Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

PERFORMING ADVANCED QUERIES & AGGREGATION FRAMEWORK (WEEKS 5-6)

UNDERSTANDING QUERY OPERATORS IN MONGODB ($EQ, $GT, $LT, $IN, ETC.)

CHAPTER 1: INTRODUCTION TO MONGODB QUERY OPERATORS

1.1 What Are Query Operators in MongoDB?

MongoDB **query operators** allow us to filter and retrieve documents based on **specific conditions**. These operators are used with **find()**, **findOne()**, and aggregation queries to fetch data efficiently.

MongoDB query operators are categorized into:

✓ **Comparison Operators** – Compare values ($eq, $gt, $lt, $in).

✓ **Logical Operators** – Combine multiple conditions ($and, $or, $not).

✓ **Element Operators** – Check field existence ($exists, $type).

✓ **Evaluation Operators** – Perform regular expressions ($regex), text search ($text).

These operators help optimize **data retrieval**, ensuring queries run **faster and more efficiently**.

CHAPTER 2: COMPARISON OPERATORS IN MONGODB

2.1 $eq (Equal To Operator)

Finds documents where a field **equals** a specified value.

Example: Finding a User by Email

```
db.users.find({ email: { $eq: "alice@example.com" } });
```

✓ Retrieves a user where email == "alice@example.com".

2.2 $gt (Greater Than Operator)

Finds documents where a field is **greater than** a specified value.

Example: Finding Users Older Than 30

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

✓ Returns users where age > 30.

2.3 $lt (Less Than Operator)

Finds documents where a field is **less than** a specified value.

Example: Finding Products Cheaper Than $50

```
db.products.find({ price: { $lt: 50 } });
```

✓ Retrieves products where price < 50.

2.4 $gte (Greater Than or Equal To Operator)

Finds documents where a field is **greater than or equal to** a specified value.

Example: Finding Employees with Salary >= 50000

db.employees.find({ salary: { $gte: 50000 } });

✓ Returns employees with salary >= 50000.

2.5 $lte (Less Than or Equal To Operator)

Finds documents where a field is **less than or equal to** a specified value.

Example: Finding Books with Rating ≤ 4.5

db.books.find({ rating: { $lte: 4.5 } });

✓ Retrieves books where rating <= 4.5.

2.6 $ne (Not Equal Operator)

Finds documents where a field **is not equal to** a specified value.

Example: Finding All Users Except Admins

db.users.find({ role: { $ne: "admin" } });

✓ Returns all users except those with role == "admin".

2.7 $in (Matches Any in an Array)

Finds documents where a field matches **any value from a list**.

Example: Finding Users in Specific Cities

db.users.find({ city: { $in: ["New York", "Los Angeles", "Chicago"] } });

✓ Retrieves users from New York, Los Angeles, or Chicago.

2.8 $nin (Matches None in an Array)

Finds documents where a field **does not match any values** in a list.

Example: Excluding Users from Specific Cities

```
db.users.find({ city: { $nin: ["New York", "Los Angeles"] } });
```

✓ Retrieves users **not** in New York or Los Angeles.

CHAPTER 3: LOGICAL OPERATORS IN MONGODB

3.1 $and (Combining Multiple Conditions)

Finds documents where **all conditions are true**.

Example: Finding Employees in New York with Salary > 60000

```
db.employees.find({
  $and: [
    { city: "New York" },
    { salary: { $gt: 60000 } }
  ]
});
```

✓ Retrieves employees in **New York** with salary > 60000.

3.2 $or (At Least One Condition Must Be True)

Finds documents where **at least one condition is true**.

Example: Finding Products Priced Below $50 or Rated Above 4.5

```
db.products.find({
```

```
  $or: [
    { price: { $lt: 50 } },
    { rating: { $gt: 4.5 } }
  ]
});
```

✓ Retrieves products that are either **cheap** or **highly rated**.

3.3 $not (Negates a Condition)

Finds documents where a condition **is not true**.

Example: Excluding Users Younger Than 30

```
db.users.find({ age: { $not: { $lt: 30 } } });
```

✓ Retrieves users **30 years or older**.

CHAPTER 4: ELEMENT OPERATORS IN MONGODB

4.1 $exists (Checking Field Presence)

Finds documents where a field **exists or does not exist**.

Example: Finding Users Who Have a Phone Number

```
db.users.find({ phone: { $exists: true } });
```

✓ Retrieves users **who have a phone number**.

4.2 $type (Checking Field Type)

Finds documents where a field is of a **specific data type**.

Example: Finding Documents Where age is a Number

```
db.users.find({ age: { $type: "number" } });
```

✓ Retrieves users where age is stored as a **number**.

CHAPTER 5: EVALUATION OPERATORS IN MONGODB

5.1 $regex (Pattern Matching for Text Search)

Finds documents where a field **matches a regex pattern**.

Example: Finding Users Whose Name Starts with "A"

```
db.users.find({ name: { $regex: "^A" } });
```

✓ Retrieves users **whose name starts with 'A'**.

5.2 $text (Full-Text Search)

Finds documents **using text indexes** for efficient searching.

Example: Searching for Articles Containing "MongoDB"

```
db.articles.createIndex({ content: "text" });
db.articles.find({ $text: { $search: "MongoDB" } });
```

✓ Finds articles where "MongoDB" appears in **content**.

Case Study: How Uber Uses MongoDB Query Optimization

Background

Uber manages **millions of ride requests** per day, requiring **fast, efficient** queries to match drivers with riders.

Challenges

- **Quickly retrieving available drivers near a user.**
- **Filtering results based on location, rating, and car type.**
- **Ensuring low latency for ride-matching.**

Solution: Implementing MongoDB Query Operators

✓ Used **$near and $gt operators** to find available drivers nearby.
✓ Applied **compound indexing** to optimize location-based queries.
✓ Leveraged **$in and $or operators** to match user preferences.

By **optimizing queries**, Uber **reduced query time by 60%,** improving ride-matching efficiency.

Exercise

1. Write a MongoDB query to **find all users who are older than 25 and live in Los Angeles**.
2. Use $or to find **products that are either cheaper than $50 or have a rating above 4.5**.
3. Retrieve documents where the email field **does not exist**.

Conclusion

In this section, we explored:
✓ **Comparison operators ($eq, $gt, $lt, $in, etc.)**.
✓ **Logical operators ($and, $or, $not) for combining conditions**.
✓ **Element operators ($exists, $type) for field validation**.
✓ **Text search using $regex and $text**.
✓ **How Uber optimizes MongoDB queries for ride-matching**.

FILTERING & SORTING DATA IN MONGODB

CHAPTER 1: INTRODUCTION TO FILTERING AND SORTING IN MONGODB

1.1 Understanding Filtering and Sorting

One of the key advantages of **MongoDB** is its ability to retrieve and organize data efficiently using **filtering and sorting**.

✓ **Filtering** – Retrieves only the documents that match specific conditions.
✓ **Sorting** – Arranges retrieved documents based on one or more fields.

These operations are essential for **query optimization, performance improvement, and user experience** in applications.

1.2 Why Filtering and Sorting Matter?

✓ **Improves Performance** – Fetches only relevant data instead of scanning the entire collection.
✓ **Enhances User Experience** – Displays **sorted search results** in e-commerce, blogs, and news feeds.
✓ **Optimizes Data Processing** – Reduces server load by **retrieving only necessary documents**.

MongoDB provides various **operators** ($eq, $gt, $lt, $in, $regex, etc.) and **methods** (find(), sort(), limit()) to perform these operations efficiently.

CHAPTER 2: FILTERING DATA IN MONGODB

2.1 Using find() to Retrieve Documents

The **find() method** is used to filter and retrieve documents that match a **specific query condition**.

Example: Finding All Users in a Collection

db.users.find()

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

2.2 Using Query Operators for Filtering

Filtering by a Single Field ($eq - Equal to)

To find a user with a specific email:

db.users.find({ email: "john@example.com" })

✓ Retrieves only the document **where email matches "john@example.com"**.

Filtering by Numeric Values ($gt, $lt, $gte, $lte)

To find users older than 25:

db.users.find({ age: { $gt: 25 } })

✓ Returns **all users with age > 25**.

To find users aged **between 18 and 30**:

db.users.find({ age: { $gte: 18, $lte: 30 } })

✓ Retrieves **all users between 18 and 30 years old**.

2.3 Filtering by Multiple Conditions ($and, $or)

Using $and (Match Multiple Conditions)

Find users who are **older than 25** and **are admins**:

```
db.users.find({
  $and: [
```

```
    { age: { $gt: 25 } },
    { isAdmin: true }
  ]
})
```

✓ Returns **only users who meet both conditions**.

Using $or (Match One of Multiple Conditions)

Find users who are **either older than 25 or admins**:

```
db.users.find({
  $or: [
    { age: { $gt: 25 } },
    { isAdmin: true }
  ]
})
```

✓ Returns users who meet **at least one condition**.

2.4 Filtering Arrays ($in, $nin, $all)

Using $in to Match Any Value in an Array

Find users with **specific roles** ("admin" or "editor"):

```
db.users.find({ role: { $in: ["admin", "editor"] } })
```

✓ Returns users with **either "admin" or "editor" roles**.

Using $all to Match All Values in an Array

Find products with **both "electronics" and "gadget" tags**:

```
db.products.find({ tags: { $all: ["electronics", "gadget"] } })
```

✓ Retrieves only products that contain **both tags**.

2.5 Using Regular Expressions for Text Filtering ($regex)

Find users whose **names start with "A"**:

```
db.users.find({ name: { $regex: "^A", $options: "i" } })
```

✓ The ^A pattern ensures names **start with "A"** (case-insensitive).

CHAPTER 3: SORTING DATA IN MONGODB

3.1 Sorting Using the sort() Method

The **sort() method** arranges the retrieved documents based on specified field(s).

Sorting in Ascending Order (1)

Sort users **by age in ascending order**:

```
db.users.find().sort({ age: 1 })
```

✓ Returns users **from youngest to oldest**.

Sorting in Descending Order (-1)

Sort users **by age in descending order**:

```
db.users.find().sort({ age: -1 })
```

✓ Returns users **from oldest to youngest**.

3.2 Sorting by Multiple Fields

Sort users **by isAdmin first, then by age**:

```
db.users.find().sort({ isAdmin: -1, age: 1 })
```

✓ Admin users are displayed **first,** sorted by age **in ascending order**.

CHAPTER 4: COMBINING FILTERING AND SORTING

4.1 Finding and Sorting Together

To find **all users older than 20 and sort them by age (ascending)**:

```
db.users.find({ age: { $gt: 20 } }).sort({ age: 1 })
```

✓ Filters users **older than 20,** then **sorts them by age**.

4.2 Using limit() to Restrict Results

To get the **top 5 oldest users**:

```
db.users.find().sort({ age: -1 }).limit(5)
```

✓ Retrieves only **5 users,** sorted by age **in descending order**.

4.3 Using skip() for Pagination

To **skip the first 5 results and return the next 5**:

```
db.users.find().sort({ age: -1 }).skip(5).limit(5)
```

✓ Useful for **pagination,** e.g., displaying **page 2** of search results.

Case Study: How a News Website Used Filtering & Sorting for User Experience

Background

A news website needed to **display articles efficiently** with:

✓ **Category-based filtering** (e.g., Technology, Politics, Sports).
✓ **Sorting by popularity and date**.
✓ **Pagination for faster browsing**.

Challenges

- Querying **millions of articles** slowed down performance.
- Users wanted **customized filtering** (e.g., trending news, latest news).
- Backend processing of large datasets **increased server load**.

Solution: Implementing Filtering & Sorting in MongoDB

The team optimized queries by:

✓ Using **find({ category: "Technology" })** to fetch relevant news.
✓ Sorting articles **by views and timestamp** using sort().
✓ Implementing **pagination with skip() and limit()**.

Results

- **Faster response times**, reducing page load time by **50%**.
- **Increased engagement**, as users found relevant articles quickly.
- **Optimized backend performance**, reducing server workload.

By leveraging **MongoDB's filtering and sorting**, the website significantly improved **user experience and search efficiency**.

Exercise

1. Write a MongoDB query to **find all products** in the "electronics" category.
2. Modify the query to **sort products by price in descending order**.
3. Retrieve **only the top 10 most expensive products** using limit().

4. Implement pagination by **skipping the first 10 results and fetching the next 10**.

Conclusion

In this section, we explored:

✓ **How to filter documents using find() and query operators ($gt, $or, $regex)**.

✓ **How to sort data using sort() for ascending and descending order**.

✓ **How to implement pagination using skip() and limit()**.

USING PROJECTION TO OPTIMIZE QUERY PERFORMANCE IN MONGODB

CHAPTER 1: INTRODUCTION TO QUERY PROJECTION

1.1 Understanding Projection in MongoDB

Projection in MongoDB allows you to **selectively retrieve specific fields** from a document instead of fetching the entire document. This helps:

✓ **Reduce query execution time** by fetching only the necessary data.
✓ **Minimize network bandwidth usage** when transferring data.
✓ **Improve database performance** by optimizing resource consumption.

MongoDB projections are used with the .find() and .findOne() queries to limit the fields returned in the query result.

Example: Without Projection (Fetching All Fields)

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

This query retrieves **all fields** for Alice, which may be unnecessary if only the email is required.

Example: With Projection (Fetching Specific Fields)

```
db.users.find({ name: "Alice" }, { email: 1, _id: 0 })
```

✓ **Only the email field is returned,** excluding _id.
✓ **Reduces the amount of data retrieved,** improving efficiency.

CHAPTER 2: IMPLEMENTING PROJECTION IN MONGODB QUERIES

2.1 Syntax of Projection in MongoDB

The **projection object** follows this structure:

db.collection.find(query, { field1: 1, field2: 1, _id: 0 })

- **1** ⟶ Includes the field.
- **0** ⟶ Excludes the field.
- **By default, _id is included** unless explicitly set to 0.

2.2 Projection with Mongoose in Node.js

Mongoose also supports projections using the .select() method.

Example: Fetching Specific Fields in Mongoose

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


User.find({}, 'name email -_id') // Includes 'name' and 'email', excludes '_id'
  .then(users => console.log(users))
  .catch(err => console.error(err));
```

✓ Fetches only the name and email fields.
✓ _id is excluded using - notation (-_id).

CHAPTER 3: USE CASES FOR PROJECTION IN REAL APPLICATIONS

3.1 Improving API Performance

When building **REST APIs**, it is inefficient to send **unnecessary fields** to the client.

Example: Optimizing API Response in Express.js

```
app.get('/users', async (req, res) => {
  try {
    const users = await User.find({}, 'name email'); // Only return name and email
    res.json(users);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

✓ **Improves API response time** by limiting returned data.

3.2 Protecting Sensitive Data

Sensitive fields like **passwords** and **tokens** should not be exposed in API responses.

Example: Excluding Password Field

```
User.find({}, '-password')
  .then(users => console.log(users));
```

✓ Prevents **password leaks**, improving **security**.

3.3 Using Projection with Nested Documents

Example: Fetching Specific Fields from Embedded Documents

```
db.orders.find({}, { "customer.name": 1, "customer.email": 1, _id: 0 })
```

✓ Retrieves **only customer names and emails** from the orders collection.

CHAPTER 4: OPTIMIZING LARGE-SCALE QUERIES WITH PROJECTION

4.1 Using Projection with Indexing

Combining **projections with indexes** significantly improves **query performance**.

Example: Indexing and Projection Together

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

db.users.find({}, { email: 1, _id: 0 });
```

✓ **Indexing improves query speed**, and projection **reduces data size**.

4.2 Limiting Data for Paginated Queries

When paginating results, projection ensures that only required fields are retrieved.

Example: Fetching Limited Fields with Pagination

```
User.find({}, 'name email')
  .skip(10) // Skip first 10 results
  .limit(5) // Retrieve next 5 results
  .then(users => console.log(users));
```

✓ Reduces **database load** when paginating large datasets.

Case Study: How a Social Media App Improved Query Performance Using Projection

Background

A social media platform faced **slow API responses** due to **large user documents**.

Challenges

✓ Fetching **entire user profiles** for every API request.
✓ High **network bandwidth usage** causing slow responses.

Solution: Implementing Projection

✓ Modified queries to **return only required fields** (name, profilePicture).
✓ Used **indexing and projection together** to speed up lookups.

Results

- **60% faster API responses**.
- **Reduced network load,** improving scalability.
- **Improved user experience** with faster data retrieval.

This case study highlights how **projection significantly improves MongoDB query performance**.

Exercise

1. Write a **MongoDB query** that retrieves only title and price fields from a products collection.
2. Modify an API to **exclude the password field** when returning user data.

3. Optimize a query by **combining projection with pagination**.

Conclusion

✓ **Projection improves query performance** by fetching only needed fields.

✓ **Helps protect sensitive data** (e.g., passwords) in APIs.

✓ **Enhances scalability** by reducing database load.

INTRODUCTION TO AGGREGATION PIPELINES

CHAPTER 1: UNDERSTANDING AGGREGATION IN MONGODB

1.1 What is Aggregation?

Aggregation in MongoDB is a **data processing method** that allows complex transformations and computations on large datasets. It is used to **summarize, analyze, and manipulate data** efficiently.

Aggregation is similar to SQL's **GROUP BY** and **JOIN** operations, but it offers **more flexibility** for working with JSON-like documents. It enables operations such as:

- Summarizing data (e.g., **total sales per month**)
- Filtering and transforming documents
- Grouping and sorting records
- Performing calculations (e.g., **average order value**)

Aggregation pipelines allow for **chaining multiple stages** to process data in a **structured manner**.

1.2 Why Use Aggregation Pipelines?

✓ **Efficient Data Processing** – Queries large datasets quickly.
✓ **Flexible & Scalable** – Works well with complex NoSQL documents.
✓ **Reduces Application Load** – Moves data transformation logic to the database.

Example Use Cases of Aggregation

Use Case	Example
Sales Report	Calculate total sales per month
Customer Insights	Find top customers based on purchases
Website Analytics	Count daily active users
Product Performance	Find the most viewed or purchased products
Fraud Detection	Detect unusual transaction patterns

CHAPTER 2: BASICS OF MONGODB AGGREGATION PIPELINES

2.1 What is an Aggregation Pipeline?

The **aggregation pipeline** is a framework in MongoDB that processes documents in **multiple stages,** transforming them into meaningful results.

Aggregation Pipeline Structure

```
db.collection.aggregate([
  { Stage 1: Operation },
  { Stage 2: Operation },
  { Stage 3: Operation }
]);
```

Each stage processes the input documents and passes the transformed output to the next stage.

2.2 Common Stages in Aggregation Pipelines

Stage Operator	Description
$match	Filters documents (like WHERE in SQL)
$group	Groups documents by a field (like GROUP BY)
$sort	Sorts documents (like ORDER BY)
$project	Reshapes documents (selects specific fields)
$limit	Limits the number of output documents
$skip	Skips a specified number of documents
$count	Counts the number of documents in a group
$lookup	Joins documents from another collection (like JOIN in SQL)

CHAPTER 3: IMPLEMENTING AGGREGATION PIPELINES

3.1 Using $match for Filtering Data

The $match stage filters documents based on conditions, similar to an SQL WHERE clause.

Example: Find all users from "New York"

```
db.users.aggregate([
  { $match: { city: "New York" } }
]);
```

✓ **Filters documents before passing them to the next stage**.
✓ Improves performance by reducing the number of documents processed.

3.2 Using $group for Summarizing Data

The $group stage aggregates documents by a field, similar to SQL's GROUP BY.

Example: Total sales per month

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

✓ Groups sales by **month** and calculates **total sales**.

3.3 Using $sort for Sorting Data

The $sort stage arranges documents in ascending (1) or descending (-1) order.

Example: Sort users by age in descending order

```
db.users.aggregate([
  { $sort: { age: -1 } }
]);
```

✓ Retrieves users **from oldest to youngest**.

3.4 Using $project to Reshape Documents

The $project stage allows us to **include, exclude, or transform fields** in the output.

Example: Retrieve only name and email, excluding _id

```
db.users.aggregate([
  {
    $project: {
      _id: 0,
      name: 1,
      email: 1
    }
  }
]);
```

✓ Reshapes documents by selecting specific fields.

CHAPTER 4: ADVANCED AGGREGATION TECHNIQUES

4.1 Combining Multiple Stages in a Pipeline

Aggregation pipelines allow **combining multiple stages** for advanced queries.

Example: Find the top 3 highest-spending customers

```
db.orders.aggregate([
  { $group: { _id: "$customer_id", total_spent: { $sum: "$amount" } } },
```

```
  { $sort: { total_spent: -1 } },
  { $limit: 3 }
]);
```

✓ Groups orders by customer_id, **sums purchases**, sorts them, and returns the **top 3 customers**.

4.2 Using $lookup for Joining Collections

MongoDB does not support SQL-style JOINs, but $lookup allows joining data from different collections.

Example: Join users with their orders

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

✓ Retrieves **user details along with their orders** in a single query.

Case Study: How Netflix Uses Aggregation for Personalized Recommendations

Background

Netflix, a global streaming service, uses MongoDB to manage its vast collection of **movies, TV shows, and user watch history**.

Challenges

- Analyzing **millions of user preferences** in real time.
- Generating **personalized recommendations** efficiently.
- Managing **large datasets of viewing history and ratings**.

Solution: Using Aggregation Pipelines

✓ **Grouped user watch history** to find favorite genres.
✓ **Used $lookup to fetch movie details** and ratings.
✓ **Implemented $sort and $limit** to recommend top-rated shows.

Results

- **30% faster recommendation generation**.
- **Higher user engagement** due to personalized content.
- **Reduced query load,** improving server efficiency.

This case study highlights how **aggregation pipelines power real-time data analytics**.

Exercise

1. What is the purpose of the $group stage in MongoDB aggregation?

2. Write an aggregation query to find the **total number of products in each category**.
3. How does $lookup help in combining data from different collections?

Conclusion

In this section, we explored:

✓ **What aggregation pipelines are and why they are useful.**

✓ **Basic and advanced aggregation stages, including $match, $group, $sort, and $lookup**.

✓ **How to design efficient aggregation pipelines for real-world applications**.

STAGES IN AGGREGATION: $MATCH, $GROUP, $PROJECT, $LOOKUP IN MONGODB

CHAPTER 1: INTRODUCTION TO AGGREGATION IN MONGODB

1.1 What is Aggregation in MongoDB?

Aggregation in **MongoDB** is a powerful way to process and transform data. Unlike **basic queries**, aggregation **performs complex operations like filtering, grouping, and joining collections** before returning results.

The **Aggregation Framework** allows **multi-stage pipelines**, where each stage processes the data step by step.

✓ **Filters data using $match** (similar to WHERE in SQL).
✓ **Groups data using $group** (like GROUP BY in SQL).
✓ **Projects specific fields using $project** (like SELECT in SQL).
✓ **Joins collections using $lookup** (similar to JOIN in SQL).

1.2 Why Use Aggregation?

Aggregation is useful for:
✓ **Generating reports** (e.g., total sales per month).
✓ **Performing analytics** (e.g., average ratings of products).
✓ **Filtering and transforming large datasets** efficiently.
✓ **Joining collections** without needing relational databases.

CHAPTER 2: USING $MATCH TO FILTER DATA

2.1 What is $match?

The $match stage filters documents **before further processing,** improving performance. It works like **SQL's WHERE clause**.

2.2 Example: Filtering Users Above Age 30

```
db.users.aggregate([
  { $match: { age: { $gt: 30 } } }
]);
```

✓ Returns **users older than 30**.

2.3 Example: Filtering Orders from a Specific City

```
db.orders.aggregate([
  { $match: { city: "New York" } }
]);
```

✓ Retrieves **only orders from New York**.

CHAPTER 3: USING $GROUP TO AGGREGATE DATA

3.1 What is $group?

The $group stage **groups documents** by a specified field and applies **aggregation functions** like:

- **$sum** – Calculates the total sum.
- **$avg** – Finds the average value.
- **$min/$max** – Finds minimum or maximum values.

3.2 Example: Calculating Total Sales per City

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

✓ Groups orders **by city** and calculates **total sales**.

3.3 Example: Finding the Average Age of Users

```
db.users.aggregate([
 { $group: { _id: null, avgAge: { $avg: "$age" } } }
]);
```

✓ Returns the **average age of all users**.

3.4 Example: Counting the Number of Users Per City

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

✓ Counts **how many users exist per city**.

CHAPTER 4: USING $PROJECT TO RESTRUCTURE OUTPUT

4.1 What is $project?

The $project stage **selects specific fields** and can **modify, rename, or exclude** fields. It is similar to SQL's **SELECT statement**.

4.2 Example: Selecting Only Name and Email from Users

```
db.users.aggregate([
  { $project: { _id: 0, name: 1, email: 1 } }
]);
```

✓ Only **name and email** fields are returned.

4.3 Example: Adding a New Field (fullName)

```
db.users.aggregate([
  { $project: { _id: 0, fullName: { $concat: ["$firstName", " ", "$lastName"] } } }
]);
```

✓ Creates a **new field fullName** by merging firstName and lastName.

4.4 Example: Renaming Fields in the Output

```
db.products.aggregate([
  { $project: { _id: 0, productName: "$name", productPrice: "$price" } }
]);
```

✓ Renames name → productName and price → productPrice.

CHAPTER 5: USING $LOOKUP TO JOIN COLLECTIONS

5.1 What is $lookup?

The $lookup stage **performs joins between two collections,** similar to SQL's JOIN. It allows fetching **related documents from another collection**.

5.2 Example: Joining users with orders Collection

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

✓ Fetches **orders related to each user** and stores them in userOrders.

5.3 Example: Joining products with categories

```
db.products.aggregate([
 {
  $lookup: {
```

```
    from: "categories",
    localField: "categoryId",
    foreignField: "_id",
    as: "categoryDetails"
  }
 }
]);
```

✓ Returns **product details with corresponding category info**.

Case Study: How Airbnb Uses Aggregation to Analyze Listings

Background

Airbnb processes **millions of rental listings** and needs **fast analytics** for pricing, availability, and customer trends.

Challenges

- **Grouping rentals by city to analyze average prices.**
- **Filtering data based on available listings.**
- **Joining listings with customer reviews to track satisfaction.**

Solution: Using Aggregation in MongoDB

✓ **Used $match to filter active listings**.
✓ **Implemented $group to calculate average price per city**.
✓ **Joined listings with customer reviews using $lookup**.

By optimizing aggregation queries, Airbnb improved **search speed by 40%** and enhanced **pricing insights** for hosts.

Exercise

1. Write an aggregation query using $match to **find users from "Los Angeles"**.
2. Use $group to **calculate total revenue per month** in an orders collection.
3. Apply $project to **return only name and price** from a products collection.
4. Use $lookup to **join employees with departments,** showing **department details** for each employee.

Conclusion

In this section, we explored:

✓ **How $match filters data for better performance**.

✓ **Using $group to aggregate data with sum, count, and average**.

✓ **How $project customizes output by renaming, selecting, and modifying fields**.

✓ **Using $lookup to join multiple collections** for relational queries.

✓ **How Airbnb optimizes MongoDB queries for analytics and reporting**.

PERFORMING COMPLEX DATA ANALYSIS IN MONGODB

CHAPTER 1: INTRODUCTION TO COMPLEX DATA ANALYSIS IN MONGODB

1.1 Understanding Data Analysis in NoSQL Databases

In **relational databases (SQL)**, complex data analysis is performed using **JOINs, subqueries, and aggregate functions**. However, MongoDB, as a **NoSQL document-oriented database**, uses the **Aggregation Framework** to process and analyze large amounts of data efficiently.

✓ **Performs real-time analytics** without requiring external tools.
✓ **Handles complex queries** like grouping, filtering, and transformations.
✓ **Optimized for large-scale data processing** (e.g., IoT, e-commerce, finance).

1.2 Why Use Aggregation for Data Analysis?

✓ **Faster than traditional queries** – Aggregates data efficiently within MongoDB.
✓ **Eliminates post-processing** – Reduces the need for application-side computations.
✓ **Scales with big data** – Suitable for analyzing millions of records.

MongoDB's **Aggregation Framework** allows **data transformation and computation** using the **aggregate() method**.

CHAPTER 2: UNDERSTANDING THE AGGREGATION PIPELINE

2.1 What is the Aggregation Pipeline?

The **Aggregation Pipeline** is a **multi-stage query processing framework** in MongoDB. Instead of running multiple queries, data is **passed through stages** that filter, transform, and summarize the results.

✓ Each stage processes documents and **passes them to the next stage**.
✓ Stages use **operators** (e.g., $match, $group, $sort, $limit).
✓ Efficient for **large datasets** due to its **pipeline-based execution**.

2.2 Common Aggregation Stages

Stage	Description
$match	Filters documents (like find()).
$group	Groups documents by a field and applies aggregations.
$sort	Sorts documents in ascending or descending order.
$limit	Limits the number of documents returned.
$project	Selects specific fields to return.
$unwind	Breaks down arrays into separate documents.

Each stage is **executed in order**, allowing complex data transformations.

CHAPTER 3: FILTERING DATA USING $MATCH

3.1 Using $match to Filter Documents

The $match stage **filters documents** at the beginning of the pipeline, improving performance.

Example: Filtering Users Aged 25 and Above

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

✓ Retrieves **only users aged 25 and older,** reducing unnecessary processing.

Example: Filtering Orders from a Specific Date

```
db.orders.aggregate([
  { $match: { orderDate: { $gte: ISODate("2023-01-01") } } }
])
```

✓ Retrieves **orders placed after January 1, 2023**.

CHAPTER 4: GROUPING AND SUMMARIZING DATA USING $GROUP

4.1 Using $group to Aggregate Data

The $group stage **groups documents by a specified field** and applies **aggregation operators** like $sum, $avg, $min, and $max.

Example: Counting Users by Age Group

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

✓ Groups users **by age** and counts how many users belong to each group.

Example: Calculating Total Sales for Each Product

```
db.orders.aggregate([
```

```
  { $group: { _id: "$productId", totalSales: { $sum: "$amount" } } }
])
```

✓ Groups orders **by productId** and calculates **total sales per product**.

4.2 Using $group with Multiple Fields

Example: Summarizing Sales by Product and Year

```
db.orders.aggregate([
  { $group: {
    _id: { product: "$productId", year: { $year: "$orderDate" } },
    totalSales: { $sum: "$amount" }
  }}
])
```

✓ Groups sales **by product and year**, useful for **yearly financial reports**.

CHAPTER 5: SORTING AND LIMITING RESULTS USING $SORT AND $LIMIT

5.1 Using $sort to Arrange Data

The $sort stage arranges documents **in ascending (1) or descending (-1) order**.

Example: Sorting Users by Age (Descending)

```
db.users.aggregate([
  { $sort: { age: -1 } }
])
```

✓ Returns users **from oldest to youngest**.

5.2 Using $limit to Reduce Result Size

The $limit stage **restricts the number of documents returned,** useful for **pagination and performance optimization**.

Example: Getting the Top 5 Highest Sales

```
db.orders.aggregate([
  { $sort: { totalSales: -1 } },
  { $limit: 5 }
])
```

✓ Returns **only the top 5 products with the highest sales**.

CHAPTER 6: EXPANDING DATA USING $UNWIND

6.1 Using $unwind to Handle Arrays

The $unwind stage **breaks down array fields into separate documents,** allowing detailed analysis.

Example: Expanding Orders with Multiple Products

```
db.orders.aggregate([
  { $unwind: "$products" },
  { $group: { _id: "$products", totalOrders: { $sum: 1 } } }
])
```

✓ Converts an order with multiple products **into separate documents,** allowing **per-product sales analysis**.

Case Study: How a Retail Company Used MongoDB for Sales Analytics

Background

A retail company needed to analyze **customer orders and sales trends**. Challenges included:

✓ **Slow performance** in generating reports.
✓ **High computational load** for aggregating sales data.
✓ **Difficulty in identifying top-selling products and peak sales seasons**.

Solution: Using MongoDB Aggregation for Analysis

The company implemented MongoDB's **Aggregation Pipeline** to:

✓ **Filter sales data** by year using $match.
✓ **Group orders by product and calculate total sales** using $group.
✓ **Sort top-selling products** using $sort and $limit.

Results

- **Sales reports generated 5x faster**.
- **Identified best-selling products,** leading to **targeted marketing campaigns**.
- **Optimized inventory management,** reducing **overstock and shortages**.

By leveraging **MongoDB aggregation,** the company gained **real-time business insights,** improving decision-making and profitability.

Exercise

1. Write an aggregation query to **count users by country**.

2. Retrieve the **top 10 most expensive products** sorted by price.
3. Find the **total sales for each month** in 2023.
4. Unwind an **array of tags** in a blog collection and count the most used tags.

Conclusion

In this section, we explored:

✓ **How to filter data efficiently using $match.**

✓ **How to group and summarize data using $group.**

✓ **How to sort and limit results for analytics using $sort and $limit.**

✓ **How to handle array data using $unwind.**

ASSIGNMENT:

BUILD A REPORT USING THE AGGREGATION FRAMEWORK FOR SALES ANALYTICS

SOLUTION GUIDE: BUILD A REPORT USING THE AGGREGATION FRAMEWORK FOR SALES ANALYTICS

Step 1: Set Up the Project and Install Dependencies

1.1 Initialize a Node.js Project

mkdir sales-analytics

cd sales-analytics

npm init -y

1.2 Install Required Packages

npm install mongoose dotenv

✓ **Mongoose** – For MongoDB interaction.
✓ **dotenv** – To manage environment variables.

Step 2: Configure MongoDB Connection

2.1 Set Up MongoDB and Create a .env File

Modify **.env** file:

MONGO_URI=mongodb://localhost:27017/salesDB

PORT=5000

2.2 Create the Database Connection File

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;
```

Step 3: Define the Sales Schema

Create **models/Sale.js**:

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

const saleSchema = new mongoose.Schema({
  product: { type: String, required: true },
  category: { type: String, required: true },
  price: { type: Number, required: true },
  quantity: { type: Number, required: true },
  total: { type: Number, required: true },
  date: { type: Date, default: Date.now }
```

});

module.exports = mongoose.model('Sale', saleSchema);

✓ **Each sale includes product details, price, and quantity**.
✓ The **total field** stores price * quantity for faster analytics.

Step 4: Insert Sample Sales Data

Create **seed.js** to populate the database:

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

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

const salesData = [

{ product: "Laptop", category: "Electronics", price: 1200, quantity: 2, total: 2400, date: "2024-03-01" },

{ product: "Phone", category: "Electronics", price: 800, quantity: 5, total: 4000, date: "2024-03-02" },

{ product: "Headphones", category: "Accessories", price: 200, quantity: 10, total: 2000, date: "2024-03-03" },

{ product: "Monitor", category: "Electronics", price: 300, quantity: 3, total: 900, date: "2024-03-04" },

{ product: "Keyboard", category: "Accessories", price: 100, quantity: 6, total: 600, date: "2024-03-05" }

];

```
const seedDatabase = async () => {
  try {
    await Sale.insertMany(salesData);
    console.log("Sample sales data inserted!");
  } catch (error) {
    console.error("Error seeding database:", error);
  } finally {
    mongoose.connection.close();
  }
};

seedDatabase();
```

Run the script to insert sample sales data:

```
node seed.js
```

✓ Populates the database with initial sales records.

Step 5: Build the Sales Analytics Report Using Aggregation

Create **reports.js**:

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

const generateSalesReport = async () => {
```

```
  try {
    const report = await Sale.aggregate([
      {
        $group: {
          _id: "$category",
          totalSales: { $sum: "$total" },
          totalQuantity: { $sum: "$quantity" },
          averageSalePrice: { $avg: "$price" }
        }
      },
      { $sort: { totalSales: -1 } }
    ]);

    console.log("Sales Report by Category:", report);
  } catch (error) {
    console.error("Error generating sales report:", error);
  } finally {
    mongoose.connection.close();
  }
};

generateSalesReport();
```

✓ **Groups sales by category** and calculates:

- **Total revenue per category** (totalSales).
- **Total units sold** (totalQuantity).
- **Average price per sale** (averageSalePrice).
  ✓ **Sorts categories by total sales (highest to lowest).**

Run the report:

node reports.js

Step 6: Additional Reports Using Aggregation

6.1 Find the Best-Selling Product

Modify **reports.js** to include:

```
const bestSellingProduct = async () => {
  try {
    const bestProduct = await Sale.aggregate([
      {
        $group: {
          _id: "$product",
          totalQuantity: { $sum: "$quantity" }
        }
      },
      { $sort: { totalQuantity: -1 } },
      { $limit: 1 }
    ]);
```

```
    console.log("Best-Selling Product:", bestProduct);
  } catch (error) {
    console.error("Error finding best-selling product:", error);
  } finally {
    mongoose.connection.close();
  }
};

bestSellingProduct();
```

✓ Groups sales **by product** and finds the **most sold product**.

Run the report:

```
node reports.js
```

6.2 Calculate Monthly Sales Trends

Modify **reports.js** to include:

```
const monthlySalesTrend = async () => {
  try {
    const trends = await Sale.aggregate([
      {
        $group: {
          _id: { $month: "$date" },
```

```
            totalRevenue: { $sum: "$total" }
          }
        },
        { $sort: { _id: 1 } }
      ]);

      console.log("Monthly Sales Trends:", trends);
    } catch (error) {
      console.error("Error generating monthly trends:", error);
    } finally {
      mongoose.connection.close();
    }
};

monthlySalesTrend();
```

✓ Groups sales **by month** to analyze trends.

Run the report:

node reports.js

Case Study: How an Online Retailer Used MongoDB Aggregation for Sales Insights

Background

An online retailer needed **real-time analytics** to track sales performance.

Challenges

- **Slow query performance** when calculating revenue.
- **Inefficient reporting** due to large data sets.
- **Lack of sales insights** for strategic planning.

Solution: Implementing Aggregation Framework

✓ Used $group to calculate **category-wise revenue**.
✓ Implemented $sort and $limit to find **top-selling products**.
✓ Automated **monthly trend analysis** for forecasting.

Results

- **40% reduction in reporting time,** improving decision-making.
- **Increased sales performance,** identifying high-revenue categories.
- **Better stock management,** reducing unsold inventory.

This case study highlights how **aggregation enhances data analytics and business intelligence**.

Exercise

1. Modify the sales report to include **average revenue per order**.
2. Find the **lowest-selling product** using the aggregation framework.
3. Create an aggregation query to **analyze daily sales performance**.

Conclusion

✓ **MongoDB's Aggregation Framework** is powerful for **real-time sales analytics**.
✓ **Optimized reporting queries** improve database efficiency.
✓ **Sales insights help businesses make data-driven decisions**.