**Phase 1: Installation & Setup**

1. **Introduction to MongoDB**
   * What is MongoDB?
   * NoSQL vs. SQL
   * Features & Use Cases
2. **Installing MongoDB on Windows**
   * Downloading MongoDB
   * Installing MongoDB Community Edition
   * Setting up MongoDB as a Windows Service
3. **MongoDB Compass & Shell**
   * Installing MongoDB Compass
   * Using MongoDB Shell (mongosh)

**Phase 1: Installation & Setup** of MongoDB on **Windows**. 🚀

### **1️⃣ Introduction to MongoDB**

#### **What is MongoDB?**

* MongoDB is a **NoSQL database** that stores data in **JSON-like documents**.
* It is **schema-less**, meaning you don't need to define a rigid structure like in SQL databases.
* It is designed to handle **large-scale applications** with high performance, scalability, and flexibility.

#### **NoSQL vs SQL**

|  |  |  |
| --- | --- | --- |
| Feature | SQL (Relational) | NoSQL (MongoDB) |
| **Data Storage** | Tables & Rows | Collections & Documents |
| **Schema** | Fixed Schema | Flexible Schema |
| **Joins** | Uses Foreign Keys | Embedded Documents & References |
| **Scalability** | Vertical Scaling | Horizontal Scaling (Sharding) |

#### **Use Cases of MongoDB**

✅ Real-time analytics  
✅ Content management systems  
✅ E-commerce applications  
✅ IoT & Big Data applications

### **Comparison: Data Storage in MySQL vs MongoDB**

Let’s compare **how data is stored** in **MySQL (Relational DB)** vs **MongoDB (NoSQL Document DB)** using an **Employee Management** example.

### **1️⃣ MySQL (SQL - Relational Database)**

In MySQL, data is stored in **tables** with predefined **columns and relationships**.

#### **Employee Table (SQL)**

|  |  |  |  |
| --- | --- | --- | --- |
| id | name | email | department\_id |
| 101 | Alice | [alice@example.com](mailto:alice@example.com) | 1 |
| 102 | Bob | [bob@example.com](mailto:bob@example.com) | 2 |

#### **Department Table (SQL)**

|  |  |
| --- | --- |
| department\_id | department\_name |
| 1 | HR |
| 2 | IT |

#### **Retrieving Employee with Department Name (SQL Query)**

SELECT e.id, e.name, e.email, d.department\_name

FROM Employee e

JOIN Department d ON e.department\_id = d.department\_id

WHERE e.id = 101;

**Problem with SQL Approach**

* Data is **split across multiple tables**, requiring **JOINS**.
* Schema is **rigid**, meaning if we want to add a new field (e.g., phone number), we must **alter the table**.

### **2️⃣ MongoDB (NoSQL - Document Database)**

In MongoDB, data is stored as **JSON-like documents** inside **collections** instead of tables.

#### **Employee Collection (MongoDB)**

{

"\_id": 101,

"name": "Alice",

"email": "alice@example.com",

"department": {

"department\_id": 1,

"department\_name": "HR"

}

}

#### **Retrieving Employee with Department (MongoDB Query)**

db.employees.findOne({ \_id: 101 })

**Advantages of MongoDB Approach** ✅ **No Need for JOINS** – The department is stored inside the employee document.  
✅ **Flexible Schema** – We can add new fields (e.g., phone number) **without altering a schema**.  
✅ **Better Performance for Reads** – Since data is stored together, fetching an employee’s department is **faster**.

### **Comparison Summary**

|  |  |  |
| --- | --- | --- |
| Feature | MySQL (SQL) | MongoDB (NoSQL) |
| **Storage Format** | Tables (Rows & Columns) | Collections (JSON Documents) |
| **Schema** | Fixed Schema (Needs ALTER) | Flexible Schema (Dynamic) |
| **Relationships** | Foreign Keys & JOINS | Embedded Documents (No Joins) |
| **Querying** | SQL Queries (JOINs) | JSON-based Queries (BSON) |
| **Scalability** | Vertical Scaling (Single Server) | Horizontal Scaling (Sharding) |

### **🚀 When to Choose What?**

* **Use MySQL** if your data is **highly structured** with **complex relationships** (e.g., banking, ERP).
* **Use MongoDB** if you need **scalability, flexibility**, and work with **large, unstructured data** (e.g., real-time apps, logs, IoT).

### **2️⃣ Installing MongoDB on Windows**

#### **Step 1: Download MongoDB**

1. Go to the [MongoDB Download Center](https://www.mongodb.com/try/download/community).
2. Select **Windows**, then choose **MSI package**.
3. Click **Download**.

#### **Step 2: Install MongoDB**

1. Open the downloaded .msi file.
2. Follow the installation wizard:
   * Select **Complete Installation**.
   * Enable **MongoDB as a Windows Service** (Recommended).
   * Keep default paths unless you prefer custom installation.

#### **Step 3: Verify Installation**

1. Open **Command Prompt (cmd)** and type:
2. mongod --version

If installed correctly, it should display the MongoDB version.

### **3️⃣ Starting & Stopping MongoDB**

#### **Start MongoDB as a Service**

MongoDB runs as a background service automatically.  
If not running, start it manually:

net start MongoDB

#### **Stop MongoDB Service**

net stop MongoDB

#### **Run MongoDB Manually**

1. Open a terminal and type:
2. mongod
3. Open a new terminal and type:
4. mongosh

If successful, you’ll enter the MongoDB shell.

### **4️⃣ MongoDB Compass & Shell**

#### **MongoDB Compass (GUI)**

* **MongoDB Compass** is a GUI tool to visualize your database.
* Download it from [MongoDB Compass](https://www.mongodb.com/products/compass) and install it.
* Open Compass and **connect to MongoDB** (mongodb://localhost:27017).

#### **MongoDB Shell (**mongosh**)**

* The command-line interface for MongoDB.
* Run:
* mongosh

and start interacting with MongoDB.

**Phase 2: Basic MongoDB Operations**

1. **Understanding MongoDB Database & Collections**
   * Creating & Dropping Databases
   * Creating & Dropping Collections
2. **CRUD Operations**
   * Insert Documents
   * Read (Find) Documents
   * Update Documents
   * Delete Documents
3. **MongoDB Query Operators**
   * Comparison Operators ($eq, $ne, $gt, $lt, etc.)
   * Logical Operators ($and, $or, $not, etc.)
   * Projection & Sorting
4. **Indexing in MongoDB**
   * What is an Index?
   * Creating & Managing Indexes

### **Understanding MongoDB Databases & Collections**

MongoDB organizes data into **databases**, and each database contains multiple **collections** (like tables in SQL, but without a strict schema). Each collection stores **documents** (JSON-like objects).

#### **1️⃣ Creating & Dropping Databases** (**Real-World Banking Example)**

In a banking system, we might need a database called **bankDB** to store customer, account, and transaction details.

##### **📌 Creating a Database**

To create or switch to a new database, use:

use bankDB

📌 **MongoDB will create the database only when you insert data into it.**

##### **📌 Checking Available Databases**

show dbs

💡 **Note:** If bankDB is empty, it won’t appear in show dbs.

##### **📌 Dropping a Database**

To delete the bankDB database:

use bankDB

db.dropDatabase()

🚨 **Warning:** This removes all collections and data inside the database!

#### **2️⃣ Creating & Dropping Collections**

Collections in MongoDB store related documents.

##### **📌 Creating Collections**

We need collections like customers, accounts, and transactions for our banking system.

1️⃣ **Create a customers collection manually:**

db.createCollection("customers")

2️⃣ **Insert data into accounts (collection auto-created):**

db.accounts.insertOne({ accountNumber: "1234567890", type: "Savings", balance: 5000 })

💡 **MongoDB creates the collection automatically when you insert a document.**

##### **📌 Viewing Collections**

show collections

##### **📌 Dropping Collections**

To remove the customers collection:

db.customers.drop()

🚨 **Warning:** This removes all documents in the collection!

### **CRUD Operations in MongoDB** 🏦

Now that we have the bankDB database with collections like customers, accounts, and transactions, let's perform **CRUD (Create, Read, Update, Delete) operations**.

## ****1️⃣ Insert Documents (CREATE Operation)****

We insert data into collections using insertOne() (single document) and insertMany() (multiple documents).

### **🏦 Example: Insert Customers**

db.customers.insertOne({

customerId: "C1001",

name: "Dineshkumar Thangavel",

email: "dk@example.com",

phone: "+919876543210",

address: { city: "Chennai", zip: "600001" }

})

✅ **A new customer is added to the customers collection.**

### **🏦 Insert Multiple Customers**

db.customers.insertMany([

{

customerId: "C1002",

name: "Divya Dineshkumar",

email: "dd@example.com",

phone: "+919812345678",

address: { city: "Chennai", zip: "600001" }

},

{

customerId: "C1003",

name: "Darwin Divya Dinesh",

email: "ddd@example.com",

phone: "+919700123456",

address: { city: "Chennai", zip: "600001" }

}

])

✅ **Now we have three customers in the database.**

## ****2️⃣ Read Documents (READ Operation)****

To **fetch data**, we use find().

### **🏦 Retrieve All Customers**

db.customers.find()

📌 **This returns all documents.**

### **🏦 Retrieve a Specific Customer**

db.customers.findOne({ customerId: "C1002" })

✅ **Finds Bob Williams' details.**

### **🏦 Retrieve Only Names & Emails (Projection)**

db.customers.find({}, { name: 1, email: 1, \_id: 0 })

📌 **This returns only name and email, excluding \_id.**

## ****3️⃣ Update Documents (UPDATE Operation)****

We use updateOne() and updateMany().

### **🏦 Update a Customer's Email**

db.customers.updateOne(

{ customerId: "C1001" },

{ $set: { email: "dk.new@example.com" } }

)

✅ **Alice's email is updated.**

### **🏦 Update Multiple Customers in the Same City**

db.customers.updateMany(

{ "address.city": "Hyderabad" },

{ $set: { "address.zip": "60602" } }

)

✅ **All customers in Chicago get their ZIP code updated.**

🏦 Rename a Field Name

db.customers.updateOne(

{ customerId: "C1001" },

{ $rename: { "nickname": "alias", "cell": "mobile" } }

)

✅ Customer's fields are renamed.

🏦 Rename the Collection

db.customers.renameCollection("clients")

✅ The `customers` collection is renamed to `clients`.

## ****4️⃣ Delete Documents (DELETE Operation)****

We use deleteOne() and deleteMany().

### **🏦 Delete a Specific Customer**

db.customers.deleteOne({ customerId: "C1003" })

✅ **Charlie Brown is removed.**

### **🏦 Delete All Customers in a Specific City**

db.customers.deleteMany({ "address.city": "Chennai" })

✅ **All customers from Los Angeles are deleted.**

## ****MongoDB Query Operators with Real-Time Banking Example**** 🏦

Now, let's explore **Comparison Operators, Logical Operators, Projection, and Sorting** using the bankDB database.

## ****1️⃣ Comparison Operators**** 🔍

These help filter documents based on conditions.

### **📌** $in **– Match Any Value in an Array**

Find all customers in **New York or Los Angeles**.

db.customers.find({ "address.city": { $in: ["New York", "Los Angeles"] } })

✅ Returns customers from New York or LA.

### **📌** $nin **– Not in the Given Values**

Find customers **NOT** in Chicago or Houston.

db.customers.find({ "address.city": { $nin: ["Chicago", "Houston"] } })

✅ Excludes customers from these cities.

### **📌** $gt **(Greater Than) &** $gte **(Greater Than or Equal To)**

Find all **accounts with a balance greater than $5000**.

db.accounts.find({ balance: { $gt: 5000 } })

✅ Only returns accounts with balance above $5000.

Find **accounts with balance at least $5000**.

db.accounts.find({ balance: { $gte: 5000 } })

✅ Includes accounts with balance = $5000 too.

### **📌** $lt **(Less Than) &** $lte **(Less Than or Equal To)**

Find **transactions below $1000**.

db.transactions.find({ amount: { $lt: 1000 } })

✅ Retrieves transactions with amount < 1000.

Find **transactions of $1000 or less**.

db.transactions.find({ amount: { $lte: 1000 } })

✅ Includes $1000 transactions.

### **📌** $eq **(Equal) &** $ne **(Not Equal)**

Find all **savings accounts**.

db.accounts.find({ accountType: { $eq: "Savings" } })

✅ Returns all Savings accounts.

Find all **non-Savings accounts**.

db.accounts.find({ accountType: { $ne: "Savings" } })

✅ Returns all except Savings accounts.

## ****2️⃣ Logical Operators**** 🧠

These help combine multiple conditions.

### **📌** $and **– Both Conditions Must be True**

Find **transactions above $5000** made by a customer from **New York**.

db.transactions.find({

$and: [{ amount: { $gt: 5000 } }, { "customer.city": "New York" }]

})

✅ Returns high-value transactions from NY customers.

### **📌** $or **– Either Condition Can be True**

Find **customers from New York or customers with a balance above $10,000**.

db.customers.find({

$or: [{ "address.city": "New York" }, { balance: { $gt: 10000 } }]

})

✅ Returns either NY customers or customers with a high balance.

### **📌** $not **– Negate a Condition**

Find customers **NOT from New York**.

db.customers.find({ "address.city": { $not: { $eq: "New York" } } })

✅ Returns everyone except New Yorkers.

### **📌** $exists **– Check if a Field Exists**

Find **all customers who have an email address**.

db.customers.find({ email: { $exists: true } })

✅ Only customers with an email will be returned.

## ****3️⃣ Projection – Return Specific Fields**** 📜

By default, MongoDB returns full documents. **Projection** lets us return only specific fields.

### **📌 Show Only Name and Email (Exclude** \_id**)**

db.customers.find({}, { name: 1, email: 1, \_id: 0 })

✅ Returns documents with **only** name and email.

### **📌 Hide Address Field**

db.customers.find({}, { address: 0 })

✅ Returns everything **except** address.

## ****4️⃣ Sorting – Order Results**** 🔽🔼

### **📌 Sort by Balance in Descending Order**

db.accounts.find().sort({ balance: -1 })

✅ Highest balance first.

### **📌 Sort by Name in Ascending Order**

db.customers.find().sort({ name: 1 })

✅ Alphabetical order (A-Z).

### **📌 Sort by City Ascending, then Balance Descending**

db.customers.find().sort({ "address.city": 1, balance: -1 })

✅ Groups customers by **city**, and within each city, sorts by **highest balance first**.

**Phase 3: Advanced MongoDB Concepts**

1. **Indexing**
2. **Aggregation Framework**
   * Understanding Aggregations
   * Using $match, $group, $sort, $project, $limit
3. **Relationships in MongoDB**
   * Embedded vs. Referenced Documents
   * Implementing One-to-One, One-to-Many, and Many-to-Many
4. **Transactions in MongoDB**

* ACID Transactions in MongoDB
* Multi-document Transactions

**📌 Step 1: Indexing in MongoDB**

#### **What is Indexing?**

* **Indexing** in MongoDB is similar to an **index in a book**—it speeds up data retrieval.
* Without indexes, MongoDB must scan every document in a collection, which is **slow** for large datasets.
* Indexes store a **sorted** list of values from a field, making lookups faster.

#### **Types of Indexes in MongoDB**

1. **Single Field Index** – Index on a single field.
2. **Compound Index** – Index on multiple fields.
3. **Text Index** – For searching text efficiently.
4. **Hashed Index** – Distributes values for sharding.
5. **Geospatial Index** – For location-based queries.

### **🛠 Practical Example: Indexing on Banking Data**

Let's **create an index** on the phone field in the **customers collection** to speed up customer lookups.

#### **1️⃣ Create a Single Field Index**

db.customers.createIndex({ "phone": 1 })

📌 **This will create an index on the phone field in ascending order (1 = ascending, -1 = descending).**

#### **2️⃣ Check Existing Indexes**

db.customers.getIndexes()

🔍 **This will list all indexes in the customers collection.**

#### **3️⃣ Create a Compound Index**

Suppose we often search for customers using both **city** and **state**. We can optimize this with a compound index.

db.customers.createIndex({ "address.city": 1, "address.state": 1 })

📌 **Now MongoDB can efficiently search for customers in a specific city & state.**

#### **4️⃣ Use Index in Queries**

**Without an index:**

db.customers.find({ "phone": "+919876543210" }).explain("executionStats")

**With an index:** 🔥

db.customers.find({ "phone": "+919876543210" }).hint({ "phone": 1 }).explain("executionStats")

📌 **You will notice a significant performance improvement!**

#### **5️⃣ Delete an Index**

db.customers.dropIndex("phone\_1")

📌 **This removes the index on the phone field.**

### **✅ Summary**

* **Indexes improve query performance** by reducing scan time.
* **Use Single & Compound indexes** based on query patterns.
* **Check & manage indexes** using getIndexes(), explain(), and dropIndex().

## ****📌 Advanced Indexing Concepts in MongoDB****

### **1️⃣ Unique Index**

* Ensures that **duplicate values** are not inserted in a specific field.
* Useful for fields like **phone numbers, email IDs, account numbers, etc.** in a banking system.

#### **✅ Create a Unique Index**

db.customers.createIndex({ "phone": 1 }, { unique: true })

🔹 **Now, MongoDB will prevent duplicate phone numbers from being inserted.**

#### **⛔ Trying to Insert a Duplicate Phone Number**

db.customers.insertOne({ "name": "Rahul", "phone": "+919876543210" })

📌 **If a document with this phone already exists, MongoDB will throw an error!**

### **2️⃣ Naming an Index**

* By default, MongoDB assigns index names like {fieldname}\_1, but we can give **custom names**.

#### **✅ Create an Index with a Custom Name**

db.customers.createIndex({ "email": 1 }, { name: "UniqueEmailIndex" })

🔹 **Easier to manage and drop later!**

#### **Check Existing Indexes**

db.customers.getIndexes()

📌 **You'll see your index with the custom name.**

### **3️⃣ TTL (Time-to-Live) Index**

* **Automatically deletes documents** after a certain period.
* Useful for **temporary records**, such as **OTP verifications, session data, or logs**.

#### **✅ Create a TTL Index**

db.otp\_requests.createIndex({ "createdAt": 1 }, { expireAfterSeconds: 300 })

🔹 **Documents in otp\_requests will be deleted after 5 minutes (300 seconds).**

### **4️⃣ Partial Filter Index**

* **Indexes only documents that meet a condition.**
* Saves storage and improves performance for specific queries.

#### **✅ Create a Partial Index**

db.transactions.createIndex({ "status": 1 }, { partialFilterExpression: { "status": { $eq: "Pending" } } })

🔹 **Only transactions with "status": "Pending" will be indexed.**  
🔹 **This speeds up searches on pending transactions while reducing index size.**

### **5️⃣ Wildcard Index**

* Useful when the **fields are dynamic** (e.g., JSON-based schemas in NoSQL).
* Instead of defining **specific fields**, we can index **all fields**.

#### **✅ Create a Wildcard Index**

db.logs.createIndex({ "$\*\*": 1 })

🔹 **Indexes all fields dynamically.**  
🔹 Useful for **searching in flexible, schema-less collections.**

### **6️⃣ Custom Collation (Case-Insensitive Indexing)**

* Used for **sorting & searching case-insensitively**.
* In banking apps, we may want **case-insensitive email lookups**.

#### **✅ Create a Case-Insensitive Index**

db.customers.createIndex(

{ "email": 1 },

{ collation: { locale: "en", strength: 2 } }

)

🔹 **Now searches on email will be case-insensitive.**  
🔹 "JohnDoe@gmail.com" and "johndoe@gmail.com" are treated as the same.

### **7️⃣ Sparse Index**

* **Skips indexing null or missing values**, reducing index size.
* Useful for **optional fields** (e.g., PAN Card numbers in customer profiles).

#### **✅ Create a Sparse Index**

db.customers.createIndex({ "pan\_card": 1 }, { sparse: true })

🔹 **Documents without pan\_card won't be indexed.**  
🔹 **Saves space & improves efficiency for optional fields.**

## ****✅ Summary****

|  |  |
| --- | --- |
| Index Type | Use Case Example |
| **Unique Index** | Prevents duplicate emails or phone numbers. |
| **Named Index** | Easier index management with custom names. |
| **TTL Index** | Auto-delete expired OTPs or session data. |
| **Partial Filter Index** | Index only pending transactions for faster queries. |
| **Wildcard Index** | Index all fields dynamically in logs or flexible schemas. |
| **Custom Collation** | Case-insensitive email lookups. |
| **Sparse Index** | Saves space by indexing only non-null values. |

### **🔹 MongoDB Aggregation with Banking Database (Detailed Explanation)**

## ****🚀 Aggregation Pipeline Operators****

Here are the most important operators with **banking database examples**:

1️⃣ **$match** – Filter documents (like SQL WHERE)  
2️⃣ **$lookup** – Join collections (like SQL JOIN)  
3️⃣ **$unwind** – Flatten arrays (like SQL UNNEST)  
4️⃣ **$group** – Group data (like SQL GROUP BY)  
5️⃣ **$sort** – Sort results (like SQL ORDER BY)  
6️⃣ **$skip** – Skip records (like SQL OFFSET)  
7️⃣ **$limit** – Limit records (like SQL LIMIT)  
8️⃣ **$project** – Restructure fields (like SQL SELECT)

## ****1️⃣**** $match ****- Filter Transactions Greater than ₹1000****

Filters transactions where amount > 1000 (like WHERE in SQL).

db.transactions.aggregate([

{ $match: { amount: { $gt: 1000 } } }

])

✅ **SQL Equivalent:**

SELECT \* FROM transactions WHERE amount > 1000;

## ****2️⃣**** $lookup ****- Join Customers with Their Accounts****

This joins customers with their accounts (like JOIN in SQL).

db.customers.aggregate([

{

$lookup: {

from: "accounts",

localField: "\_id",

foreignField: "customer\_id",

as: "accounts"

}

}

])

✅ **SQL Equivalent:**

SELECT \* FROM customers

JOIN accounts ON customers.\_id = accounts.customer\_id;

## ****3️⃣**** $unwind ****- Flatten Accounts for Each Customer****

Since $lookup returns an **array**, $unwind flattens it.

db.customers.aggregate([

{ $lookup: { from: "accounts", localField: "\_id", foreignField: "customer\_id", as: "accounts" } },

{ $unwind: "$accounts" }

])

✅ **SQL Equivalent:**

SELECT customers.\*, accounts.\* FROM customers

JOIN accounts ON customers.\_id = accounts.customer\_id;

## ****4️⃣**** $group ****- Total Balance Per Customer****

Groups transactions by customer\_id and sums the balance.

db.accounts.aggregate([

{ $group: { \_id: "$customer\_id", totalBalance: { $sum: "$balance" } } }

])

✅ **SQL Equivalent:**

SELECT customer\_id, SUM(balance) AS totalBalance

FROM accounts

GROUP BY customer\_id;

## ****5️⃣**** $sort ****- Sort Customers by Balance (Descending)****

Sorts customers by their total balance from highest to lowest.

db.accounts.aggregate([

{ $sort: { balance: -1 } }

])

✅ **SQL Equivalent:**

SELECT \* FROM accounts ORDER BY balance DESC;

## ****6️⃣**** $skip ****- Skip First 2 Transactions****

Skips the first 2 transactions (useful for pagination).

db.transactions.aggregate([

{ $sort: { amount: -1 } },

{ $skip: 2 }

])

✅ **SQL Equivalent:**

SELECT \* FROM transactions ORDER BY amount DESC OFFSET 2;

## ****7️⃣**** $limit ****- Get Top 3 Transactions****

Limits the result to **3 transactions only**.

db.transactions.aggregate([

{ $sort: { amount: -1 } },

{ $limit: 3 }

])

✅ **SQL Equivalent:**

SELECT \* FROM transactions ORDER BY amount DESC LIMIT 3;

## ****8️⃣**** $project ****- Select Only Required Fields****

Selects only accountNumber, amount, and type.

db.transactions.aggregate([

{ $project: { accountNumber: 1, amount: 1, type: 1, \_id: 0 } }

])

✅ **SQL Equivalent:**

SELECT accountNumber, amount, type FROM transactions;

### **🚀 Summary**

|  |  |  |
| --- | --- | --- |
| Operator | Purpose | SQL Equivalent |
| $match | Filters data | WHERE |
| $lookup | Joins collections | JOIN |
| $unwind | Flattens arrays | UNNEST |
| $group | Groups data | GROUP BY |
| $sort | Sorts results | ORDER BY |
| $skip | Skips records | OFFSET |
| $limit | Limits records | LIMIT |
| $project | Restructures fields | SELECT |

# Aggregation Questions and Queries

**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

}

}

])

**Advanced aggregation concepts**

### **1. $facet (Multi-Pipeline Processing)**

* **Use case:** Get multiple results from a single query, such as total customers, high-value customers, and accounts per branch.

**Example:**

db.customers.aggregate([

{

$facet: {

totalCustomers: [{ $count: "count" }], // Count total customers

highValueCustomers: [

{ $match: { totalBalance: { $gt: 1000000 } } },

{ $project: { name: 1, totalBalance: 1, \_id: 0 } }

],

accountsPerBranch: [

{ $group: { \_id: "$branch\_id", totalAccounts: { $sum: 1 } } }

]

}

}

])

🔹 **Explanation:** This query:

* Counts the **total customers**.
* Finds **high-value customers** (with a balance > ₹10 Lakhs).
* Groups customers by **branch\_id** and counts the number of accounts per branch.

### **2. $bucket (Range-Based Grouping)**

* **Use case:** Categorize customers into balance ranges (low, medium, high balance).

**Example:**

db.customers.aggregate([

{

$bucket: {

groupBy: "$totalBalance",

boundaries: [0, 50000, 200000, 1000000, Infinity],

default: "Unknown",

output: {

count: { $sum: 1 },

customers: { $push: "$name" }

}

}

}

])

🔹 **Explanation:** This groups customers into different balance **ranges**:

* 0-50K
* 50K-2L
* 2L-10L
* 10L+

### **3. $bucketAuto (Dynamic Binning)**

* **Use case:** Automatically group customers based on balance, dividing them into equal-sized buckets.

**Example:**

db.customers.aggregate([

{

$bucketAuto: {

groupBy: "$totalBalance",

buckets: 4,

output: { count: { $sum: 1 }, avgBalance: { $avg: "$totalBalance" } }

}

}

])

🔹 **Explanation:** Instead of fixed ranges like $bucket, MongoDB **automatically** determines 4 **equal-sized** groups.

### **4. $addFields (Add Computed Fields)**

* **Use case:** Add a new field (status) based on balance.

**Example:**

db.customers.aggregate([

{

$addFields: {

status: {

$cond: { if: { $gte: ["$totalBalance", 500000] }, then: "VIP", else: "Regular" }

}

}

}

])

🔹 **Explanation:** If a customer has **₹5L+**, they are tagged as **VIP**, otherwise **Regular**.

### **5. $redact (Access Control on Nested Documents)**

* **Use case:** Hide account details unless the balance is greater than ₹1L.

**Example:**

db.accounts.aggregate([

{

$redact: {

$cond: {

if: { $gt: ["$balance", 100000] },

then: "$$KEEP",

else: "$$PRUNE"

}

}

}

])

🔹 **Explanation:**

* **Keeps** accounts **only if balance > ₹1L**.
* **Hides** other documents.

### **6. $setWindowFields (Running Totals & Moving Averages)**

* **Use case:** Calculate a running balance for a customer's transactions.

**Example:**

db.transactions.aggregate([

{

$setWindowFields: {

partitionBy: "$account\_id",

sortBy: { date: 1 },

output: {

runningBalance: {

$sum: "$amount",

window: { documents: ["unbounded", "current"] }

}

}

}

}

])

🔹 **Explanation:**

* Calculates **running balance** per account\_id, sorted by date.

### **7. $merge (Store Aggregation Results in a New Collection)**

* **Use case:** Store high-value customers in a separate collection.

**Example:**

db.customers.aggregate([

{ $match: { totalBalance: { $gt: 1000000 } } },

{ $out: "high\_value\_customers" }

])

🔹 **Explanation:**

* Stores customers with **balance > ₹10L** in high\_value\_customers.

## ****Transactions in MongoDB****

### **1️⃣ What Are Transactions?**

* Transactions in MongoDB **ensure ACID (Atomicity, Consistency, Isolation, Durability)** properties.
* They allow **multiple operations across multiple documents and collections** to be executed as a single unit.
* If any operation fails, the **entire transaction is rolled back**, ensuring **data consistency**.

### **2️⃣ When to Use Transactions?**

* **Banking Applications** (e.g., transferring money from one account to another)
* **E-commerce Orders** (e.g., updating stock inventory & customer orders together)
* **Multi-Document Updates** that must stay consistent across collections

## ****Step 1: Start a Transaction****

Transactions are supported only in **replica sets** or **sharded clusters**. If using a standalone MongoDB instance, transactions won't work.

### **Example: Transferring Money Between Two Accounts**

👉 Let’s assume we have two collections:

1. **accounts** (stores user balances)
2. **transactions** (logs transactions)

MongoDB Atlas: Basics & Transactions in Mongo Shell

## 1️⃣ Introduction to MongoDB Atlas

MongoDB Atlas is a cloud-based **Database as a Service (DBaaS)** that provides a fully managed MongoDB deployment. It simplifies database setup, scaling, and management.

### 🔹 Why Use MongoDB Atlas?

✅ Fully managed cloud service (no need to set up a local environment)

✅ Built-in security, backups, and monitoring

✅ Supports **replica sets** by default (needed for transactions)

✅ Easier to collaborate with teams

### 🔹 Setting Up MongoDB Atlas

1️⃣ **Sign up for MongoDB Atlas** at <https://www.mongodb.com/atlas>.

2️⃣ **Create a new cluster:** Choose a free-tier cluster to get started.

3️⃣ **Connect to your cluster:**

* Go to "Connect" in your cluster.
* Select **MongoDB Shell** and copy the connection string.
* Paste and run the connection string in your command line to connect.

## ****Example 1: Money Transfer Between Two Accounts****

✅ **Scenario**: Transfer **₹500** from **ACC001** (Customer CUST001) to **ACC002** (Customer CUST002).

### **Transaction Steps**

1️⃣ Deduct **₹500** from ACC001 (Sender).  
2️⃣ Add **₹500** to ACC002 (Receiver).  
3️⃣ Record the transaction in the transactions collection.

### **Mongo Shell Code**

// Connect to the MongoDB session

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

session.startTransaction();

try {

  const accountsCollection = session.getDatabase("banking").accounts;

  const transactionsCollection = session.getDatabase("banking").transactions;

  // 1️⃣ Transfer ₹500 from ACC001 to ACC002

  let account1 = accountsCollection.findOne({ accountId: "ACC001" });

  let account2 = accountsCollection.findOne({ accountId: "ACC002" });

  if (account1.balance >= 500) {

    accountsCollection.updateOne(

      { accountId: "ACC001" },

      { $inc: { balance: -500 } }

    );

    accountsCollection.updateOne(

      { accountId: "ACC012" },

      { $inc: { balance: 500 } }

    );

    transactionsCollection.insertOne({

      transactionId: "TXN001",

      fromAccountId: "ACC001",

      toAccountId: "ACC002",

      amount: 500,

      transactionType: "Transfer",

      date: new Date(),

    });

  } else {

    throw "Insufficient balance in ACC001!";

  }

  // 2️⃣ Loan Payment - Transfer ₹1000 from ACC002 to LOAN\_ACC

  let loanAccount = accountsCollection.findOne({ accountId: "LOAN\_ACC" });

  if (account2.balance >= 1000) {

    accountsCollection.updateOne(

      { accountId: "ACC002" },

      { $inc: { balance: -1000 } }

    );

    accountsCollection.updateOne(

      { accountId: "LOAN\_ACC" },

      { $inc: { balance: 1000 } }

    );

    transactionsCollection.insertOne({

      transactionId: "TXN003",

      fromAccountId: "ACC002",

      toAccountId: "LOAN\_ACC",

      amount: 1000,

      transactionType: "Loan Payment",

      date: new Date(),

    });

  } else {

    throw "Insufficient balance in ACC002 for Loan Payment!";

  }

  // 3️⃣ Simultaneous Transactions (Isolation Demonstration)

  let concurrentTransaction1 = accountsCollection.updateOne(

    { accountId: "ACC002" },

    { $inc: { balance: -1000 } }

  );

  let concurrentTransaction2 = accountsCollection.updateOne(

    { accountId: "ACC003" },

    { $inc: { balance: 2000 } }

  );

  transactionsCollection.insertMany([

    {

      transactionId: "TXN010",

      fromAccountId: "ACC002",

      toAccountId: "LOAN\_ACC",

      amount: 1000,

      transactionType: "Loan Payment",

      date: new Date(),

    },

    {

      transactionId: "TXN011",

      fromAccountId: "ACC002",

      toAccountId: "ACC003",

      amount: 2000,

      transactionType: "Withdrawal",

      date: new Date(),

    },

  ]);

  // ✅ Commit Transaction

  session.commitTransaction();

  print("Transaction committed successfully!");

} catch (error) {

  // ❌ Rollback Transaction if any error occurs

  session.abortTransaction();

  print("Transaction aborted due to error:", error);

} finally {

  session.endSession();

}

📌 **Expected Outcome**:

## ****Transaction committed successfully!****

## ****true****

## ****Example 2: Loan Payment Deduction****

✅ **Scenario**: Customer **CUST002 (Priya Verma)** repays a **loan installment of ₹1000**.

### **Transaction Steps**

1️⃣ Deduct **₹1000** from **CUST002’s balance**.  
2️⃣ Add **₹1000** to the **bank's loan recovery account** (LOAN\_ACC).  
3️⃣ Record the **loan repayment transaction**.

### **Mongo Shell Code**

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

session.startTransaction();

try {

  const db = session.getDatabase("bankDB");

  // Deduct ₹1000 from Priya Verma (CUST002)

  db.customers.updateOne(

    { customerId: "CUST002" },

    { $inc: { balance: -1000 } }

  );

  // Add ₹1000 to Loan Account

  db.accounts.updateOne({ accountId: "LOAN\_ACC" }, { $inc: { balance: 1000 } });

  // Log Loan Payment Transaction

  db.transactions.insertOne({

    transactionId: new ObjectId(), // Generate unique transaction ID

    fromCustomerId: "CUST002",

    toAccountId: "LOAN\_ACC",

    amount: 1000,

    transactionType: "Loan Payment",

    date: new Date(),

    status: "Success",

  });

  session.commitTransaction();

  session.endSession();

  print("✅ Loan Payment Successful!");

} catch (error) {

  session.abortTransaction();

  session.endSession();

  print("❌ Loan Payment Failed: " + error.message);

}

📌 **Expected Outcome**:

## ****✅ Loan Payment Successful!****

## ****True****

## ****Example 3: Insufficient Funds Rollback****

❌ **Scenario**: Customer **CUST001 (ACC001)** tries to withdraw **₹5000**, but has only **₹100**.

### **Transaction Steps**

1️⃣ Check **if the balance is sufficient**.  
2️⃣ If **insufficient funds**, abort the transaction.  
3️⃣ If sufficient, deduct amount and log withdrawal.

### **Mongo Shell Code**

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

session.startTransaction();

try {

// Check balance of ACC001

let sender = session.getDatabase("bankDB").accounts.findOne({ accountId: "ACC001" });

if (sender.balance < 5000) {

print("❌ Insufficient Funds! Rolling back transaction.");

session.abortTransaction();

session.endSession();

} else {

// Deduct ₹5000 from ACC001

session.getDatabase("bankDB").accounts.updateOne(

{ accountId: "ACC001" },

{ $inc: { balance: -5000 } }

);

// Log Withdrawal

session.getDatabase("bankDB").transactions.insertOne({

transactionId: "TXN004",

fromAccountId: "ACC001",

amount: 5000,

transactionType: "Withdrawal",

date: new Date(),

status: "Success"

});

session.commitTransaction();

session.endSession();

print("✅ Withdrawal Successful!");

}

} catch (error) {

session.abortTransaction();

session.endSession();

print("❌ Withdrawal Failed: " + error);

}

📌 **Expected Outcome**:

* ❌ **Transaction is aborted** due to **insufficient balance**.
* ✅ No changes made to the database.

### **Summary**

✔ **Example 1: Money Transfer** → Deduct from sender, add to receiver, log transaction.  
✔ **Example 2: Loan Payment** → Deduct from customer, add to bank loan account, log repayment.  
✔ **Example 3: Insufficient Funds Rollback** → Check balance, rollback if insufficient.

### **ACID Properties in MongoDB Transactions (Using Banking Example)**

MongoDB supports **ACID (Atomicity, Consistency, Isolation, Durability)** properties for multi-document transactions. Let's analyze how these properties apply to our **banking transactions**.

## ****1️⃣ Atomicity (All or Nothing)****

**Definition:** A transaction is **atomic** if **all operations succeed together** or **none execute** (rollback on failure).

### **Example: Money Transfer (₹500 from ACC001 to ACC002)**

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

session.startTransaction();

try {

// Deduct ₹500 from ACC001

session.getDatabase("bankDB").accounts.updateOne(

{ accountId: "ACC001" },

{ $inc: { balance: -500 } }

);

// Add ₹500 to ACC002

session.getDatabase("bankDB").accounts.updateOne(

{ accountId: "ACC002" },

{ $inc: { balance: 500 } }

);

// Log the transaction

session.getDatabase("bankDB").transactions.insertOne({

transactionId: "TXN002",

fromAccountId: "ACC001",

toAccountId: "ACC002",

amount: 500,

transactionType: "Transfer",

date: new Date(),

status: "Success"

});

session.commitTransaction();

session.endSession();

print("✅ Transaction Successful!");

} catch (error) {

session.abortTransaction();

session.endSession();

print("❌ Transaction Failed: " + error);

}

### **Atomicity Ensured:**

✅ If **all three operations succeed**, the transaction is committed.  
❌ If **even one fails**, the entire transaction rolls back.

For example, if the **balance deduction on ACC001 fails**, then:

* **No money is deducted from ACC001**.
* **No money is credited to ACC002**.
* **No transaction is logged**.

Thus, **atomicity** ensures that a **partial transfer does not happen**.

## ****2️⃣ Consistency (Database Integrity Maintained)****

**Definition:** The database remains **consistent** before and after a transaction. No invalid state occurs.

### **Example: Loan Payment of ₹1000**

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

session.startTransaction();

try {

// Deduct ₹1000 from Priya (CUST002)

session.getDatabase("bankDB").customers.updateOne(

{ customerId: "CUST002" },

{ $inc: { balance: -1000 } }

);

// Add ₹1000 to Loan Account

session.getDatabase("bankDB").accounts.updateOne(

{ accountId: "LOAN\_ACC" },

{ $inc: { balance: 1000 } }

);

// Log Loan Payment Transaction

session.getDatabase("bankDB").transactions.insertOne({

transactionId: "TXN003",

fromCustomerId: "CUST002",

toAccountId: "LOAN\_ACC",

amount: 1000,

transactionType: "Loan Payment",

date: new Date(),

status: "Success"

});

session.commitTransaction();

session.endSession();

print("✅ Loan Payment Successful!");

} catch (error) {

session.abortTransaction();

session.endSession();

print("❌ Loan Payment Failed: " + error);

}

### **Consistency Ensured:**

✅ Before the transaction:

* **CUST002 balance:** ₹8000
* **LOAN\_ACC balance:** ₹0

✅ After a successful transaction:

* **CUST002 balance:** ₹7000
* **LOAN\_ACC balance:** ₹1000
* **Transaction is recorded in the logs**.

❌ If the transaction fails midway (e.g., due to a server crash), the balance remains unchanged, ensuring **no corruption of data**.

Thus, **consistency** ensures that the **database remains in a valid state**.

## ****3️⃣ Isolation (Transactions Don’t Interfere)****

**Definition:** Each transaction runs **independently** without affecting other transactions until committed.

### **Example: Simultaneous Transactions**

Let's assume:

* **Transaction 1:** **Priya (CUST002) repays ₹1000**.
* **Transaction 2:** **Priya (CUST002) withdraws ₹2000** **at the same time**.

If **Transaction 1 starts but is not committed yet**, then **Transaction 2 must wait** to avoid reading an **inconsistent state**.

### **Demonstration of Isolation**

session1 = db.getMongo().startSession();

session1.startTransaction();

session2 = db.getMongo().startSession();

session2.startTransaction();

try {

// Transaction 1: Loan Payment (Priya repays ₹1000)

session1.getDatabase("bankDB").customers.updateOne(

{ customerId: "CUST002" },

{ $inc: { balance: -1000 } }

);

// Transaction 2: Withdrawal (Priya withdraws ₹2000)

session2.getDatabase("bankDB").customers.updateOne(

{ customerId: "CUST002" },

{ $inc: { balance: -2000 } }

);

// Committing Transactions

session1.commitTransaction();

session2.commitTransaction();

session1.endSession();

session2.endSession();

print("✅ Both Transactions Successful!");

} catch (error) {

session1.abortTransaction();

session2.abortTransaction();

session1.endSession();

session2.endSession();

print("❌ Transaction Failed: " + error);

}

### **Isolation Ensured:**

* **If Priya’s balance was ₹8000 initially**:
  + **Transaction 1 (Loan Payment)** **reduces it to ₹7000**.
  + **Transaction 2 (Withdrawal ₹2000)** **should not read the old ₹8000 balance**.
* If **Transaction 1 is not committed yet**, **Transaction 2 waits** or reads the latest balance.

Thus, **isolation prevents dirty reads or partial updates**.

## ****4️⃣ Durability (Changes are Permanent)****

**Definition:** Once a transaction is **committed**, changes **persist** even after a **system failure or crash**.

### **Example: System Crash After Transfer**

✅ Assume **ACC001 transfers ₹500 to ACC002** and the transaction is **committed**.  
✅ Suddenly, the **server crashes**.

When MongoDB **restarts**, the **transaction remains saved**.

### **How MongoDB Ensures Durability**

1️⃣ MongoDB writes transactions to the **WiredTiger journal** before committing.  
2️⃣ If a **system crash** occurs, MongoDB **replays the journal** and applies committed transactions.  
3️⃣ Thus, **data remains safe** even after failures.

### **Example Check After Restart**

db.transactions.find({ transactionId: "TXN002" })

If the **transaction is visible after restart**, **durability is maintained**.

### **🎯 Summary of ACID Properties**

| **Property** | **Definition** | **Example in Banking** |
| --- | --- | --- |
| **Atomicity** | **All or nothing** | Money transfer rolls back if any step fails. |
| **Consistency** | **Database remains valid** | No negative balance, loan payment updates correctly. |
| **Isolation** | **Transactions don't interfere** | Withdrawal & loan payment don’t read partial updates. |
| **Durability** | **Committed data is permanent** | Even after a crash, transactions are saved. |

**Relationships in MongoDB**

In relational databases like MySQL or PostgreSQL, relationships are handled using **foreign keys** and **JOIN operations**. However, MongoDB is a **NoSQL document database** that stores data in **JSON-like BSON documents**, so it handles relationships differently.

In MongoDB, relationships are managed using two approaches:

1. **Embedded Documents (Denormalization)**
2. **Referenced Documents (Normalization)**

## ****1️⃣ Embedded vs. Referenced Documents****

### **🔹 Embedded Documents (Denormalization)**

* Stores related data inside a single document.
* Improves **read performance** because no additional queries are needed.
* Best suited for **one-to-few relationships** where related data does not grow significantly.

✅ **Example: Storing an order with items (One-to-Many)**

{

"\_id": 1,

"customerName": "Alice",

"items": [

{ "productId": 101, "name": "Laptop", "price": 800 },

{ "productId": 102, "name": "Mouse", "price": 20 }

],

"totalAmount": 820

}

📌 **Advantages**: Faster reads, fewer queries.  
📌 **Disadvantages**: Data redundancy, difficult updates when sub-documents are large.

### **🔹 Referenced Documents (Normalization)**

* Stores related data in **separate collections** and links them using **references (ObjectId)**.
* Helps **reduce redundancy** and improves **write performance**.
* Best suited for **one-to-many or many-to-many relationships** where data changes frequently.

✅ **Example: Storing users and their orders separately (One-to-Many)**

#### **User Collection**

{

"\_id": ObjectId("642f9a4b8a3e"),

"name": "Alice",

"email": "alice@example.com",

"orders": [ ObjectId("642f9a4b8a3f") ]

}

#### **Order Collection**

{

"\_id": ObjectId("642f9a4b8a3f"),

"customerId": ObjectId("642f9a4b8a3e"),

"items": [

{ "productId": 101, "name": "Laptop", "price": 800 }

],

"totalAmount": 800

}

📌 **Advantages**: Less duplication, easier updates.  
📌 **Disadvantages**: Requires additional queries ($lookup) to retrieve related data.

## ****2️⃣ Implementing Relationships in MongoDB****

### **🔸 One-to-One Relationship**

* Each document in **Collection A** is related to only one document in **Collection B**.
* Example: A **user** has **one profile**.

#### **Implementation using Referenced Documents**

// Users Collection

{

"\_id": ObjectId("642f9a4b8a3e"),

"name": "John Doe",

"email": "john@example.com",

"profileId": ObjectId("642f9a4b8a3f")

}

// Profiles Collection

{

"\_id": ObjectId("642f9a4b8a3f"),

"bio": "Software Engineer",

"linkedin": "linkedin.com/johndoe"

}

✅ Query to fetch user with profile:

db.users.aggregate([

{

$lookup: {

from: "profiles",

localField: "profileId",

foreignField: "\_id",

as: "profile"

}

}

]);

### **🔸 One-to-Many Relationship**

* One document in **Collection A** is related to **multiple** documents in **Collection B**.
* Example: A **customer** has multiple **orders**.

#### **Implementation using Embedded Documents**

{

"\_id": ObjectId("642f9a4b8a3e"),

"name": "Alice",

"orders": [

{ "orderId": 1, "product": "Laptop", "price": 800 },

{ "orderId": 2, "product": "Mouse", "price": 20 }

]

}

✅ Query to fetch a customer’s orders:

db.customers.find({ name: "Alice" }, { orders: 1 });

#### **Implementation using Referenced Documents**

// Customers Collection

{

"\_id": ObjectId("642f9a4b8a3e"),

"name": "Alice",

"email": "alice@example.com"

}

// Orders Collection

{

"\_id": ObjectId("642f9a4b8a3f"),

"customerId": ObjectId("642f9a4b8a3e"),

"items": [

{ "productId": 101, "name": "Laptop", "price": 800 }

],

"totalAmount": 800

}

✅ Query using $lookup:

db.customers.aggregate([

{

$lookup: {

from: "orders",

localField: "\_id",

foreignField: "customerId",

as: "customer\_orders"

}

}

]);

### **🔸 Many-to-Many Relationship**

* Multiple documents in **Collection A** are related to multiple documents in **Collection B**.
* Example: A **student** can enroll in multiple **courses**, and a **course** can have multiple students.

#### **Implementation using Referenced Documents**

// Students Collection

{

"\_id": ObjectId("642f9a4b8a3e"),

"name": "John Doe",

"enrolledCourses": [ ObjectId("642f9a4b8a3f"), ObjectId("642f9a4b8a40") ]

}

// Courses Collection

{

"\_id": ObjectId("642f9a4b8a3f"),

"courseName": "MongoDB Basics",

"students": [ ObjectId("642f9a4b8a3e"), ObjectId("642f9a4b8a41") ]

}

✅ Query using $lookup:

db.students.aggregate([

{

$lookup: {

from: "courses",

localField: "enrolledCourses",

foreignField: "\_id",

as: "courses"

}

}

]);

## ****🔹 When to Use Embedded vs. Referenced?****

|  |  |  |
| --- | --- | --- |
| Use Case | Embedded Documents | Referenced Documents |
| **Data does not change often** | ✅ Best choice | ❌ Avoid it |
| **Data is frequently updated** | ❌ Not recommended | ✅ Better choice |
| **One-to-Few relationships** | ✅ Efficient | ❌ Unnecessary |
| **One-to-Many / Many-to-Many** | ❌ Causes redundancy | ✅ Scalable |

## ****✅ Summary****

* **Embedded Documents**: Best for small, frequently accessed related data.
* **Referenced Documents**: Best for large, frequently updated related data.
* **One-to-One**: Use embedding for small data; reference for large data.
* **One-to-Many**: Embed for few related documents; reference for many documents.
* **Many-to-Many**: Always use references and $lookup.

## ****Phase 4: Spring Boot REST API Development with MongoDB****

#### **1. Spring Boot & MongoDB Integration**

* Setting up MongoDB with Spring Boot
* Configuring application.properties
* Using MongoRepository for basic database operations

#### **2. REST API Basics**

* Understanding REST principles
* Creating simple GET, POST, PUT, DELETE endpoints

#### **3. CRUD Operations with MongoDB**

* Implementing full CRUD operations using MongoRepository
* Handling ObjectId and custom ID fields

#### **4. Querying MongoDB in Spring Boot**

* Custom Queries using @Query annotation
* Aggregation Queries

#### **5. Exception Handling & Transactions**

* Global Exception Handling (@ControllerAdvice)
* Implementing Transactions in MongoDB

#### **6. Authentication & Authorization (JWT)**

* Securing APIs using JWT authentication
* Role-based access control

#### **7. API Documentation (Swagger/OpenAPI)**

* Integrating Swagger for API documentation

#### **8. Testing & Deployment**

* Writing JUnit & Mockito tests for controllers, services, and repositories
* Containerizing the application with Docker
* Deploying to a cloud platform