Architectural Differences

Relational Database	MongoDB Document Database
Fixed schema	Schema flexible
Row-based storage	Document-based
Normalization	Embedding for performance
Joins required	Mostly joinless
Rigid migrations	Dynamic schema changes
Optimal for structured, transactional workloads	Optimal for semi-structured, hierarchical data

---

When RDBMS is preferred

- Complex joins
- Strict ACID transactional systems (e.g., banking)
- Fixed schema regulatory systems

When MongoDB is preferred

- Rapidly changing data models
- Large-scale document repositories
- Hierarchical, nested, or JSON-based data
- Global applications with scale-out architecture

CSC Example Comparison

Use Case	Why Relational DB?	Why MongoDB?
Employee payroll	Stable schema	Not needed
Entity management	High variation across jurisdictions	✓ MongoDB
Legal document storage	Nested metadata, attachments	✓ MongoDB
Domain/Trademark portfolio	Hierarchical relationships	✓ MongoDB

———————————————

C. BSON & JSON Structure (10 minutes)

JSON (JavaScript Object Notation)

- Human readable
- Limited data types

BSON (Binary JSON)

- MongoDB's internal storage format
- Supports rich types:
  - ObjectId
  - Date
  - Decimal128
  - Binary
  - Arrays
  - Embedded documents

Example JSON Document (CSC Legal Entity)

```
{
  "entityName": "CSC Global Holdings LLC",
  "jurisdiction": "Delaware",
  "status": "Active",
  "officers": [
    { "name": "John Doe", "role": "Director" },
    { "name": "Sarah Lee", "role": "Secretary" }
  ],
  "filings": {
    "annualReport": true,
    "lastFiled": "2024-09-15"
  }
}
```

Why BSON Helps CSC

- Date type ensures accurate compliance deadlines
- Nested fields reflect officer roles easily
- Array support suits multi-director structures
- ObjectId ensures globally unique identification

D. Collections, Databases & Clusters (10 minutes)

Database

A logical group of collections.

E.g.,

- entity_management
- legal_compliance
- digital_brand_protection

Collection

Stores **documents** related to one business entity type.

Examples for CSC:

- entities
- officers
- trademarks
- domainPortfolio
- auditLogs

Cluster

A deployment of MongoDB servers:

- Standalone
- Replica Set (HA)
- Sharded Cluster (scaling)

For global companies like CSC, sharded clusters enable:

- Distributed workloads across APAC, EMEA, US
- Localized low-latency access
- Geo-redundancy for compliance

E. Where MongoDB Fits in Modern Applications (10 minutes)

Ideal Use Cases

1. **Microservices Architectures**
2. **High-ingestion data systems**
3. **Analytics & search-driven applications**

4. **Compliance and regulatory logging**
5. **Document workflows**
6. **API-first systems**

CSC Global Application Examples

1. Legal Entity Management Platform

MongoDB stores:

- Entity profiles
- Ownership chains
- Filing histories

2. Trademark/Digital Brand Portfolio Management

Stores:

- Trademark metadata
- Renewal cycles
- Jurisdiction rules

3. Compliance Audit System

MongoDB handles:

- Change logs
- User access logs
- Versioning of legal records

4. Domain Registration & DNS Metadata

MongoDB supports:

- Nested DNS structure
- WHOIS metadata
- Global registry details

Summary for Section 1

After completing this module, CSC participants will understand:

- What MongoDB is
- Why it’s chosen for global business/legal applications
- Differences from relational databases
- BSON/JSON document structure
- Collections, databases, clusters
- MongoDB’s role in modern distributed systems

---

2. MongoDB and the Document Model (1 hr)

- Embedded vs Referenced Documents
- Flexible Schema
- Sample document structures
- Lab: Explore sample documents using MongoDB Shell

A. Embedded vs Referenced Documents (20 minutes)

MongoDB supports two major approaches to modeling relationships:

1. Embedded Documents (Denormalization)

Data related to an entity is stored *inside* the same parent document.

✔ **Best For:**

- One-to-few relationships
- High read performance
- Data that is always used together

📌 CSC Example – Legal Entity with Officers (Embedded)

```
{
  "entityName": "CSC Global Holdings LLC",
  "jurisdiction": "Delaware",
  "status": "Active",
  "officers": [
    { "name": "John Doe", "role": "Director" },
    { "name": "Sarah Lee", "role": "Secretary" }
  ],
  "compliance": {
    "lastFiled": "2024-09-15",
    "nextDue": "2025-09-15"
  }
}
```

✔ **CSC Benefits:**

- Quick retrieval of full entity profile
- Officers are typically few → natural embedding
- Reduces query complexity (no joins)

---

2. Referenced Documents (Normalization)

Store relationships using references (similar to foreign keys).

✔ **Best For:**

- Large or growing related datasets
- Many-to-many relationships
- Data updated independently
- Compliance record history (thousands of entries)

📌 CSC Example – Entity referencing Audit Logs

Entity Document

```
{
  "_id": ObjectId("66a5f3a8c9"),
  "entityName": "CSC Digital Services India Pvt Ltd",
  "jurisdiction": "Karnataka",
  "auditLogs": [
    ObjectId("99ac8b1e31"),
    ObjectId("99ac8b1e32")
  ]
}
```

Audit Log Document

```
{
  "_id": ObjectId("99ac8b1e31"),
  "action": "Update Officer",
  "user": "surendra.p",
  "timestamp": ISODate("2025-01-12T10:15:00Z")
}
```

CSC Benefits:

- Audit logs can grow into thousands → referencing required
- Each log entry updated independently
- Improves write performance

When to Embed vs Reference? (Cheat Sheet)

Requirement	Use Embedded	Use Reference
Small related set	✓	
Large or unbounded set		✓
Always read together	✓	
Access independently		✓
Fast reads	✓	
Avoid duplication		✓

B. Flexible Schema (15 minutes)

MongoDB offers schema flexibility: documents in the same collection **may** have different fields.

This is ideal for **CSC Global**, where regulatory and filing rules vary across:

- Countries
- States
- Registries
- Business verticals

✓ **Example: Dynamic Compliance Requirements**

US-Based Entity

```
{
  "entityName": "CSC Compliance LLC",
  "type": "LLC",
  "annualReport": true,
  "stateTaxId": "TX-99821"
}
```

EU-Based Entity

```
{
  "entityName": "CSC Europe BV",
  "type": "BV",
  "vatNumber": "NL123456",
  "chamberOfCommerceId": "KFK-98721"
}
```

Benefits for CSC:

- No schema migration required when new attributes added
- Supports region-specific compliance fields
- Faster adaptation to regulatory changes

C. Sample Document Structures (10 minutes)

Below are real-world structures relevant to CSC’s legal & compliance domain.

---

1. Corporate Entity Document

```
{
  "entityId": "CSC-DEL-2025-001",
  "entityName": "CSC Corporate Solutions Pvt Ltd",
  "jurisdiction": "Delhi",
  "officers": [
    { "name": "Vikram Rao", "role": "Director", "since":
"2022" }
  ],
  "filings": {
    "annualReport": true,
    "lastFiled": "2024-07-15"
  }
}
```

---

2. Trademark Portfolio Document

```
{
  "trademark": "CSC GLOBAL",
  "classes": [9, 35, 42],
  "countries": ["US", "UK", "IN"],
  "status": "Registered",
  "renewalDue": "2026-11-20"
}
```

---

3. Digital Brand Asset Document

```
{
  "domain": "cscglobal.com",
  "dns": {
    "A": "104.16.44.1",
    "CNAME": "cscglobal.com.edge.net"
  },
  "sslExpiry": "2025-08-01"
}
```

---

3. MongoDB Data Modelling Introduction (1 hr)

- Data modelling patterns
- One-to-One, One-to-Many, Many-to-Many
- Polymorphic patterns
- Lab: Model a simple E-commerce schema

MongoDB's flexible schema provides freedom—but modeling still requires **structure, intent, and performance-driven design**.

This module teaches how to design **efficient, scalable, maintainable** document structures.

A. Data Modelling Patterns (20 minutes)

MongoDB supports several modelling approaches depending on:

- Access patterns
- Read/write frequency
- Data size
- Cardinality (1:1, 1:many, many:many)
- Growth expectations
- Performance needs

---

1. Embedding Pattern (Denormalization)

Used when related data is frequently read together.

✓ Strengths

- Single document lookup
- High read performance
- Simpler application logic

✓ Weakness

- Document growth risk
- Duplicate data across documents

---

2. Referencing Pattern (Normalization)

Used when related data is large or independently accessed.

✓ Strengths

- Smaller documents
- Independent updates
- Better scalability

✓ Weakness

- Requires $lookup or multiple queries

---

3. Extended Reference Pattern

Document stores reference *with additional metadata*.

Example:

```
{
  "productId": "P123",
  "vendor": {
    "vendorId": "V1001",
    "vendorName": "CSC Digital Brands"
  }
}
```

Used for performance: avoid join when metadata is frequently accessed.

---

4. Attribute Pattern

Flatten nested keys into searchable fields.

Example

```
{
  "attribute_color": "red",
  "attribute_size": "L"
}
```

Improves indexing & searching on flexible fields.

---

5. Bucket Pattern

Group time-series or event data into buckets.

Example: order logs grouped into one document per day.

---

6. Outlier Pattern

Move large subdocuments into separate documents to prevent exceeding 16MB BSON limit.

B. Relationship Types (1:1, 1:Many, Many:Many) (15 minutes)

Understanding cardinality is essential for MongoDB schema design.

---

1. One-to-One Relationship

✔ Use Embedding when:

- Data is frequently accessed together
- Size is small

Example: User Profile

```
{
  "_id": "U001",
  "name": "Alice",
  "profile": {
    "email": "alice@example.com",
    "phone": "99887766"
  }
}
```

✔ Use Referencing when:

- Sensitive information stored separately
- Security or access control differs

Example:

```
{
  "_id": "U001",
  "name": "Alice",
  "profileId": "PR001"
}
```

---

2. One-to-Many Relationship

Common pattern in:

- Orders → Order Items
- Entity → Officers
- Categories → Products

✓ Embed when:

- "Many" side is small
- Accessed together

Example: Order with Invoice Items

```
{
  "_id": "ORD001",
  "customerName": "Rahul",
  "items": [
    { "product": "Laptop", "qty": 1 },
    { "product": "Mouse", "qty": 2 }
  ]
}
```

✓ Reference when:

- "Many" side is huge or unbounded
- Separate updates needed

Example: Product Reviews

```
{
  "_id": "PR001",
  "name": "Laptop"
}
```

```
{
  "productId": "PR001",
  "rating": 5,
  "comment": "Great!"
}
```

---

3. Many-to-Many Relationship

Examples:

- Products <-> Categories
- Students <-> Courses
- Users <-> User Groups

MongoDB options:

A. Array of References

```
{
  "product": "Laptop",
  "categories": ["Electronics", "Office"]
}
```

B. Linking Collection (recommended when data is huge)

```
{
  "productId": "P123",
  "categoryId": "C500"
}
```

C. Polymorphic Patterns (10 minutes)

MongoDB supports documents with multiple “types” using the **same collection**.

Example: Multiple types of products in one collection

Document Type: “Electronics”

```
{
  "type": "electronics",
  "brand": "Dell",
  "warranty": 24
}
```

Document Type: “Clothing”

```
{
  "type": "clothing",
  "brand": "Nike",
  "size": "L",
  "material": "Cotton"
}
```

Benefits:

- Handles diverse product metadata
- Simplifies querying
- Flexible schema reduces migration effort

✓ Use Cases in CSC Context

Polymorphism applies to:

- Different legal entity types (LLC, LLP, Corp, BV…)
- Trademark classes and variations
- Domain asset types (DNS, SSL, WHOIS…)

MongoDB lets all types coexist in one collection.

---

4. The MongoDB Shell (mongosh) (1 hr)

- Basic shell operations
- Connecting to standalone, replica set
- Shell queries and scripting basics
- Lab: Connect and explore collections

The MongoDB Shell (**mongosh**) is the primary interactive tool for DBAs and developers to connect, query, administer, and inspect MongoDB deployments.

A. Basic Shell Operations (20 minutes)

What is mongosh?

A JavaScript-powered shell used to:

- Connect to MongoDB deployments
- Execute CRUD operations
- Manage users, roles, indexes
- Run administrative commands
- Inspect performance and cluster metadata

---

1. Start the Shell

To start the shell (assuming default local installation):

```
mongosh
```

---

2. Show Databases

```
show dbs
```

Displays all databases with size > 0.

---

3. Create or Switch Database

```
use mydb
```

MongoDB creates the DB only after inserting the first document.

---

4. Show Collections

```
show collections
```

---

5. Insert a Document

```
db.users.insertOne({ name: "Amit", city: "Pune" });
```

---

6. Find Documents

```
db.users.find();
```

Pretty format:

```
db.users.find().pretty();
```

---

7. Update and Delete

```
db.users.updateOne({ name: "Amit" }, { $set: { city:
"Bangalore" } });
db.users.deleteOne({ name: "Amit" });
```

---

8. Check Shell Help

```
help
db.help()
```

B. Connecting to Standalone & Replica Set (15 minutes)

1. Connect to a Standalone MongoDB Server

```
mongosh "mongodb://localhost:27017"
```

If username/password exists:

```
mongosh "mongodb://adminUser:P@ssw0rd@localhost:27017/admin"
```

---

2. Connect to a Replica Set

Replica set connection format:

```
mongodb://host1:27017,host2:27018,host3:27019/?replicaSet=rs0
```

Example:

```
mongosh
"mongodb://localhost:27017,localhost:27018,localhost:27019/?re
plicaSet=gktcsrs"
```

Check Replica Set Status

```
rs.status();
```

Check Primary/Secondary Node

```
rs.isMaster();
```

or in newer versions:

```
db.hello();
```

---

3. Connecting to MongoDB Atlas

```
mongosh
"mongodb+srv://user:password@cluster0.abcd.mongodb.net/myDB"
```

C. Shell Queries & Scripting Basics (15 minutes)

mongosh supports **JavaScript**, making it powerful for automation.

---

1. Running JavaScript Variables

```
let username = "Rahul";
db.users.find({ name: username });
```

---

2. Loops for Data Generation

```
for (let i = 1; i <= 10; i++) {
  db.logs.insertOne({ logId: i, ts: new Date() });
}
```

3. Writing Simple Functions

```
function findByCity(city) {
  return db.users.find({ city: city }).pretty();
}

findByCity("Pune");
```

4. Running External JavaScript File

Useful for DBA automation tasks.

Create a file: script.js

```
db.customers.insertOne({ name: "CSC Global", location:
"Bangalore" });
```

Run it:

```
mongosh script.js
```

5. Inspecting Server & DB Metadata

List DB Stats:

```
db.stats();
```

Collection Stats:

```
db.users.stats();
```

Server Build Info:

```
db.version();

db.serverStatus();
```

5. Connecting to a MongoDB Database (30 min)

- Connection string concepts
- Users, roles, authentication
- Connect using mongosh, Compass, and drivers

Connecting to MongoDB successfully and securely is a critical skill for DBAs and developers. This module covers connection strings, authentication models, and connection via tools.

A. Connection String Concepts (10 minutes)

A **MongoDB connection string** (URI) tells the MongoDB client how to connect to a cluster.

---

1. Basic Format

```
mongodb://host:port
```

Example:

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

✓

2. With Username + Password

```
mongodb://username:password@host:port/database
```

Example:

```
mongodb://adminUser:AdminPass123@localhost:27017/admin
```

---

3. Replica Set Connection String

Use multiple hosts + replicaSet parameter.

```
mongodb://host1:27017,host2:27018,host3:27019/?replicaSet=gktcsrs
```

✓

4. MongoDB Atlas URI (SRV format)

```
mongodb+srv://user:password@cluster.abcd.mongodb.net/mydb
```

SRV records simplify:

- Load balancing
- Node discovery
- SSL configuration

---

5. Common Connection String Parameters

Parameter	Purpose
replicaSet	Name of replica set
authSource	DB where credentials are stored
retryWrites	Auto retry write operations
tls=true	Enable encryption
connectTimeoutMS	Timeout for initial connection

Example with parameters:

```
mongodb://user:pass@localhost:27017/mydb?authSource=admin&retr
yWrites=true&tls=false
```

✓ CSC Security Requirements (Best Practices)

- Use **strong passwords** & **SCRAM-SHA-256**
- Use **TLS/SSL** for production
- Use **internal DNS hostnames** (not IPs)
- Always specify **authSource=admin** for superuser accounts
- Use **application-specific users** (not admin user)

B. Users, Roles, and Authentication (10 minutes)

MongoDB uses **Role-Based Access Control (RBAC)**.

Authentication mechanisms

- SCRAM-SHA-1
- SCRAM-SHA-256 (recommended)
- LDAP (enterprise)
- Kerberos

1. Create an Admin User

Run in the **admin** database:

```
use admin;

db.createUser({
  user: "cscAdmin",
  pwd: "StrongPass#2025",
  roles: [ "root" ]
});
```

2. Create Application User

Used for applications or microservices.

```
use ecommerce;

db.createUser({
  user: "appUser",
  pwd: "AppPass123",
  roles: [
    { role: "readWrite", db: "ecommerce" }
  ]
});
```

3. Common Built-In Roles

Role	Permissions
read	Read-only access
readWrite	Read + Write
dbAdmin	Create indexes, view stats
clusterAdmin	Replica set & sharding ops
root	Full access

4. Check Current User & Roles

```
db.runCommand({ connectionStatus: 1 })
```

✓ CSC Guideline for User Access**

- Admin accounts → Only DBAs
- readWrite → Apps & services
- read-only → Audit, reporting, compliance teams
- clusterAdmin → Infra team

C. Connecting Using mongosh, Compass, and Drivers (10 minutes)

MongoDB can be accessed via:

- **mongosh (CLI)**
- **Compass (GUI)**
- **Application drivers** (Node.js, Python, Java, Go…)

1. Connect Using mongosh

✓ Connect to Local DB

```
mongosh "mongodb://localhost:27017"
```

✓ Connect with Authentication

```
mongosh
"mongodb://cscAdmin:StrongPass#2025@localhost:27017/admin"
```

✓ Connect to Replica Set

```
mongosh
"mongodb://node1:27017,node2:27018,node3:27019/?replicaSet=csc
RS"
```

✓ Connect to Atlas

```
mongosh
"mongodb+srv://cscAdmin:StrongPass#2025@cluster0.mongodb.net/m
ydb"
```

---

2. Connect Using MongoDB Compass (GUI)

Steps:

1. Open Compass
2. Click **"New Connection"**
3. Paste connection string

4. Click **“Connect”**

✔ Example (Standalone)

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

✔ Example (Auth)

```
mongodb://appUser:AppPass123@localhost:27017/ecommerce?authSource=ecommerce
```

✔ Atlas Example

```
mongodb+srv://cscAdmin:<password>@cluster0.mongodb.net/
```

Compass automatically:

- Detects TLS
- Displays cluster topology
- Shows query performance

---

3. Connect using Drivers (Node.js, Python, Java)

✔ Node.js Example

```
const { MongoClient } = require("mongodb");
const uri =
"mongodb://appUser:AppPass123@localhost:27017/ecommerce";

const client = new MongoClient(uri);

async function run() {
  await client.connect();
  console.log("Connected to MongoDB");
}
run();
```

✔ Python Example

```
from pymongo import MongoClient

client =
MongoClient("mongodb://appUser:AppPass123@localhost:27017/ecommerce")
db = client.ecommerce
print(db.list_collection_names())
```

✓ Java Example

```
MongoClient client =
MongoClients.create("mongodb://appUser:AppPass123@localhost:27
017/ecommerce");
MongoDatabase db = client.getDatabase("ecommerce");
```

CSC Guidance for Applications

- Use app-specific users, not admin
- Store credentials in Vault / AWS Secret Manager
- Always enable retryWrites
- Use TLS=true in production

MODULE 6: MongoDB CRUD Operations – Insert & Find (1 hr)

CRUD = **Create, Read, Update, Delete**

This module focuses on the **Create (Insert)** and **Read (Find)** operations.

A. insertOne() and insertMany() (20 minutes)

1. insertOne()

Adds a **single document** to a collection.

Syntax:

```
db.collection.insertOne(document)
```

Example 1 – Insert a user document

```
db.users.insertOne({
  name: "Ravi Kumar",
  email: "ravi.kumar@example.com",
  city: "Bangalore"
});
```

Example 2 – Insert a CSC-style Legal Entity

```
db.entities.insertOne({
  entityName: "CSC India Pvt Ltd",
  jurisdiction: "Karnataka",
  active: true
});
```

---

2. insertMany()

Adds **multiple documents** at once.

Syntax:

```
db.collection.insertMany([doc1, doc2, doc3])
```

Example – Insert multiple products

```
db.products.insertMany([
  { name: "Laptop", category: "Electronics", price: 75000 },
  { name: "Mouse", category: "Electronics", price: 1200 },
  { name: "Notebook", category: "Stationery", price: 50 }
]);
```

Auto-generated _id

If not specified, MongoDB generates _id: ObjectId()

B. find() and findOne() (20 minutes)

Reading data is the most frequent operation in MongoDB.

---

1. find()

Returns a **cursor** containing all matching documents.

Syntax:

```
db.collection.find(query, projection)
```

Example – Find all documents

```
db.products.find();
```

Example – Find documents with condition

```
db.products.find({ category: "Electronics" });
```

---

2. findOne()

Returns **first matching document**.

Syntax:

```
db.collection.findOne(query)
```

Example

```
db.users.findOne({ city: "Bangalore" });
```

C. Query Filters & Projection (15 minutes)

MongoDB queries use **JSON-based filters**.

---

1. Equality Filter

```
db.products.find({ price: 1200 });
```

---

2. Comparison Operators

Operator	Meaning
$gt	Greater than
$gte	Greater than or equal
$lt	Less than
$lte	Less than or equal
$ne	Not equal

Examples

Find products more expensive than ₹10,000:

```
db.products.find({ price: { $gt: 10000 } });
```

Find products NOT in Electronics:

```
db.products.find({ category: { $ne: "Electronics" } });
```

---

3. Logical Operators

Operator	Meaning
$and	AND condition
$or	OR condition
$in	Matches any value in list
$nin	Not in list

Examples

```
db.products.find({
  $or: [
    { category: "Electronics" },
    { price: { $lt: 100 } }
  ]
});
```

---

4. Projection

Used to **select specific fields**.

Syntax:

```
db.collection.find(query, { field: 1, _id: 0 })
```

Examples

Return only name and price:

```
db.products.find({}, { name: 1, price: 1, _id: 0 });
```

Hide large embedded fields

```
db.entities.find({}, { officers: 0 });
```

---

5. Querying Embedded Fields

```
db.entities.find({ "officers.role": "Director" });
```

D. LAB – Perform CRUD Queries on a Sample Dataset (15 minutes)

Students will create a small dataset and perform CRUD reads using filters, projections, and operators.

Step 1 – Select Database

```
use csc_crud_training
```

Step 2 – Insert Sample Dataset

Insert multiple employees

```
db.employees.insertMany([
  { name: "Anita", dept: "HR", city: "Bangalore", salary:
60000 },
  { name: "Rohit", dept: "IT", city: "Pune", salary: 85000 },
  { name: "Samuel", dept: "IT", city: "Hyderabad", salary:
92000 },
  { name: "Priya", dept: "Finance", city: "Bangalore", salary:
75000 }
]);
```

Step 3 – Simple Reads

Fetch all employees:

```
db.employees.find();
```

Find employees in Bangalore:

```
db.employees.find({ city: "Bangalore" });
```

Step 4 – Comparison Filters

```
db.employees.find({ salary: { $gt: 80000 } });
```

Step 5 – Logical Filters

```
db.employees.find({
  $and: [
    { dept: "IT" },
    { salary: { $gt: 85000 } }
  ]
});
```

Step 6 – Projection Example

Show only name and salary:

```
db.employees.find({}, { name: 1, salary: 1, _id: 0 });
```

Step 7 – Embedded Document Example (Insert & Query)

```
db.customers.insertOne({
  name: "Rakesh",
  address: { city: "Delhi", pincode: 110001 },
  orders: [
    { id: "ORD001", amount: 1200 },
    { id: "ORD002", amount: 2200 }
  ]
});
```

Query embedded array:

```
db.customers.find({ "orders.id": "ORD002" });
```

🎯 LAB Outcomes

Learners will understand:

- How to insert documents
- How to use find() and findOne()
- Filter documents using operators
- Use projection to optimize reads
- Query embedded and nested structures

---

7. MongoDB CRUD Operations – Replace & Delete (30 min)

- replaceOne()
- deleteOne(), deleteMany()
- Lab: Data cleanup operations

Below is the **complete instructor-ready module** for:

This module introduces the **Replace** and **Delete** operations in MongoDB, used by DBAs and developers for updating entire documents and performing cleanup tasks.

A. replaceOne() (10 minutes)

replaceOne() replaces the **entire document** except for _id.

✓ Syntax:

```
db.collection.replaceOne(filter, replacementDocument)
```

✓ When to use:

- Replace outdated/incorrect documents
- Overwrite entire records during migration
- Reset document structure

❗ Important:

- _id **cannot** be changed
- If fields are missing in the new document → they are removed

---

✓ Example 1 — Replace a user document

Original document:

```
{
  _id: ObjectId("..."),
  name: "Ravi",
  city: "Pune",
  email: "ravi@test.com"
}
```

Replace:

```
db.users.replaceOne(
  { name: "Ravi" },
  { name: "Ravi Kumar", city: "Bangalore", active: true }
);
```

The field email is now removed (because not included in replacement).

✓ Example 2 — CSC Legal Entity update

```
db.entities.replaceOne(
  { entityName: "CSC India Pvt Ltd" },
  {
    entityName: "CSC India Pvt Ltd",
    jurisdiction: "Karnataka",
    complianceStatus: "Active",
    updatedBy: "adminUser"
  }
);
```

B. deleteOne() and deleteMany() (10 minutes)

✓ 1. deleteOne()

Deletes **the first** document matching the filter.

```
db.collection.deleteOne({ field: value })
```

Example:

```
db.products.deleteOne({ name: "Notebook" });
```

✓ 2. deleteMany()

Deletes **all documents** matching the filter.

```
db.collection.deleteMany({ category: "Expired" });
```

✓ Example: Delete inactive CSC compliance logs

```
db.complianceLogs.deleteMany({ status: "obsolete" });
```

✓ Example: Delete employees from a specific city

```
db.employees.deleteMany({ city: "Delhi" });
```

! Safety Tips for Delete Operations

- Always run find() with same filter before delete
- Ensure backups exist
- For high-risk operations, use transactions (in replica sets/sharded clusters)
- Avoid empty filter {} unless intentional

---

8. Modifying Query Results (1 hr)

- sort(), skip(), limit()
- Aggregation preview
- Query optimization basics
- Lab: Apply query modifiers for performance

MongoDB allows result modification using:

- sort()
- limit()
- skip()

  These operations help optimize read queries, especially for pagination and reporting.

A. sort(), limit(), skip() (25 minutes)

These functions modify **how results are returned**, not how they are stored.

1. sort()

Sorts documents based on one or more fields.

✓ Syntax:

```
db.collection.find().sort({ field: 1 })    // ascending
db.collection.find().sort({ field: -1 })   // descending
```

✓ Example – Sort employees by salary (highest first)

```
db.employees.find().sort({ salary: -1 });
```

✓ Example – Multi-field sort

Sort by department ascending, then salary descending:

```
db.employees.find().sort({ dept: 1, salary: -1 });
```

⚠ Performance Note:

- Sorting **without an index** can cause blocking operations.
- Create an index on sorted fields for faster results.

---

2. limit()

Restricts number of documents returned.

✓ Syntax:

```
db.collection.find().limit(5)
```

Example – Get top 5 highest salaries:

```
db.employees.find().sort({ salary: -1 }).limit(5);
```

---

3. skip()

Skip first N documents.

✓ Syntax:

```
db.collection.find().skip(10)
```

Example – Pagination:

```
db.employees.find().sort({ name: 1 }).skip(20).limit(10);
```

⭐ Pagination Pattern (skip & limit)

Used for:

- CSC dashboards
- Paginated corporate records
- Trademark search results

Typical pattern:

```
db.collection.find()
    .sort({ createdAt: -1 })
    .skip(page * pageSize)
    .limit(pageSize)
```

B. Aggregation Preview (10 minutes)

Aggregation pipeline = MongoDB’s equivalent of SQL GROUP BY + WHERE + ORDER + FUNCTIONS.

⭐ Common Stages:

✓ 1. $match – Filter documents

```
{ $match: { dept: "IT" } }
```

✓ 2. $group – Group and aggregate

```
{ $group: { _id: "$dept", totalSalary: { $sum: "$salary" } } }
```

✔ 3. $sort – Sort aggregated output

```
{ $sort: { totalSalary: -1 } }
```

⭐ Example – Department salary totals

```
db.employees.aggregate([
  { $match: {} },
  { $group: { _id: "$dept", total: { $sum: "$salary" } } },
  { $sort: { total: -1 } }
]);
```

⭐ Why Aggregation Matters for CSC?

- Generating compliance dashboards
- Risk scoring
- Trademark classification reports
- Officer distribution summaries

C. Query Optimization Basics (10 minutes)

Efficient queries reduce latency and resource usage.

⭐ 1. Use Indexes on Sorted or Filtered Fields

Sorting without an index forces **in-memory sort** → slow.

Example:

```
db.employees.find().sort({ salary: -1 });
```

▶️ Needs index:

```
db.employees.createIndex({ salary: -1 });
```

⭐ 2. Avoid skip() for large offsets

skip(1000000) becomes slow.

➡️ Alternatives:

- Range queries
- Bookmark-based pagination

⭐ 3. Use Projection to Reduce Data Size

Return only required fields:

```
db.employees.find({}, { name: 1, dept: 1, _id: 0 })
```

⭐ 4. Analyze Query Plan with explain()

```
db.employees.find({ dept: "IT" }).explain("executionStats");
```

Key terms:

- **COLLSCAN** → full collection scan (slow)
- **IXSCAN** → index scan (fast)
- **nReturned**
- **executionTimeMillis**

⭐ 5. Avoid OR conditions when possible

Prefer:

```
$in: ["Pune", "Bangalore"]
```

instead of:

```
$or: [{ city: "Pune" }, { city: "Bangalore" }]
```

==

D. LAB – Apply Query Modifiers for Performance (15 minutes)

Goal:

Use sort(), limit(), skip() and analyze performance with explain().

Step 1 – Dataset Setup

```
use csc_querylab;

for (let i = 1; i <= 50000; i++) {
  db.employees.insertOne({
    empId: i,
    name: "Employee" + i,
    dept: ["IT", "HR", "Finance", "Legal"][i % 4],
    salary: Math.floor(Math.random() * 90000) + 30000,
    city: ["Bangalore", "Pune", "Chennai", "Delhi"][i % 4]
  });
}
```

Step 2 – Query Without Index

```
db.employees.find({ city: "Bangalore" }).sort({ salary: -1
}).limit(5);
```

Step 3 – Check Query Plan

```
db.employees.find({ city: "Bangalore" })
    .sort({ salary: -1 })
    .limit(5)
    .explain("executionStats");
```

Look for:

- COLLSCAN
- High nScannedDocs
- High executionTimeMillis

Step 4 – Create Index

```
db.employees.createIndex({ city: 1, salary: -1 });
```

Step 5 – Run Query Again

```
db.employees.find({ city: "Bangalore" })
    .sort({ salary: -1 })
    .limit(5)
    .explain("executionStats");
```

Expected:

- IXSCAN
- Fewer scanned docs
- Lower execution time

Step 6 – Pagination Exercise

```
db.employees.find({ dept: "IT" })
    .sort({ empId: 1 })
    .skip(100)
    .limit(10);
```

End of Day Assignment

- CRUD tasks on sample collections
- Short quiz (MCQ + practical)

Day 1 – End-of-Day Assignment

Covers Modules 1–8 (MongoDB Basics + CRUD + Query Modifiers)

PART A — CRUD TASKS ON SAMPLE COLLECTIONS

(Hands-On Assignment – 45 minutes)

Use a new database:

```
use day1_assignment;
```

1️⃣ Create Collections & Insert Documents

Task 1. Insert sample employees

Insert **5 employee** documents with fields:

- name
- department
- city
- salary
- active (boolean)

Use **insertMany()**.

Task 2. Insert CSC-style legal entities

Insert **3 documents** with fields:

- entityName
- jurisdiction
- status
- officers (array of embedded documents)
- lastFiled (Date)

Use **insertOne()**.

2️⃣ Find & Filter Operations

Task 3. Read operations

Write queries using **find()** to:

a) Get all employees from Bangalore

b) Get employees with salary > 75,000

c) Get legal entities with status = "Active"

d) Find officers whose role = “Director” using embedded query

e) Project only name and salary fields for employees

3️⃣ Update & Replace Operations

Task 4. Update employee city

Update city of **one employee** from Bangalore → Pune.

Use:

```
updateOne({ ... }, { $set: { ... } })
```

Task 5. Replace an entire entity document

Replace one entity document with a **new structure** (except _id).

Use:

```
replaceOne()
```

4️⃣ Delete Operations

Task 6. Delete inactive employees

Delete employees where:

```
{ active: false }
```

Task 7. Delete officers with specific role inside an entity

(Use $pull operator)

Example:

```
$pull: { officers: { role: "Secretary" } }
```

==

PART B — QUERY MODIFIER TASKS

(sort, skip, limit)

Task 8. Sort employees by salary, descending

```
sort({ salary: -1 })
```

Task 9. Return the top 3 highest-paid employees

Use sort + limit.

Task 10. Apply pagination

Fetch **page 2** of results, page size = 2 employees.

Use:

```
sort().skip().limit()
```

PART C — PRACTICAL QUESTIONS (Short Coding Problems)

(20 minutes)

Q1. Insert 100 sample audit logs using a loop.

Each log should contain:

- logId
- user
- ts (timestamp)

Q2. Query audit logs created today.

Q3. Write a query to fetch employees with salary BETWEEN 60,000 and 90,000.

Use $gte and $lte.

Q4. Write a query to return only employees from IT or HR department.

Use $in.

Q5. Write a projection query to hide _id and show only name + department.

Q6. Find all customers whose order total > ₹10,000 using embedded query.

Dataset example:

```
{
  name: "...",
  orders: [
    { id: "O1", amount: 5000 },
    { id: "O2", amount: 6000 }
  ]
}
```

Q7. Explain the difference between find() and findOne().

Q8. Why is limit() important for optimization? Give 1 example.

PART D — SHORT MCQ QUIZ (10 Questions) (10 minutes)

1. Which command inserts multiple documents?

a) insertAll()

b) insertMany()

c) createMany()

d) saveMany()

---

2. Which query returns only the first matching document?

a) find()

b) findFirst()

c) findOne()

d) findTop()

---

3. Which operator finds values greater than 100?

a) $gt

b) $gte

c) $greater

d) >

---

4. What does projection control?

a) Which documents to insert

b) Which fields to show

c) Which documents to delete

d) Which database to create

---

5. replaceOne() removes missing fields in the new document.

a) True

b) False

---

6. deleteMany() deletes:

a) First matching document

b) All matching documents

c) Documents with _id only

d) None

---

7. sort({salary: -1}) means:

a) Sort by salary ascending

b) Sort by salary descending

c) Group by salary

d) Remove salary field

---

8. skip(5) means:

a) Delete 5 documents

b) Skip the first 5 documents

c) Sort 5 documents

d) Insert 5 documents

---

9. Which operator matches ANY of the listed values?

a) $match

b) $in

c) $or

d) $contains

---

10. explain("executionStats") helps understand:

a) Document size

b) Query plan + performance

c) Password encryption

d) Replica set members

ANSWER KEY (MCQ)

1-b

2-c

3-a

4-b

5-a

6-b

7-b

8-b

9-b

10-b