MongoDB 2025

Developers HandBook

compiled by Steven

📘 MongoDB Basics – Table of Contents

✅ Chapter 1: Introduction to MongoDB + Installation on Windows

---

📌 What is MongoDB?

MongoDB is a **NoSQL database** designed for **flexibility**, **scalability**, and **performance**. Unlike traditional relational databases that store data in tables, MongoDB stores data in **collections of documents** (similar to JSON objects).

🧠 Key Concepts:

- **NoSQL:** Not Only SQL – works without rigid tables/relations.
- **Document-based:** Stores data in JSON-like format (BSON).
- **Schema-less:** Different documents in the same collection can have different structures.
- **Scalable:** Easy to distribute across multiple servers.
- **Used in modern stacks:** MERN (MongoDB, Express, React, Node)

📦 Real-Life Analogy:

RDBMS (SQL)	MongoDB (NoSQL)
Database	Database
Table	Collection
Row	Document (JSON)
Column	Field

🧰 Use Cases of MongoDB:

- User profile storage
- Logs and analytics
- Real-time data apps (IoT, chats)
- E-commerce product catalogs
- Content management systems

💻 How to Install MongoDB on Windows (2025)

✅ Step-by-Step Guide

1. **Download MongoDB:**
    - Visit: https://www.mongodb.com/try/download/community
    - Select:
        - Version: Latest Stable
        - Platform: Windows
        - Package: `.msi`
2. **Install MongoDB:**
    - Double-click the downloaded `.msi` file.
    - Choose **Complete Setup**.
    - Tick **Install MongoDB as a Service** (recommended).
    - Also install **MongoDB Shell (mongosh)**.
3. **Set the Environment Variable (Optional but helpful):**
    - Add `C:\Program Files\MongoDB\Server\<version>\bin` to your `PATH` environment variable.
4. **Verify Installation:**
    - Open **Command Prompt** and type:

```
mongosh
```

If you see the MongoDB shell prompt (`>`) — you're ready!

📁 MongoDB Folders (Default)

Path	Description
`C:\Program Files\MongoDB\Server\X.X\bin\`	MongoDB executables
`C:\data\db\`	Default data storage (create this manually if needed)

🛠️ If `mongod` doesn't run, make sure `C:\data\db\` exists.

🧪 First MongoDB Commands

```
mongosh
```

Inside the shell:

```
show dbs
use testDB
db.test.insertOne({ name: "MongoDB", type: "NoSQL" })
db.test.find()
```

📝 Assignment

🔧 Tasks:

1. Install MongoDB on Windows using the steps above.
2. Create a folder: `C:\data\db\` (if `mongod` complains).
3. Start `mongosh` and create a new database named `practiceDB`.
4. Insert the following document:

```
db.books.insertOne({ title: "Mongo Mastery", author: "John Doe", pages: 250 })
```

5. Run the following commands:
    - `show dbs`
    - `use practiceDB`
    - `show collections`
    - `db.books.find()`

✅ Chapter 2: Comparing SQL vs MongoDB (Side-by-Side)

Understanding how MongoDB differs from traditional SQL databases is essential before writing real queries.

📊 Relational (SQL) vs Document (MongoDB)

Feature	SQL (MySQL, PostgreSQL)	MongoDB
Data Model	Tables (rows and columns)	Collections (documents - BSON)
Schema	Fixed schema (must define upfront)	Dynamic schema (flexible)
Query Language	SQL (Structured Query Language)	JSON-like query syntax
Relationships	JOINs across multiple tables	Embedded documents or `$lookup`
Transactions	Supported (ACID)	Supported (since v4.0+, ACID)
Scalability	Vertical (add more power to one)	Horizontal (add more servers)
Data Format	Row-based	Document-based (key-value pairs)
Best Suited For	Structured data with relationships	Unstructured/semi-structured

---

📚 SQL vs MongoDB – Terminology Mapping

SQL Term	MongoDB Equivalent
Database	Database
Table	Collection
Row	Document (JSON/BSON)
Column	Field
Primary Key	`_id` field (auto-gen)
Foreign Key	Embedded document / Ref
JOIN	`$lookup` in aggregation

🧠 Example Comparison

🛠️ SQL Table

```
CREATE TABLE students (
  id INT PRIMARY KEY,
  name VARCHAR(50),
  age INT
);
INSERT INTO students VALUES (1, 'Alice', 22);
SELECT * FROM students;
```

🍃 MongoDB Collection

```
db.students.insertOne({ _id: 1, name: "Alice", age: 22 });
db.students.find();
```

🧪 Sample Scenario

SQL Format:

```
SELECT name FROM students WHERE age > 20;
```

MongoDB Equivalent:

```
db.students.find({ age: { $gt: 20 } }, { name: 1, _id: 0 });
```

📝 Assignment

Consider the following **MongoDB collection**:

```
db.users.insertMany([
  { _id: 1, name: "John", age: 28, email: "john@example.com" },
  { _id: 2, name: "Sara", age: 24, email: "sara@example.com" },
  { _id: 3, name: "Mike", age: 31, email: "mike@example.com" }
]);
```

🔍 Tasks (Convert from SQL to MongoDB):

1. **SQL:**

```
SELECT * FROM users;
```

MongoDB:

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

2. **SQL:**

```
SELECT name FROM users WHERE age > 25;
```

MongoDB:

```
db.users.find({ age: { $gt: 25 } }, { name: 1, _id: 0 });
```

3. **SQL:**

```
SELECT COUNT(*) FROM users;
```

MongoDB:

```
db.users.countDocuments();
```

4. **SQL:**

```
DELETE FROM users WHERE age < 30;
```

MongoDB:

```
db.users.deleteMany({ age: { $lt: 30 } });
```

5. **SQL:**

```
UPDATE users SET age = 26 WHERE name = 'Sara';
```

MongoDB:

```
db.users.updateOne({ name: "Sara" }, { $set: { age: 26 } });
```

✅ Chapter 3: Basic Data Types in MongoDB

MongoDB supports a wide range of data types, stored internally in **BSON** (Binary JSON). As a developer, you'll mostly interact using JSON-like syntax, but understanding data types helps in accurate querying and schema design.

🧩 1. Common MongoDB Data Types

Data Type	BSON Type	Example	Description
String	`string`	`"name": "Alice"`	Text value (default for most fields)
Number	`int`, `double`, `long`, `decimal`	`"age": 25`	Integer or floating-point
Boolean	`bool`	`"isActive": true`	true/false values
Date	`date`	`new Date("2023-06-01")`	Date/time object
Array	`array`	`"hobbies": ["reading", "chess"]`	List of values
Object	`object`	`"address": { "city": "NY" }`	Nested key-value pair
Null	`null`	`"middleName": null`	Empty or undefined
ObjectId	`objectId`	`"_id": ObjectId("...")`	Unique 12-byte document ID
Embedded Doc	`object`	`"author": { "name": "Tom" }`	Document inside another

📦 2. Inserting Documents with Various Types

```
db.samples.insertOne({
  name: "LoneCoder",
  age: 30,
  isActive: true,
  skills: ["Node.js", "MongoDB"],
  address: {
    city: "Mumbai",
    pin: 400001
  },
  joinedOn: new Date("2023-01-01"),
  rating: null
});
```

🛠️ 3. Using ObjectId

MongoDB assigns a unique `_id` automatically if not provided:

```
db.students.insertOne({ name: "Ava", age: 23 }); // _id auto-generated
```

You can also manually assign:

```
db.students.insertOne({
  _id: ObjectId("66baaeefb3a4a0f1c8a5b6e9"),
  name: "Sara"
});
```

📁 4. Sample Collection for Practice

```
db.products.insertMany([
  {
    name: "Keyboard",
    price: 799.99,
    inStock: true,
    tags: ["electronics", "input"],
    meta: { weight: "400g", color: "black" },
    launched: new Date("2024-11-01")
  },
  {
    name: "Mouse",
    price: 499.50,
    inStock: false,
    tags: ["electronics", "input"],
    meta: { weight: "150g", color: "white" },
    launched: new Date("2023-09-15")
  }
]);
```

🧠 Notes

- Dates must use `new Date(...)` syntax.
- Arrays can be used in search with `$in`, `$all`, etc.
- Nested objects are powerful for modeling real-world data.

📝 Assignment

Consider the following `students` collection:

```
db.students.insertMany([
  {
    name: "Anika",
    age: 21,
    isEnrolled: true,
    subjects: ["Math", "Physics"],
    contact: { email: "anika@mail.com", phone: "1234567890" },
    joined: new Date("2024-06-01")
  },
  {
    name: "Ravi",
    age: 24,
    isEnrolled: false,
    subjects: ["History", "English"],
    contact: { email: "ravi@mail.com", phone: "9876543210" },
    joined: new Date("2023-09-15")
  }
]);
```

Tasks:

1. Insert the data above into a `students` collection.
2. Write a query to fetch all enrolled students.
3. Write a query to get students who joined after `2024-01-01`.
4. Fetch only `name` and `email` of all students.
5. Find students whose subject includes **Math**.
6. What is the BSON type of `joined` field? (Hint: use `typeof` or schema inspection)

✅ Chapter 4: Various Operators in MongoDB

Operators in MongoDB are used to filter, modify, and project data. They are similar in purpose to SQL operators but follow a different (JSON-like) syntax.

🔍 1. Comparison Operators

MongoDB Operator	Description	Example
`$eq`	Equal to	`{ age: { $eq: 25 } }`
`$ne`	Not equal to	`{ age: { $ne: 30 } }`
`$gt`	Greater than	`{ age: { $gt: 21 } }`
`$lt`	Less than	`{ age: { $lt: 30 } }`
`$gte`	Greater than or equal	`{ age: { $gte: 18 } }`
`$lte`	Less than or equal	`{ age: { $lte: 60 } }`

📜 2. Logical Operators

Operator	Description	Example
`$and`	Match all conditions	`{ $and: [{ age: { $gt: 18 } }, { city: "Delhi" }] }`
`$or`	Match any condition	`{ $or: [{ city: "Delhi" }, { city: "Mumbai" }] }`
`$not`	Inverts condition	`{ age: { $not: { $lt: 18 } } }`
`$nor`	Neither condition true	`{ $nor: [{ city: "Delhi" }, { city: "Mumbai" }] }`

📦 3. Element Operators

Operator	Description	Example
`$exists`	Checks if field exists	`{ email: { $exists: true } }`
`$type`	Matches BSON data type	`{ age: { $type: "int" } }`

---

4. Array Operators

Operator	Description	Example
`$in`	Matches any in a list	`{ city: { $in: ["Delhi", "Mumbai"] } }`
`$all`	All elements must be present	`{ skills: { $all: ["MongoDB", "Node.js"] } }`
`$size`	Array has this length	`{ skills: { $size: 3 } }`
`$elemMatch`	Match based on conditions inside array	`{ results: { $elemMatch: { score: { $gt: 80 } } } }`

---

5. Sample Collection

```
db.employees.insertMany([
  {
    name: "Amit",
    age: 28,
    city: "Delhi",
    skills: ["Node.js", "MongoDB", "Express"],
    isActive: true
  },
  {
    name: "Meera",
    age: 35,
    city: "Mumbai",
    skills: ["React", "Node.js"],
    isActive: false
  },
  {
    name: "Zara",
    age: 24,
    city: "Bangalore",
    skills: ["Python", "Flask"],
    isActive: true
  }
]);
```

🧪 6. Example Queries

```
// Find employees from Delhi
db.employees.find({ city: { $eq: "Delhi" } });

// Employees with age > 30
db.employees.find({ age: { $gt: 30 } });

// Active employees with skill "Node.js"
db.employees.find({
  $and: [
    { isActive: true },
    { skills: { $in: ["Node.js"] } }
  ]
});

// Employees with exactly 2 skills
db.employees.find({ skills: { $size: 2 } });

// Employees with no city field
db.employees.find({ city: { $exists: false } });
```

📝 Assignment

Using the above `employees` collection, write queries for:

1. Find employees aged less than 30.
2. Find employees who are **not** from Delhi.
3. Find employees skilled in either **React** or **MongoDB**.
4. Find employees who have **both** Node.js and MongoDB skills.
5. Find all inactive employees.
6. Find employees with exactly 3 skills.
7. Find employees with age of type integer.
8. Find employees whose city is neither Delhi nor Mumbai.

✅ Chapter 5: `find()` – Select Queries with Basic Operators in MongoDB

In MongoDB, **data is queried using the `find()` method**, which returns documents that match a given filter condition. This chapter covers **selecting data using comparison, logical, and array operators**, along with **field projection**.

📌 1. Basic Syntax of `find()`

```
db.collection.find({ <filter> }, { <projection> });
```

- **Filter** = What to search for (like `WHERE` clause)
- **Projection** = Fields to include or exclude (`1 = include`, `0 = exclude`)

🧪 2. Sample Collection

```
db.students.insertMany([
  {
    name: "Arjun",
    age: 21,
    course: "B.Tech",
    marks: 82,
    subjects: ["Math", "Physics"],
    isActive: true
  },
  {
    name: "Naina",
    age: 22,
    course: "B.Sc",
    marks: 91,
    subjects: ["Biology", "Chemistry"],
    isActive: false
  },
  {
    name: "Kabir",
    age: 20,
    course: "BCA",
    marks: 75,
```

```
    subjects: ["Math", "Computer"],
    isActive: true
  }
]);
```

🧩 3. Simple Queries

```
// Get all students
db.students.find();

// Get students who scored more than 80
db.students.find({ marks: { $gt: 80 } });

// Get students with course B.Tech
db.students.find({ course: "B.Tech" });
```

⚙️ 4. Using Logical Operators

```
// Marks > 80 and course is B.Sc
db.students.find({ $and: [{ marks: { $gt: 80 } }, { course: "B.Sc" }] });

// Marks > 80 or course is BCA
db.students.find({ $or: [{ marks: { $gt: 80 } }, { course: "BCA" }] });

// Not active students
db.students.find({ isActive: { $ne: true } });
```

📂 5. Using Array Queries

```
// Students having 'Math' in subjects
db.students.find({ subjects: "Math" });

// Using $in
db.students.find({ subjects: { $in: ["Biology", "Physics"] } });

// Students with exactly 2 subjects
db.students.find({ subjects: { $size: 2 } });
```

---

6. Field Projection

```
// Get only name and marks
db.students.find({}, { name: 1, marks: 1, _id: 0 });
// Get all data excluding subjects
db.students.find({}, { subjects: 0 });
```

✅ 7. Comparison with SQL

SQL	MongoDB
`SELECT * FROM students`	`db.students.find()`
`SELECT name FROM students`	`db.students.find({}, { name: 1 })`
`WHERE age > 20`	`{ age: { $gt: 20 } }`
`AND`, `OR`	`$and`, `$or`
`IN ('Math', 'Bio')`	`{ subject: { $in: ['Math', 'Bio'] } }`

Assignment

Consider the above `students` collection and write MongoDB queries for:

1. Find all students enrolled in "B.Sc".
2. Get students with marks >= 80.
3. Find all students with **"Math"** as one of their subjects.
4. List only the **names and courses** of all students.
5. Find students who are either active OR from course "BCA".
6. Fetch students who have more than one subject.
7. Exclude `_id` and show all fields **except** `subjects`.

✅ Chapter 6: `Update` & `Delete` Operations in MongoDB

MongoDB provides flexible commands to **modify** and **remove** documents using the `updateOne`, `updateMany`, `deleteOne`, and `deleteMany` methods.

📌 1. Sample Collection

Let's use this for all demo commands:

```
db.employees.insertMany([
  { name: "Amit", age: 30, city: "Delhi", salary: 40000 },
  { name: "Neha", age: 27, city: "Mumbai", salary: 50000 },
  { name: "Rahul", age: 35, city: "Delhi", salary: 60000 },
  { name: "Priya", age: 22, city: "Bangalore", salary: 45000 }
]);
```

🛠️ 2. Update Operations

🔸 `updateOne()` – Update first matching document

```
// Update salary of Amit to 45000
db.employees.updateOne(
  { name: "Amit" },
  { $set: { salary: 45000 } }
);
```

🔸 `updateMany()` – Update all matching documents

```
// Increase salary by 5000 for all employees in Delhi
db.employees.updateMany(
  { city: "Delhi" },
  { $inc: { salary: 5000 } }
);
```

🧰 Common Update Operators

Operator	Purpose	Example
`$set`	Set or overwrite a field	`{ $set: { age: 29 } }`
`$inc`	Increment numeric value	`{ $inc: { salary: 2000 } }`
`$rename`	Rename a field	`{ $rename: { city: "location" } }`
`$unset`	Remove a field	`{ $unset: { city: "" } }`

Operator	Purpose	Example
`$mul`	Multiply a numeric field	`{ $mul: { salary: 1.1 } }`

❌ 3. Delete Operations

🔸 `deleteOne()` – Deletes first matching document

```
// Delete Neha's record
db.employees.deleteOne({ name: "Neha" });
```

🔸 `deleteMany()` – Deletes all matching documents

```
// Delete all employees from Delhi
db.employees.deleteMany({ city: "Delhi" });
```

📜 4. Full Example (Before & After)

🔹 Insert:

```
db.employees.insertOne({ name: "Ravi", age: 28, city: "Chennai", salary: 38000 });
```

🔹 Update:

```
db.employees.updateOne(
  { name: "Ravi" },
  { $set: { salary: 40000 }, $inc: { age: 1 } }
);
```

🔹 Delete:

```
db.employees.deleteOne({ name: "Ravi" });
```

📘 Notes

- Updates do **not** return the updated document by default.
- You can chain updates using multiple operators inside the same `$set`, `$inc`, etc.
- If a field doesn't exist, `$set` will **create** it.

Assignment

Using the original `employees` collection:

1. Update **Rahul's** salary to 65000.
2. Increase salary by 3000 for all employees in **Bangalore**.
3. Rename the field `city` to `location`.
4. Remove the `age` field from all documents.
5. Delete the employee named **Priya**.
6. Delete all employees with salary less than 45000.

✅ Chapter 7: Simple Date and Time Queries in MongoDB

📌 1. Inserting Dates

There are multiple ways to store a date in a document:

```
// Current date using Date object
db.logs.insertOne({
  user: "Arjun",
  action: "login",
  timestamp: new Date()  // returns ISODate
});
// Specific date using ISO format
db.logs.insertOne({
  user: "Neha",
  action: "logout",
  timestamp: ISODate("2024-12-25T00:00:00Z")
});
// Using string (not recommended)
db.logs.insertOne({
  user: "Ravi",
  action: "signup",
  timestamp: "2024-07-01"
});
```

✅ **Recommended:** Use `new Date()` or `ISODate(...)` for full querying support.

📆 2. Sample Collection for Demo

```
db.logs.insertMany([
  { user: "Amit", action: "login",  timestamp: ISODate("2024-06-25T10:00:00Z") },
  { user: "Nina", action: "logout", timestamp: ISODate("2024-06-27T15:30:00Z") },
  { user: "Kabir", action: "login",  timestamp: ISODate("2024-06-29T09:15:00Z") },
  { user: "Sara", action: "login",  timestamp: ISODate("2024-06-30T12:45:00Z") }
]);
```

🔍 3. Querying Based on Date

a. Find logs after a certain date

```
db.logs.find({
  timestamp: { $gt: ISODate("2024-06-28") }
});
```

b. Logs between two dates

```
db.logs.find({
  timestamp: {
    $gte: ISODate("2024-06-27"),
    $lte: ISODate("2024-06-30")
  }
});
```

c. Logs before a specific date

```
db.logs.find({
  timestamp: { $lt: ISODate("2024-06-29") }
});
```

🕐 4. Insert With Dynamic Dates

Insert current timestamp:

```
db.logs.insertOne({
  user: "Zoya",
  action: "signup",
  timestamp: new Date()
});
```

Insert a date 7 days ago:

```
db.logs.insertOne({
  user: "Karan",
  action: "forgot-password",
  timestamp: new Date(new Date().getTime() - 7 * 24 * 60 * 60 * 1000)
});
```

🧠 Tip:

- MongoDB stores time in **UTC**, not local time.
- Use `toISOString()` in client-side code to format a date for MongoDB.

📝 Assignment

Using the `logs` collection above, write queries for:

1. Fetch all logs **after 28th June 2024**.
2. Get logs between **26th and 30th June**.
3. Find users who logged in before **29th June 2024**.
4. Insert a log entry for user `"Ishaan"` with current timestamp.
5. Insert an entry for `"Tanvi"` that occurred exactly **3 days ago**.

✅ Chapter 8: Fetching Data from Multiple Collections in MongoDB (Joins using `$lookup`)

Unlike SQL, MongoDB doesn't have traditional JOINs, but it allows **joining collections** using the powerful **aggregation operator** `$lookup`.

📌 1. Sample Collections

Let's simulate two collections: `orders` and `customers`.

📜 `customers` Collection:

```
db.customers.insertMany([
  { _id: 1, name: "Amit", city: "Delhi" },
  { _id: 2, name: "Neha", city: "Mumbai" },
  { _id: 3, name: "Ravi", city: "Bangalore" }
]);
```

📜 `orders` Collection:

```
db.orders.insertMany([
  { _id: 101, customer_id: 1, amount: 5000 },
  { _id: 102, customer_id: 2, amount: 7000 },
  { _id: 103, customer_id: 1, amount: 2000 },
  { _id: 104, customer_id: 3, amount: 3000 }
]);
```

🔄 2. Using `$lookup` to Join

Join `orders` with `customers`

```
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",
      localField: "customer_id",   // in orders
      foreignField: "_id",         // in customers
      as: "customer_info"
    }
  }
]);
```

This will return each order with matching customer details under `customer_info` array.

📜 3. Result Example

```
{
  _id: 101,
  customer_id: 1,
  amount: 5000,
  customer_info: [
    { _id: 1, name: "Amit", city: "Delhi" }
  ]
}
```

🎯 4. Flattening the Output (Optional)

To remove the array (`customer_info`) and bring details up:

```
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",
      localField: "customer_id",
      foreignField: "_id",
      as: "customer_info"
    }
  },
  { $unwind: "$customer_info" }
]);
```

This returns one object per order with directly embedded customer details.

🧠 5. Key Points

Concept	Explanation
`$lookup`	Performs a left outer join
`from`	The name of the collection to join with
`localField`	The field from the input documents
`foreignField`	The field from the joined documents
`as`	Output array field name
`$unwind`	Converts the array to a flat object

📝 Assignment

Using the `orders` and `customers` collection:

1. Write a query to fetch **all orders along with customer names**.
2. Fetch only those orders with amount > 3000 and show their customer info.
3. Join and display customer `city` alongside each order.
4. Modify the `$lookup` query to **flatten the** `customer_info` using `$unwind`.
5. Find all customers who made **at least 1 order** (i.e., those appearing in `orders`).

✅ Chapter 9: Aggregation Queries in MongoDB

The **Aggregation Framework** in MongoDB is used to process data records and return computed results — similar to `GROUP BY`, `SUM()`, `COUNT()` in SQL.

You use the `.aggregate()` method with a pipeline of **stages** such as `$match`, `$group`, `$sort`, etc.

📜 Sample Collection

We'll use a `sales` collection:

```
db.sales.insertMany([
  { item: "Pen", category: "Stationery", quantity: 10, price: 5 },
  { item: "Notebook", category: "Stationery", quantity: 5, price: 20 },
  { item: "Pencil", category: "Stationery", quantity: 15, price: 3 },
  { item: "Mouse", category: "Electronics", quantity: 3, price: 500 },
  { item: "Keyboard", category: "Electronics", quantity: 2, price: 800 }
]);
```

🔍 1. `$match` – Filter Documents

```
db.sales.aggregate([
  { $match: { category: "Electronics" } }
]);
```

📊 2. `$group` – Group and Aggregate Data

Total quantity sold per category

```
db.sales.aggregate([
  {
    $group: {
      _id: "$category",
      totalQty: { $sum: "$quantity" }
    }
  }
]);
```

🩹 3. `$sum` with Computed Fields

Total revenue per category (price × quantity)

```
db.sales.aggregate([
  {
    $group: {
      _id: "$category",
      totalRevenue: { $sum: { $multiply: ["$price", "$quantity"] } }
    }
  }
]);
```

📊 4. `$sort` Results

Sort by total revenue (descending):

```
db.sales.aggregate([
  {
    $group: {
      _id: "$category",
      totalRevenue: { $sum: { $multiply: ["$price", "$quantity"] } }
    }
  },
  { $sort: { totalRevenue: -1 } }
]);
```

🧠 5. Common Aggregation Operators

Operator	Purpose
`$sum`	Adds values
`$avg`	Computes average
`$min`, `$max`	Min or Max value
`$count`	Counts number of docs
`$match`	Filters input
`$group`	Group by a field
`$sort`	Sorts results
`$project`	Reshapes output
`$multiply`, `$divide`	Math operations

Assignment

Using the `sales` collection above:

1. Find total quantity sold for each category.
2. Calculate total revenue for each product.
3. Find the average price of items per category.
4. Filter all products with quantity ≥ 5.
5. Sort all products by total revenue descending.
6. Count the number of items in each category.

✅ Chapter 10: Final Assignment — MongoDB Practice (All Topics Combined)

This assignment includes:

- A sample collection (2 collections)
- Realistic fields (e.g., name, date, sales, location)
- 10+ queries covering CRUD, operators, aggregation, `$lookup`, dates, etc.
- Answers/solutions included

📦 1. Sample Data

📜 `users` Collection

```
db.users.insertMany([
  { _id: 1, name: "Amit", age: 28, city: "Delhi", joined: ISODate("2022-01-15") },
  { _id: 2, name: "Neha", age: 34, city: "Mumbai", joined: ISODate("2023-03-10") },
  { _id: 3, name: "Ravi", age: 22, city: "Bangalore", joined: ISODate("2021-11-05") },
  { _id: 4, name: "Priya", age: 29, city: "Delhi", joined: ISODate("2022-07-18") }
]);
```

📜 `orders` Collection

```
db.orders.insertMany([
  { _id: 101, user_id: 1, product: "Phone", amount: 15000, date: ISODate("2023-01-10") },
  { _id: 102, user_id: 2, product: "Laptop", amount: 50000, date: ISODate("2023-06-11") },
  { _id: 103, user_id: 1, product: "Charger", amount: 1500, date: ISODate("2023-03-25") },
  { _id: 104, user_id: 4, product: "Tablet", amount: 18000, date: ISODate("2023-04-18") },
  { _id: 105, user_id: 3, product: "Mouse", amount: 1200, date: ISODate("2022-12-02") }
]);
```

🧠 2. Assignment Questions

Write MongoDB queries for the following:

1. Find all users from Delhi.
2. Find users who joined after Jan 1, 2022.
3. Get all orders above ₹10,000.
4. Show orders along with user details (`$lookup`).
5. List each user's total order amount (`$group`).
6. Find users who have made at least one order.
7. Get average order amount per product.
8. Count number of users per city.
9. Find the most expensive order.
10. List all orders made in the year 2023.
11. Update user Ravi’s city to "Hyderabad".
12. Delete all orders under ₹2000.

✅ 3. Answers / Solutions

1. **Users from Delhi**

```
db.users.find({ city: "Delhi" });
```

2. **Users who joined after Jan 1, 2022**

```
db.users.find({ joined: { $gt: ISODate("2022-01-01") } });
```

3. **Orders above ₹10,000**

```
db.orders.find({ amount: { $gt: 10000 } });
```

4. **Join orders with user info**

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

5. **Total order amount per user**

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

6. **Users with at least one order**

```
db.orders.distinct("user_id");
```

Then use:

```
db.users.find({ _id: { $in: [1, 2, 3, 4] } });
```

7. **Average order amount per product**

```
db.orders.aggregate([
  {
    $group: {
```

```
      _id: "$product",
      avgAmount: { $avg: "$amount" }
    }
  }
]);
```

8. **Count users per city**

```
db.users.aggregate([
  {
    $group: {
      _id: "$city",
      count: { $sum: 1 }
    }
  }
]);
```

9. **Most expensive order**

```
db.orders.find().sort({ amount: -1 }).limit(1);
```

10. **Orders made in 2023**

```
db.orders.find({
  date: {
    $gte: ISODate("2023-01-01"),
    $lt: ISODate("2024-01-01")
  }
});
```

11. **Update Ravi's city**

```
db.users.updateOne({ name: "Ravi" }, { $set: { city: "Hyderabad" } });
```

12. **Delete orders under ₹2000**

```
db.orders.deleteMany({ amount: { $lt: 2000 } });
```

✅ **Syllabus Complete!**

You've now covered:

- Installation
- SQL vs MongoDB
- Data types
- Operators
- CRUD
- Querying
- Dates
- Joins (`$lookup`)
- Aggregations (`$group`, `$sum`, `$match`, `$sort`, etc.)
