Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

INTRODUCTION TO MONGODB & NOSQL DATABASES (WEEKS 1-2)

INTRODUCTION TO NOSQL DATABASES & COMPARISON WITH SQL

CHAPTER 1: UNDERSTANDING NOSQL DATABASES

1.1 What Are NoSQL Databases?

NoSQL (**Not Only SQL**) databases are designed to handle **large-scale, unstructured, and semi-structured data** more efficiently than traditional **relational databases**. Unlike SQL databases, which store data in structured tables with predefined schemas, NoSQL databases offer **flexibility, scalability, and high availability**.

Key Characteristics of NoSQL Databases:

✓ **Schema-less** – No predefined table structure, allowing dynamic data changes.

✓ **Scalable** – Designed to handle large datasets and high traffic.

✓ **Distributed Architecture** – Spreads data across multiple servers for fault tolerance.

✓ **Flexible Data Models** – Supports **key-value, document, column-family, and graph-based** structures.

1.2 Why Use NoSQL Databases?

Traditional SQL databases work well for structured data, but modern applications require **fast and scalable data storage**. NoSQL databases are ideal for:

- **Big Data & Real-Time Analytics** (e.g., recommendation engines, IoT).
- **High-Traffic Applications** (e.g., social media, e-commerce).
- **Flexible Schema Requirements** (e.g., evolving data models).

CHAPTER 2: TYPES OF NOSQL DATABASES

2.1 Key-Value Databases

- **Data Storage:** Stores data as key-value pairs.
- **Best For:** Caching, real-time analytics.
- **Examples: Redis, DynamoDB, Riak**.

Example: Key-Value Pair in Redis

SET user:123 "John Doe"

GET user:123

2.2 Document-Oriented Databases

- **Data Storage:** Stores data as **JSON-like documents**.
- **Best For:** Content management, catalogs.
- **Examples: MongoDB, CouchDB, Firebase Firestore**.

Example: Storing a User Profile in MongoDB

```
{
 "name": "Alice",
 "email": "alice@example.com",
 "age": 30
}
```

2.3 Column-Family Databases

- **Data Storage:** Uses columns instead of rows for better query performance.
- **Best For:** Analytics, data warehousing.
- **Examples: Apache Cassandra, HBase**.

Example: Storing Data in Cassandra

UserID	Name	Age	Email
101	Alice	30	alice@example.com

2.4 Graph Databases

- **Data Storage:** Stores relationships between data points.
- **Best For:** Social networks, fraud detection.
- **Examples: Neo4j, ArangoDB, Amazon Neptune**.

Example: Representing Friendships in Neo4j

```
CREATE (Alice)-[:FRIENDS_WITH]->(Bob)
```

CHAPTER 3: COMPARING NOSQL WITH SQL DATABASES

3.1 Differences Between SQL & NoSQL Databases

Feature	SQL Databases	NoSQL Databases
Data Structure	Tables (Rows & Columns)	Documents, Key-Value, Graph
Schema	Fixed schema required	Schema-less
Scalability	Vertical scaling (limited)	Horizontal scaling (distributed)
Data Integrity	Strong ACID compliance	BASE Model (Eventual Consistency)
Best For	Structured, transactional data	Unstructured, flexible data

✓ **SQL databases** excel in **data consistency & transactions**.

✓ **NoSQL databases** excel in **performance & scalability**.

3.2 When to Use SQL vs. NoSQL?

Use Case	SQL Database?	NoSQL Database?
Banking & Financial Apps	✅ Yes	❌ No
E-Commerce Product Catalog	❌ No	✅ Yes
Social Media Platforms	❌ No	✅ Yes
Enterprise Resource Planning (ERP)	✅ Yes	❌ No
Real-Time Chat Apps	❌ No	✅ Yes

Case Study: How Amazon Uses NoSQL Databases for Scalability

Background

Amazon processes **millions of orders daily,** requiring a **highly scalable and efficient** data management system.

Challenges

- **Handling large-scale real-time transactions.**
- **Providing fast and consistent user experiences.**
- **Scaling databases during peak shopping events (Black Friday, Prime Day).**

Solution: Using NoSQL Databases

✓ Amazon implemented **DynamoDB (Key-Value Store)** to handle transactions at scale.
✓ Used **MongoDB for flexible product catalogs** with dynamic schema changes.
✓ Deployed **Neo4j (Graph Database) for personalized recommendations**.

This allowed **Amazon to scale seamlessly,** improving speed and customer experience.

Exercise

1. List three key differences between SQL and NoSQL databases.
2. Which type of NoSQL database is best for **social media applications**?

3. Write a **MongoDB document** for storing an **order** with fields orderID, userID, and items.

Conclusion

In this section, we explored:

✓ **What NoSQL databases are and their advantages**.

✓ **Different types of NoSQL databases (Key-Value, Document, Column-Family, Graph)**.

✓ **Comparison of NoSQL vs. SQL databases and their best use cases**.

✓ **How Amazon uses NoSQL to handle millions of transactions daily**.

OVERVIEW OF MONGODB ARCHITECTURE & BSON FORMAT

CHAPTER 1: INTRODUCTION TO MONGODB ARCHITECTURE

1.1 Understanding MongoDB as a NoSQL Database

MongoDB is a **NoSQL document-oriented database** that stores data in a **flexible, JSON-like format** called **BSON**. Unlike relational databases that use **tables and rows**, MongoDB structures data in **collections and documents**.

✓ **Document-based storage** – Stores data as flexible **JSON-like documents** instead of rows.
✓ **Schema-less design** – Allows dynamic fields, making it adaptable for different applications.
✓ **Horizontal scalability** – Uses **sharding** to distribute data across multiple servers.
✓ **High performance** – Supports fast reads and writes with indexing and replication.

MongoDB is commonly used for **big data, real-time analytics, and high-traffic applications** like:

- E-commerce platforms
- Social media applications
- Internet of Things (IoT) data storage
- Content management systems

1.2 Key Components of MongoDB Architecture

The MongoDB architecture consists of the following key components:

1. **Databases** – Similar to a relational database but stores collections instead of tables.
2. **Collections** – Groups of related **documents** (like tables in SQL).
3. **Documents** – The primary unit of data storage**,** represented in **BSON format**.
4. **Indexes** – Speed up data retrieval by allowing efficient querying.
5. **Replication** – Ensures **high availability** by duplicating data across multiple servers.
6. **Sharding** – Distributes data across **multiple nodes** to handle large datasets.

CHAPTER 2: DOCUMENT-BASED STORAGE IN MONGODB

2.1 How Data is Stored in MongoDB

Instead of storing data in **rows and columns,** MongoDB uses **documents** with a JSON-like structure.

Example: JSON Document in MongoDB

```
{
  "_id": "635f2a1b5f9d2c1f2d9b1234",
  "name": "John Doe",
  "email": "john@example.com",
  "age": 30,
  "isAdmin": false
}
```

✓ Each document is stored in a **collection** instead of a table.

✓ **Documents are flexible,** meaning fields can vary between documents in the same collection.

2.2 Collections vs. Tables in SQL Databases

Feature	MongoDB (NoSQL)	SQL Databases
Storage Unit	Documents	Rows & Columns
Structure	Schema-less	Fixed Schema
Scalability	Horizontal (Sharding)	Vertical (Scaling Up)
Flexibility	High	Low
Best For	Big data, real-time apps	Structured transactional data

MongoDB's **document-oriented model** makes it **ideal for applications requiring flexible schemas** and high-speed data processing.

CHAPTER 3: BSON FORMAT IN MONGODB

3.1 What is BSON?

BSON (Binary JSON) is the **binary-encoded serialization format** used by MongoDB to store documents. While it is similar to JSON, BSON provides additional **data types and better performance**.

✓ **Compact and efficient** – Uses binary encoding to reduce storage size.

✓ **Supports additional data types** – Includes **Date, Binary Data, and ObjectId**.

✓ **Fast serialization/deserialization** – Improves **query performance**.

3.2 BSON vs. JSON: Key Differences

Feature	JSON	BSON (MongoDB)
Format	Text-based (human-readable)	Binary-encoded (compact & efficient)
Data Types	Limited (string, number, boolean)	Supports additional types (Date, Binary, ObjectId)
Performance	Slower to parse	Faster parsing & storage
Size	Larger file size	Smaller, optimized for storage

3.3 Example: BSON Representation of a Document

A simple **JSON document**:

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

When converted to **BSON**, it looks like this (in binary representation):

\x05\x00\x00\x00\x02name\x00\x06\x00\x00\x00Alice\x00\x10age\x00\x19\x00\x00\x00

✓ **Compact encoding** reduces storage space.
✓ MongoDB **automatically converts JSON to BSON** when storing data.

CHAPTER 4: UNDERSTANDING MONGODB OBJECTID

4.1 What is an ObjectId in MongoDB?

Every MongoDB document contains a unique identifier called **ObjectId,** which serves as the **primary key** (_id).

Example: ObjectId in a MongoDB Document

```
{
  "_id": ObjectId("635f2a1b5f9d2c1f2d9b1234"),
  "name": "Alice",
  "email": "alice@example.com"
}
```

4.2 Structure of an ObjectId

An ObjectId is a **12-byte identifier** composed of:

1. **Timestamp** (4 bytes) – Time when the ObjectId was generated.
2. **Machine ID** (3 bytes) – Identifies the host generating the ObjectId.
3. **Process ID** (2 bytes) – Identifies the process generating the ObjectId.
4. **Counter** (3 bytes) – Ensures uniqueness among ObjectIds.

✓ The timestamp makes ObjectId **sortable**.
✓ Ensures that each document has a **unique and efficient primary key**.

4.3 Generating ObjectId in MongoDB

In MongoDB shell:

ObjectId()

In Node.js using Mongoose:

```
const mongoose = require('mongoose');

const newId = new mongoose.Types.ObjectId();

console.log(newId);
```

✓ **Ensures globally unique identifiers** without requiring an additional auto-increment field.

CHAPTER 5: MONGODB INDEXING AND PERFORMANCE OPTIMIZATION

5.1 What is Indexing in MongoDB?

Indexes **speed up database queries** by allowing MongoDB to search documents **efficiently**.

✓ Without indexing, MongoDB **scans the entire collection** (slow).
✓ With indexing, MongoDB **retrieves data faster** (optimized).

5.2 Creating an Index

To create an index on the email field:

```
db.users.createIndex({ email: 1 })
```

✓ The 1 specifies **ascending order** indexing.
✓ Indexes improve performance **for large datasets**.

5.3 Viewing Indexes

To check existing indexes in a collection:

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

✓ Indexes help optimize queries and **reduce execution time**.

Case Study: How a Social Media Platform Used MongoDB for High-Performance Data Storage

Background

A **social media startup** needed a **scalable database** to handle:

✓ **Millions of user profiles and posts**.
✓ **High-speed queries** for fetching posts and comments.
✓ **Efficient indexing** for real-time searches.

Challenges

- Traditional **relational databases** struggled with **scalability**.
- Data retrieval was **slow** due to large dataset sizes.
- Searching posts took **too long,** reducing user engagement.

Solution: Implementing MongoDB

The team used:

✓ **BSON format** for **efficient storage and fast queries**.
✓ **Indexing on username and post timestamps** for quick searches.
✓ **Sharding** to distribute data across multiple servers.

Results

- **100x faster searches,** improving user experience.
- **Scalability enabled,** handling **millions of users**.
- **Efficient storage,** reducing database costs.

By leveraging MongoDB's **BSON format and indexing,** the platform **improved performance and scalability**.

Exercise

1. Create a MongoDB document with the following fields:
    - name (string)
    - email (string, unique)
    - createdAt (Date, default: current time)
2. Convert the document into **BSON format** and observe its **binary representation**.
3. Generate a new **ObjectId** in MongoDB and extract its **timestamp**.
4. Create an **index on the email field** and verify the query speed improvement.

Conclusion

In this section, we explored:

✓ **How MongoDB uses BSON for efficient data storage**.

✓ **How ObjectId ensures unique document identification**.

✓ **How indexing improves query performance**.

INSTALLING MONGODB LOCALLY & SETTING UP MONGODB ATLAS

CHAPTER 1: INTRODUCTION TO MONGODB AND ITS SETUP

1.1 Understanding MongoDB

MongoDB is a **NoSQL database** that stores data in **JSON-like documents** instead of traditional tables. It is widely used for:

✓ **Scalability** – Handles large volumes of unstructured data.
✓ **Flexibility** – Schema-less structure allows dynamic fields.
✓ **High Performance** – Optimized for fast reads and writes.

MongoDB can be deployed in two ways:

- **Locally** – Installed on a personal computer or a server.
- **Cloud-Based (MongoDB Atlas)** – A fully managed cloud database service.

Each setup has its advantages. **Local installation** is useful for **development and testing**, while **MongoDB Atlas** is ideal for **scalability, backups, and production use**.

CHAPTER 2: INSTALLING MONGODB LOCALLY ON WINDOWS, MACOS, AND LINUX

2.1 Installing MongoDB on Windows

Step 1: Download MongoDB

1. Visit the MongoDB Download Center.
2. Select the **Community Edition** for Windows.

3. Download the .msi installer and run it.

Step 2: Install MongoDB

1. Select **Complete Installation** and check **Install MongoDB as a Service**.
2. Click **Next** and complete the installation.

Step 3: Verify Installation

Open **Command Prompt** and run:

mongod --version

To start MongoDB, use:

net start MongoDB

To stop MongoDB:

net stop MongoDB

2.2 Installing MongoDB on macOS

Step 1: Install Using Homebrew

brew tap mongodb/brew

brew install mongodb-community@6.0

Step 2: Start the MongoDB Service

brew services start mongodb-community@6.0

To check if MongoDB is running:

brew services list

Step 3: Verify Installation

mongod --version

To stop MongoDB:

brew services stop mongodb-community@6.0

2.3 Installing MongoDB on Linux (Ubuntu)

Step 1: Import MongoDB Key

wget -qO - https://www.mongodb.org/static/pgp/server-6.0.asc | sudo apt-key add -

Step 2: Add the MongoDB Repository

echo "deb [arch=amd64,arm64] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/6.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-6.0.list

Step 3: Install MongoDB

sudo apt update

sudo apt install -y mongodb-org

Step 4: Start MongoDB Service

sudo systemctl start mongod

sudo systemctl enable mongod

Step 5: Verify Installation

mongod --version

To stop MongoDB:

sudo systemctl stop mongod

CHAPTER 3: CONNECTING TO MONGODB LOCALLY

3.1 Using the MongoDB Shell

Start the MongoDB shell by running:

mongosh

To check available databases:

show dbs

To create a new database:

use myDatabase

To insert a document:

db.users.insertOne({ name: "Alice", age: 25 })

To retrieve data:

db.users.find()

CHAPTER 4: SETTING UP MONGODB ATLAS (CLOUD DATABASE)

4.1 Creating a MongoDB Atlas Account

1. **Sign up** at MongoDB Atlas.
2. Click **Create a New Project** and name it (e.g., "MyProject").
3. Click **Build a Cluster** and select **Free Shared Cluster**.
4. Choose **Cloud Provider & Region** (AWS, GCP, or Azure).
5. Set **Cluster Name** (e.g., "MyCluster").
6. Click **Create Cluster**.

4.2 Configuring Database Access

1. Go to **Database Access** → Click **Add New User**.
2. Select **Username & Password** authentication.
3. Grant **read and write permissions**.
4. Save credentials (will be used to connect).

4.3 Whitelisting Your IP Address

1. Go to **Network Access** → Click **Add IP Address**.
2. Select **Allow Access from Anywhere** (for testing) or **Add specific IP**.
3. Click **Confirm**.

CHAPTER 5: CONNECTING A NODE.JS APP TO MONGODB ATLAS

5.1 Install Mongoose

npm install mongoose

5.2 Connect to MongoDB Atlas in Node.js

Modify **.env** file:

MONGO_URI=mongodb+srv://username:password@mycluster.mongodb.net/myDatabase

Update **config/db.js**:

const mongoose = require('mongoose');

require('dotenv').config();

mongoose.connect(process.env.MONGO_URI, {

```
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('MongoDB Atlas Connected'))
.catch(err => console.error('MongoDB Connection Error:', err));
```

Case Study: How a SaaS Company Scaled Using MongoDB Atlas

Background

A **SaaS company** needed a **scalable database** to handle thousands of users.

Challenges

✓ Local MongoDB was **hard to manage** with increasing traffic.
✓ **Downtime issues** impacted performance.

Solution: Migrating to MongoDB Atlas

✓ Used **cloud-hosted clusters** for auto-scaling.
✓ Implemented **replication and backups** for high availability.
✓ Integrated **Mongoose** to manage database operations.

Results

- **99.9% uptime,** improving user experience.
- **Effortless scaling,** handling **5x more users**.
- **Automated backups,** reducing data loss risks.

This case study highlights how **MongoDB Atlas improves database reliability and scalability**.

Exercise

1. Install **MongoDB locally** and verify the installation.
2. Create a **MongoDB Atlas account** and set up a **free cluster**.
3. Write a Node.js script to **connect to MongoDB Atlas** and insert a test document.

Conclusion

✓ MongoDB can be installed locally or on the cloud (Atlas).
✓ Local installation is best for development, Atlas is ideal for production.
✓ Using Mongoose simplifies database management.

INTRODUCTION TO MONGODB SHELL & COMPASS

CHAPTER 1: INTRODUCTION TO MONGODB AND ITS TOOLS

1.1 What is MongoDB?

MongoDB is a **NoSQL database** that stores data in **JSON-like documents** instead of traditional tables. It is widely used for modern web applications because of its **flexibility, scalability, and high performance**. Unlike relational databases such as MySQL and PostgreSQL, MongoDB allows for **dynamic schemas**, making it ideal for handling unstructured and semi-structured data.

Key Features of MongoDB:

- **Document-Oriented** – Stores data in BSON (Binary JSON) format.
- **Schema-less** – No fixed schema required, making it flexible.
- **Scalability** – Supports horizontal scaling across multiple servers.
- **Indexing & High-Speed Queries** – Provides efficient indexing for faster searches.
- **Replication & Fault Tolerance** – Ensures data availability with replica sets.

1.2 MongoDB Tools: Shell & Compass

MongoDB provides two primary tools for interacting with the database:

- **MongoDB Shell (mongosh)** – A command-line interface for managing MongoDB databases.

- **MongoDB Compass** – A graphical user interface (GUI) for visually interacting with MongoDB databases.

Both tools allow developers to perform CRUD (Create, Read, Update, Delete) operations, manage indexes, and execute queries efficiently.

CHAPTER 2: INSTALLING MONGODB SHELL (MONGOSH)

2.1 Installing MongoDB

Step 1: Download MongoDB

- Visit the MongoDB official website.
- Choose the correct version for your operating system (Windows, macOS, Linux).
- Install MongoDB following the on-screen instructions.

Step 2: Verify Installation

Once installed, check if MongoDB is installed correctly by running:

```
mongod --version
```

This command displays the installed MongoDB version.

2.2 Using MongoDB Shell (mongosh)

MongoDB Shell (mongosh) is the command-line interface for MongoDB, replacing the older mongo shell. It allows users to **create, modify, and query databases** using JavaScript-based commands.

Start the MongoDB Shell

To launch mongosh, open a terminal and run:

```
mongosh
```

If MongoDB is running, you will see an interactive shell prompt:

Current MongoDB Server version: 6.0

Using MongoDB: test>

Basic MongoDB Shell Commands

1. **Show available databases**
2. show dbs
3. **Create or switch to a database**
4. use myDatabase
5. **Create a collection and insert data**
6. db.users.insertOne({ name: "Alice", age: 25, city: "New York" })
7. **Find all documents in a collection**
8. db.users.find()
9. **Update a document**
10. db.users.updateOne({ name: "Alice" }, { $set: { age: 26 } })
11. **Delete a document**
12. db.users.deleteOne({ name: "Alice" })

MongoDB Shell is useful for **quick database administration and query execution**.

CHAPTER 3: INSTALLING AND USING MONGODB COMPASS

3.1 What is MongoDB Compass?

MongoDB Compass is a **GUI tool** that provides a visual interface for interacting with MongoDB databases. It is ideal for developers who

prefer graphical database management instead of command-line operations.

3.2 Installing MongoDB Compass

Step 1: Download MongoDB Compass

- Go to the MongoDB Compass Download Page.
- Choose the correct version for your operating system.
- Install MongoDB Compass by following the setup instructions.

Step 2: Launch MongoDB Compass

Once installed, open MongoDB Compass. You will see a **connection screen** where you can enter the database connection details.

3.3 Connecting to a MongoDB Database

To connect to a **local MongoDB instance,** enter the default **connection string**:

mongodb://localhost:27017

Click **Connect,** and MongoDB Compass will display the list of available databases.

CHAPTER 4: PERFORMING DATABASE OPERATIONS WITH MONGODB COMPASS

4.1 Creating a Database and Collection

1. Click **"Create Database"** in MongoDB Compass.
2. Enter a **database name** (e.g., myDatabase).
3. Enter a **collection name** (e.g., users).
4. Click **Create Database**.

Now, you have a MongoDB database with an empty users collection.

4.2 Inserting Data into a Collection

1. Open the users collection.
2. Click **"Insert Document"**.
3. Enter the following JSON data:
4. {
5. "name": "Alice",
6. "age": 25,
7. "city": "New York"
8. }
9. Click **Insert** to save the document.

4.3 Querying Data in MongoDB Compass

1. Go to the users collection.
2. Click the **Filter** section and enter:
3. { "city": "New York" }
4. Press **Apply**, and MongoDB Compass will show all users from New York.

4.4 Updating and Deleting Data

- **To update a document,** click on a document and edit the values manually.
- **To delete a document,** select the document and click **"Delete"**.

Case Study: How eCommerce Platforms Use MongoDB for Real-Time Data Management

Background

An **eCommerce company** needed a database system that could **handle millions of product listings and real-time customer interactions**.

Challenges

- High volume of transactions requiring **fast read/write operations**.
- Managing **dynamic product information** (prices, stock availability).
- Supporting **real-time search and filtering** of products.

Solution: Using MongoDB

✓ **Stored product listings dynamically** using MongoDB's flexible schema.
✓ **Implemented real-time search** with indexing for quick product filtering.
✓ **Used MongoDB Compass for data visualization,** allowing admins to track sales and inventory.

Results

- **30% faster product search** compared to traditional SQL databases.
- **Better scalability** with millions of users accessing data simultaneously.
- **Improved backend performance,** allowing seamless transactions.

This case study highlights how **MongoDB's NoSQL capabilities enhance real-time applications**.

Exercise

1. What is the difference between MongoDB Shell and MongoDB Compass?
2. Write a command to create a new MongoDB database using mongosh.
3. In MongoDB Compass, how do you insert a new document into a collection?

Conclusion

In this section, we explored:

✓ **What MongoDB is and why it is useful for modern applications**.

✓ **How to install and use MongoDB Shell (mongosh) for command-line database management**.

✓ **How to install and use MongoDB Compass for GUI-based database interactions**.

CREATING, READING, UPDATING, AND DELETING DOCUMENTS IN MONGODB

CHAPTER 1: INTRODUCTION TO CRUD OPERATIONS IN MONGODB

1.1 Understanding CRUD in MongoDB

CRUD stands for **Create, Read, Update, and Delete**, which are the four fundamental operations used to manage data in a database. In **MongoDB**, these operations are performed on **documents** within **collections**.

✓ **Create (Insert Data)** – Add new documents to a collection.
✓ **Read (Retrieve Data)** – Fetch documents based on queries.
✓ **Update (Modify Data)** – Modify existing documents.
✓ **Delete (Remove Data)** – Remove unwanted documents.

MongoDB stores data in a **flexible, JSON-like format called BSON**, allowing **schema-less** data storage.

CHAPTER 2: CREATING DOCUMENTS IN MONGODB

2.1 Using insertOne() to Add a Single Document

To insert a **single document** into a collection, use the insertOne() method.

Example: Inserting a Single Document

```
db.users.insertOne({
  name: "Alice Johnson",
  email: "alice@example.com",
```

```
  age: 28,
  city: "New York"
});
```

✓ This will create a **document** in the users collection.

2.2 Using insertMany() to Insert Multiple Documents

To add **multiple documents** at once, use insertMany().

Example: Inserting Multiple Documents

```
db.users.insertMany([
  { name: "Bob Smith", email: "bob@example.com", age: 35, city: "Los Angeles" },
  { name: "Charlie Brown", email: "charlie@example.com", age: 29, city: "Chicago" }
]);
```

✓ This inserts multiple records at once, improving performance.

CHAPTER 3: READING DOCUMENTS IN MONGODB

3.1 Using findOne() to Retrieve a Single Document

To fetch a **single document**, use findOne().

Example: Retrieving a Single User

```
db.users.findOne({ name: "Alice Johnson" });
```

✓ Returns **one document** that matches the query.

3.2 Using find() to Retrieve Multiple Documents

To retrieve **multiple documents,** use find().

Example: Retrieving All Users

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

✓ Returns **all documents** in the users collection.

3.3 Using Query Filters in find()

To filter documents based on conditions:

Example: Finding Users Older Than 30

```
db.users.find({ age: { $gt: 30 } });
```

✓ $gt (greater than) filters users where age > 30.

3.4 Formatting Query Results with pretty()

Use pretty() for a more readable output:

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

✓ Displays results in a **well-structured format**.

CHAPTER 4: UPDATING DOCUMENTS IN MONGODB

4.1 Using updateOne() to Modify a Single Document

To update **one document,** use updateOne().

Example: Updating a User's City

```
db.users.updateOne(
```

```
  { email: "alice@example.com" },
  { $set: { city: "San Francisco" } }
);
```

✓ **Finds** the user with email "alice@example.com" and **updates** the city field.

4.2 Using updateMany() to Modify Multiple Documents

To update **multiple documents,** use updateMany().

Example: Updating City for All Users in New York

```
db.users.updateMany(
  { city: "New York" },
  { $set: { city: "Brooklyn" } }
);
```

✓ **All users** in "New York" are updated to "Brooklyn".

4.3 Incrementing Numeric Fields Using $inc

Use $inc to **increase** or **decrease** a numeric value.

Example: Increasing Age by 1 Year

```
db.users.updateOne(
  { name: "Bob Smith" },
  { $inc: { age: 1 } }
);
```

✓ Adds 1 to age.

CHAPTER 5: DELETING DOCUMENTS IN MONGODB

5.1 Using deleteOne() to Remove a Single Document

To remove **one document,** use deleteOne().

Example: Deleting a User by Email

```
db.users.deleteOne({ email: "charlie@example.com" });
```

✓ Removes the **first matching document**.

5.2 Using deleteMany() to Remove Multiple Documents

To remove **multiple documents,** use deleteMany().

Example: Deleting All Users from Brooklyn

```
db.users.deleteMany({ city: "Brooklyn" });
```

✓ Deletes **all matching documents**.

Case Study: How Facebook Uses MongoDB for User Data Storage

Background

Facebook manages **billions of user profiles,** requiring a **scalable, high-performance** database solution.

Challenges

- **Handling millions of CRUD operations per second.**
- **Ensuring fast retrieval of user data and posts.**
- **Maintaining low-latency responses for billions of users.**

Solution: Implementing MongoDB for High-Speed Data Storage

✓ **Stored user profiles & posts in MongoDB** for fast document retrieval.
✓ **Used indexing & sharding** to scale data across multiple servers.
✓ **Optimized CRUD operations** to maintain performance at global scale.

This allowed **Facebook to manage billions of real-time interactions** efficiently.

Exercise

1. Insert a new user into the users collection with fields: name, email, age, and city.
2. Retrieve all users who are older than **25 years**.
3. Update the city field for a user with a specific email.
4. Delete a user with the name = "John Doe".

Conclusion

In this section, we explored:
✓ **Creating documents using insertOne() and insertMany().**
✓ **Retrieving documents using findOne(), find(), and query filters.**
✓ **Updating documents with updateOne(), updateMany(), and $inc.**
✓ **Deleting documents using deleteOne() and deleteMany().**
✓ **How Facebook uses MongoDB to handle massive CRUD operations.**

UNDERSTANDING COLLECTIONS AND INDEXES IN MONGODB

CHAPTER 1: INTRODUCTION TO COLLECTIONS AND INDEXES IN MONGODB

1.1 Understanding Collections in MongoDB

MongoDB is a **NoSQL document-based database**, meaning it does not use **tables and rows** like relational databases (SQL). Instead, it uses **collections and documents** to store data efficiently.

✓ **Collections** – A collection is equivalent to a table in SQL, but it stores **JSON-like documents** instead of rows.
✓ **Documents** – Each document within a collection is stored in **BSON format** (Binary JSON), making it compact and fast.
✓ **Schema Flexibility** – Unlike SQL databases, collections in MongoDB do not require **a fixed schema**, meaning different documents in the same collection can have **different fields**.

1.2 Collections vs. Tables in SQL

Feature	MongoDB Collection	SQL Table
Structure	Schema-less, flexible	Fixed schema
Storage Format	BSON (Binary JSON)	Rows & Columns
Scalability	Horizontal (Sharding)	Vertical (Scaling Up)
Joins	No built-in joins	Supports joins
Best For	Big data, real-time applications	Structured, relational data

MongoDB's **collection-based storage** allows it to handle **large-scale, unstructured data** efficiently, making it ideal for **real-time applications and big data processing**.

CHAPTER 2: CREATING AND MANAGING COLLECTIONS IN MONGODB

2.1 Creating a Collection

MongoDB **automatically creates a collection** when a document is inserted. However, you can also create collections manually.

Example: Creating a Collection Manually

```
db.createCollection("users")
```

✓ This creates an empty users collection.

2.2 Inserting Documents into a Collection

To insert data into a collection, use insertOne() or insertMany().

Example: Inserting a Document into the users Collection

```
db.users.insertOne({
  name: "John Doe",
  email: "john@example.com",
  age: 30,
  isAdmin: false
})
```

✓ If the users collection does not exist, MongoDB **creates it automatically**.

To insert multiple documents:

```
db.users.insertMany([
```

```
  { name: "Alice", email: "alice@example.com", age: 25 },
  { name: "Bob", email: "bob@example.com", age: 28 }
])
```

✓ insertMany() allows **batch inserts**, improving performance.

2.3 Retrieving Data from a Collection

To fetch all users:

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

✓ Returns an **array of documents** from the users collection.

To find a specific user:

```
db.users.findOne({ email: "alice@example.com" })
```

✓ Fetches only the **first matching document**.

CHAPTER 3: INDEXING IN MONGODB

3.1 What is an Index?

An **index** is a special data structure that **improves query performance** by allowing MongoDB to locate data **faster**.

✓ Without indexes, MongoDB **scans the entire collection** to find a match.
✓ With indexes, MongoDB **retrieves data directly**, reducing query time.
✓ Indexes improve **read performance** but require **extra storage space**.

Indexes work similarly to an **index in a book** – instead of searching through every page, you jump directly to the relevant section.

CHAPTER 4: CREATING AND MANAGING INDEXES IN MONGODB

4.1 Creating an Index

To create an index on the email field of the users collection:

```
db.users.createIndex({ email: 1 })
```

✓ The 1 means **ascending order** (use -1 for descending).
✓ This makes lookups based on email **significantly faster**.

4.2 Viewing Existing Indexes

To check all indexes in a collection:

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

✓ Displays all **indexes** created in the users collection.

4.3 Removing an Index

To delete an index on the email field:

```
db.users.dropIndex("email_1")
```

✓ This removes the **specific index,** but **does not delete the data**.

CHAPTER 5: TYPES OF INDEXES IN MONGODB

5.1 Single-Field Index

An index on a **single field** improves searches for that specific field.

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

✓ Speeds up queries like:

```
db.users.find({ age: 30 })
```

5.2 Compound Index

A **compound index** is created on **multiple fields,** optimizing queries that filter by **multiple criteria**.

db.users.createIndex({ name: 1, age: -1 })

✓ This index **prioritizes name first,** then sorts by age in **descending order**.

5.3 Unique Index

A **unique index** ensures that no duplicate values exist in a field.

db.users.createIndex({ email: 1 }, { unique: true })

✓ Prevents duplicate **email addresses** in the database.

5.4 TTL Index (Time-To-Live Index)

A **TTL (Time-To-Live) index** automatically deletes documents **after a set time**. Useful for **temporary data** like logs or session tokens.

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

✓ Deletes documents **after 1 hour**.

Case Study: How an E-Commerce Platform Used Indexing for Performance Optimization

Background

An e-commerce company experienced **slow API response times** due to large datasets in the orders collection.

Challenges

✓ Queries for **recent orders** took **over 5 seconds**.
✓ **Database scans** caused high CPU usage.
✓ **High traffic loads** led to system slowdowns.

Solution: Implementing Indexing in MongoDB

The team optimized performance by:

✓ **Creating an index on userId and orderDate** to speed up order lookups.
✓ **Using compound indexes** to optimize multiple filters in a single query.
✓ **Adding a TTL index** on logs to remove outdated data automatically.

Results

- **Query execution time reduced from 5s to 50ms**.
- **CPU usage dropped by 40%,** reducing server costs.
- **Faster response times,** improving customer experience.

By leveraging MongoDB **indexes,** the company significantly **improved performance and scalability**.

Exercise

1. **Create a collection** called products and insert sample documents with fields:
    - name, category, price, and stock.
2. **Create an index on the category field** to optimize searches by category.
3. **Check all existing indexes** in the products collection.

4. **Remove an index** on the category field.

Conclusion

In this section, we explored:

✓ **How MongoDB collections store flexible, schema-less data.**

✓ **How indexes optimize queries by reducing search time.**

✓ **Different types of indexes (single-field, compound, unique, TTL).**

ASSIGNMENT:

SET UP A MONGODB DATABASE, CREATE COLLECTIONS, AND PERFORM BASIC CRUD OPERATIONS.

SOLUTION GUIDE: SET UP A MONGODB DATABASE, CREATE COLLECTIONS, AND PERFORM BASIC CRUD OPERATIONS

Step 1: Set Up a MongoDB Database

1.1 Install and Start MongoDB Locally

For **local MongoDB installation,** follow these steps:

1. Install MongoDB from MongoDB Download Center.
2. Start MongoDB:
3. mongod --dbpath=/data/db
4. Open a new terminal and start the MongoDB shell:
5. mongosh
6. Create a new database:
7. use myDatabase

1.2 Set Up MongoDB Atlas (Cloud Database)

1. **Sign up** at MongoDB Atlas.
2. **Create a cluster** and whitelist your IP address.
3. Copy your **MongoDB connection string**:
4. mongodb+srv://username:password@cluster.mongodb.net/myDatabase

Step 2: Connect a Node.js App to MongoDB

2.1 Install Dependencies

Create a new Node.js project:

mkdir mongodb-crud

cd mongodb-crud

npm init -y

Install required dependencies:

npm install mongoose dotenv

Create a **.env** file to store the MongoDB connection string:

MONGO_URI=mongodb://localhost:27017/myDatabase # For local MongoDB

#
MONGO_URI=mongodb+srv://username:password@cluster.mongodb.net/myDatabase # For MongoDB Atlas

2.2 Connect to MongoDB Using Mongoose

Create **config/db.js**:

```
const mongoose = require('mongoose');

require('dotenv').config();


mongoose.connect(process.env.MONGO_URI, {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(() => console.log('MongoDB Connected'))
```

```
.catch(err => console.error('MongoDB Connection Error:', err));


module.exports = mongoose;
```

✓ **Mongoose handles MongoDB connections efficiently**.
✓ The **.env** file stores database credentials securely.

Step 3: Create a Collection and Define a Schema

3.1 Define a Mongoose Model for Users

Create **models/User.js**:

```
const mongoose = require('mongoose');


const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  age: { type: Number, required: true }
});


module.exports = mongoose.model('User', userSchema);
```

✓ This **schema** defines the **structure** of the "users" collection.
✓ The **email** field is **unique**, preventing duplicate users.

Step 4: Implement CRUD Operations

Create **crud.js** to perform **Create, Read, Update, and Delete (CRUD) operations**.

4.1 Create a New User (INSERT Operation)

```
const mongoose = require('./config/db');

const User = require('./models/User');


const createUser = async () => {
  try {
    const newUser = new User({ name: "Alice", email: "alice@example.com", age: 25 });
    await newUser.save();
    console.log("User Created:", newUser);
  } catch (error) {
    console.error("Error creating user:", error);
  } finally {
    mongoose.connection.close();
  }
};


createUser();
```

✓ This function **creates a new user** and **saves it to MongoDB**.
✓ **Error handling** prevents application crashes.

Run the script:

node crud.js

4.2 Read Users from the Database (FIND Operation)

Modify **crud.js** to fetch users:

```
const readUsers = async () => {
  try {
    const users = await User.find();
    console.log("All Users:", users);
  } catch (error) {
    console.error("Error reading users:", error);
  } finally {
    mongoose.connection.close();
  }
};

readUsers();
```

✓ Fetches all users stored in MongoDB.
✓ Converts MongoDB documents into **JavaScript objects**.

Run the script:

node crud.js

4.3 Update a User's Information (UPDATE Operation)

Modify **crud.js** to update a user's age:

```
const updateUser = async () => {
  try {
    const updatedUser = await User.findOneAndUpdate(
      { email: "alice@example.com" },
      { age: 30 },
      { new: true }
    );
    console.log("Updated User:", updatedUser);
  } catch (error) {
    console.error("Error updating user:", error);
  } finally {
    mongoose.connection.close();
  }
};

updateUser();
```

✓ Updates the **age** field for Alice.
✓ The { new: true } option returns the **updated document**.

Run the script:

```
node crud.js
```

4.4 Delete a User (DELETE Operation)

Modify **crud.js** to delete a user:

```
const deleteUser = async () => {
  try {
    const deletedUser = await User.findOneAndDelete({ email: "alice@example.com" });
    console.log("Deleted User:", deletedUser);
  } catch (error) {
    console.error("Error deleting user:", error);
  } finally {
    mongoose.connection.close();
  }
};

deleteUser();
```

✓ Removes Alice from the **users** collection.

Run the script:

```
node crud.js
```

Case Study: How an E-Commerce Platform Used MongoDB for User Management

Background

An e-commerce startup needed a **scalable database** to handle **user accounts**.

Challenges

- **Data inconsistency** due to unstructured storage.
- **Slow queries** for retrieving user information.
- **Lack of unique user identification,** leading to duplicate records.

Solution: Implementing MongoDB with Mongoose

✓ Used **Mongoose schemas** to enforce **consistent user data**.
✓ Indexed **email fields** to **prevent duplicate users**.
✓ Optimized **queries with indexing,** reducing query time by **60%**.

Results

✓ **Faster user authentication,** improving login speed.
✓ **Scalable database architecture,** handling 1M+ users.
✓ **Increased data reliability,** reducing errors.

This case study highlights how **MongoDB and Mongoose simplify user management** in web applications.

Exercise

1. Create a **new collection** for storing products with fields:
    - name (required)
    - price (required, number)
    - category (default: "General")
    - stock (default: 0)
2. Implement CRUD operations for the **products collection**:
    - Insert a **new product**.
    - Retrieve **all products**.

- Update a **product's price**.
- Delete a **product**.

Conclusion

✓ We **installed and set up MongoDB locally and on MongoDB Atlas**.
✓ We **created collections and defined schemas** using **Mongoose**.
✓ We **performed CRUD operations** to **manage user data**.