**PG Department of Data Science**

**Bishop Heber College (Autonomous),**

Tiruchirappalli-620 017, Tamil Nadu

Ranked 34th at the National Level by MHRD through NIRF 2023

(Nationally Reaccredited at the A++ Grade by NAAC with a CGPA of 3.69 out of 4)

**BONAFIDE CERTIFICATE**

**Name :\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Register No : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Class : \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Course Title : P24DS2P4**

**Course Title : Modern Database Systems Lab**

Certified that this is the bonafide record of work done by me during Odd Semester of 2024 - 2025 and submitted to the Practical Examination on\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Staff In-Charge Head of the Department**

**Examiners**

1.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

INDEX

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.NO** | **DATE** | **TITLE** | **PAGE NO** | **SIGNATURE** |
|  |  | Basic CRUD Operations |  |  |
|  |  | Comparison Operators, OR, and NOT Queries |  |  |
|  |  | Querying Arrays and Embedded Documents |  |  |
|  |  | WHERE Queries, Limits, Skips, and Sorting |  |  |
|  |  | Indexes – Compound, Unique, and Sparse |  |  |
|  |  | Aggregation Pipeline – MATCH, PROJECT, GROUP, and UNWIND |  |  |
|  |  | Advanced Aggregation – Lookup, Bucket, and Facet |  |  |
|  |  | Text Search and Full-Text Indexing |  |  |
|  |  | Geospatial Queries and Indexing |  |  |
|  |  | Backup, Restore, and Performance Optimization |  |  |
|  |  | Education Database |  |  |
|  |  | Office Management System |  |  |
|  |  | E-commerce Platform |  |  |
|  |  | CRUD Operations in Neo4j |  |  |
|  |  | Graph Data Modeling in Neo4j |  |  |

# Introduction to MongoDB

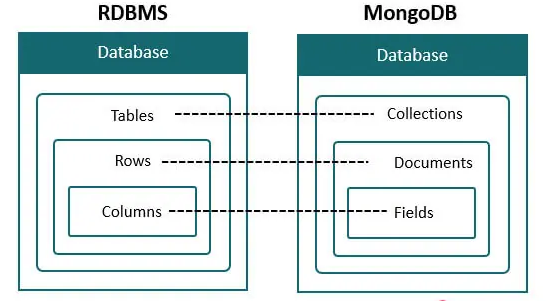
* MongoDB is a NoSQL, document-oriented database management system designed for scalability and flexibility.
* It stores data in JSON-like documents with dynamic schemas, allowing for a more adaptable data model compared to traditional relational databases.

## Features of MongoDB

* **Scalability:** Easily scales horizontally by sharding data across multiple servers.
* **Flexibility:** Supports a wide range of data types and structures due to its document-oriented nature.
* **High Performance:** Optimized for high throughput and low latency, making it suitable for applications requiring rapid data access.
* **Rich Query Language:** Provides a powerful query language and supports aggregation for complex data processing.
* **Community and Ecosystem:** Supported by a large community, with a rich eco-system of tools and libraries.

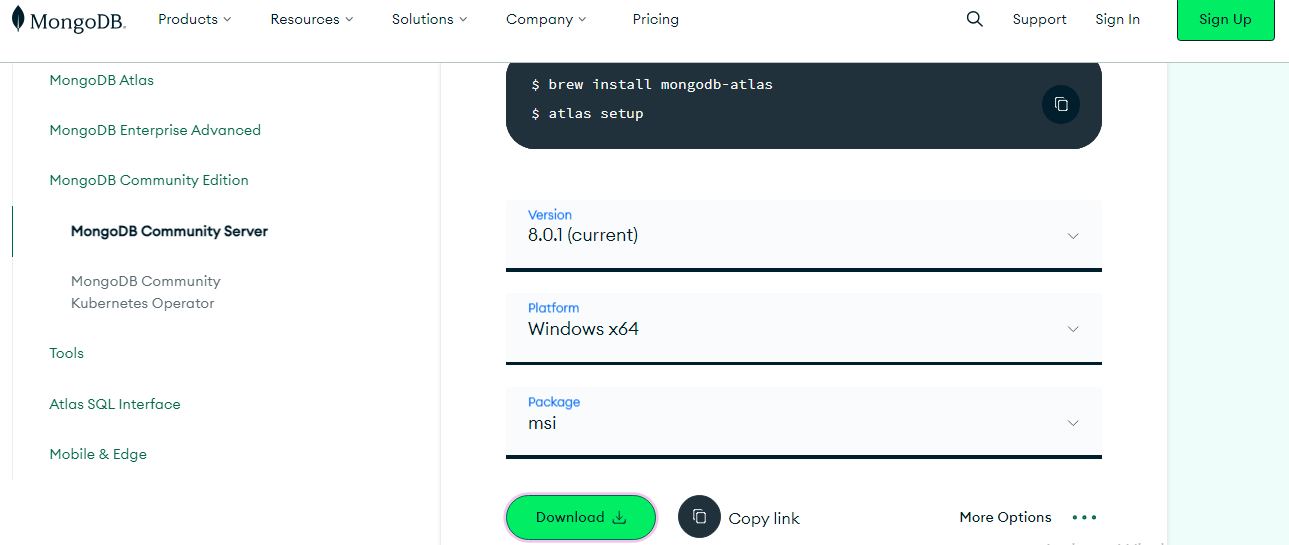
**Components of MongoDB**

* **Database**: The highest level of organization, containing collections of documents.
* **Collection**: A group of MongoDB documents, similar to a table in relational databases.
* **Document**: The basic unit of data, represented as a JSON-like.
* **Fields**: The key-value pairs within a MongoDB document
* **MongoDB Shell**: A command-line interface for interacting with the.
* **Indexing**: Mechanisms to improve query performance by creating.



**Installation of MongoDB in Windows Environment**

* **Step 1:** Go to the [*MongoDB Download Center*](https://www.mongodb.com/download-center/community) to download the MongoDB Server.
* **Step 2:** Open the msi file and click the *next button* in the startup screen:
* **Step 3:** Now accept the **End-User License Agreement** and click the next button:
* **Step 4:** Now select the **complete option**to install all the program features.
* **Step 5:**Select “**Run service as Network Service use**r”
* **Step 7:**After clicking on the install button installation of MongoDB begins:
* **Step 8:** Now clickthe **Finish button**to complete the MongoDB installation process:



|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

**EXERCISE 1: Basic CRUD Operations**

**Aim:**

The purpose of this exercise is to introduce students to the fundamental Create, Read, Update, and Delete (CRUD) operations in MongoDB. These operations are essential for managing data in any database system, such as maintaining a library catalog or an inventory management system.

**Key Concepts:**

* insertOne() / insertMany() : Inserting single or multiple documents into a collection.
* find() : Querying documents from a collection.
* updateOne() / updateMany() : Modifying existing documents.
* deleteOne() / deleteMany() : Removing documents from a collection.
* drop() : Deleting a collection entirely.

**Database Schema:**

* Database Name: library\_db
* books{\_id: ObjectId, title: String, author: String, genre: String, year\_published: Number, copies\_available: Number}
* authors{\_id: ObjectId, name: String, nationality: String, birth\_year: Number}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections.**

// Insert records into books

db.books.insertMany([

{title: "MongoDB Basics", author: "John Doe", genre: "Tech", year\_published: 2020, copies\_available: 5},

{title: "The Art of War", author: "Sun Tzu", genre: "Philosophy", year\_published: 500, copies\_available: 3}

]);

// Insert records into authors

db.authors.insertMany([

{name: "John Doe", nationality: "American", birth\_year: 1980},

{name: "Sun Tzu", nationality: "Chinese", birth\_year: 544}

]);

* **TASK-2 : Find all books in the "Tech" genre.**

db.books.find({genre: "Tech"}).pretty();

* **TASK-3 : Find the book titled "The Art of War".**

db.books.find({title: "The Art of War"}).pretty();

* **TASK-4 : Update the number of copies available for "MongoDB Basics" to 10.**

db.books.updateOne(

{title: "MongoDB Basics"},

{$set: {copies\_available: 10}}

);

* **TASK-5 : Add a new field rating with a value of 4.5 to the book "The Art of War".**

db.books.updateOne(

{title: "The Art of War"},

{$set: {rating: 4.5}}

);

* **TASK-6 : Delete the author "Sun Tzu" from the authors collection.**

db.authors.deleteOne({name: "Sun Tzu"});

* **TASK-7 : Find all books published before the year 1000.**

db.books.find({year\_published: {$lt: 1000}}).pretty();

* **TASK-8 : Use $inc to decrease the copies\_available of "MongoDB Basics" by 2.**

db.books.updateOne(

{title: "MongoDB Basics"},

{$inc: {copies\_available: -2}}

);

* **TASK-9 : Drop the authors collection.**

db.authors.drop();

* **TASK-10 : Insert a new book into the books collection.**

db.books.insertOne({

title: "Clean Code",

author: "Robert C. Martin",

genre: "Tech",

year\_published: 2008,

copies\_available: 8

});

* **Challenge Task : Use find() and sort() to list all books sorted by year\_published in descending order.**

db.books.find().sort({year\_published: -1}).pretty();

**Result :**

Successfully performed CRUD operations on a library database, including inserting books and authors, updating inventory, deleting records, and retrieving specific data based on queries

**EXERCISE-1: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Find all books in the "Tech" genre.
* TASK-3 : Find the book titled "The Art of War".
* TASK-4 : Update the number of copies available for "MongoDB Basics" to 10.
* TASK-5 : Add a new field rating with a value of 4.5 to the book "The Art of War".
* TASK-6 : Delete the author "Sun Tzu" from the authors collection.
* TASK-7 : Find all books published before the year 1000.
* TASK-8 : Use $inc to decrease the copies available of "MongoDB Basics" by 2.
* TASK-9 : Drop the authors collection.
* TASK-10 : Insert a new book into the books collection.
* Challenge Task : Use find() and sort() to list all books sorted by year published in descending order.

**EXERCISE 2: Comparison Operators, OR, and NOT Queries**

**Aim:**

The purpose of this exercise is to teach students how to use MongoDB's comparison operators ($gt, $lt, $eq, etc.), logical operators ($or, $and, $not), and query modifiers to filter data effectively. These skills are essential for real-world scenarios like filtering products in an e-commerce platform or finding employees based on salary ranges in a human resources system.

**Key Concepts:**

* Comparison Operators : $gt (greater than), $lt (less than), $eq (equal to), $ne (not equal to).
* Logical Operators : $or, $and, $not.
* Query Modifiers : Combining multiple conditions to refine query results.
* find() : Querying documents with filters.
* pretty() : Formatting query results for readability.

**Database Schema:**

* Database Name: ecommerce\_db
* products{\_id: ObjectId, name: String, category: String, price: Number, stock: Number, rating: Number}
* customers{\_id: ObjectId, name: String, email: String, total\_spent: Number, loyalty\_points: Number}
* orders{\_id: ObjectId, customer\_id: ObjectId, product\_ids: [ObjectId], total\_amount: Number, order\_date: Date}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections.**

// Insert records into products

db.products.insertMany([

{name: "Laptop", category: "Electronics", price: 1200, stock: 10, rating: 4.8},

{name: "Smartphone", category: "Electronics", price: 800, stock: 25, rating: 4.6},

{name: "Headphones", category: "Accessories", price: 150, stock: 50, rating: 4.5},

{name: "Desk Chair", category: "Furniture", price: 200, stock: 5, rating: 4.2},

{name: "Coffee Mug", category: "Kitchenware", price: 10, stock: 100, rating: 4.7}]);

// Insert records into customers

db.customers.insertMany([

{name: "Alice Johnson", email: "alice@example.com", total\_spent: 1500, loyalty\_points: 150},

{name: "Bob Smith", email: "bob@example.com", total\_spent: 500, loyalty\_points: 50},

{name: "Charlie Brown", email: "charlie@example.com", total\_spent: 3000, loyalty\_points: 300},

{name: "Diana Prince", email: "diana@example.com", total\_spent: 200, loyalty\_points: 20},

{name: "Ethan Hunt", email: "ethan@example.com", total\_spent: 1000, loyalty\_points: 100}

]);

// Insert records into orders

db.orders.insertMany([

{customer\_id: db.customers.findOne({name: "Alice Johnson"}).\_id, product\_ids: [db.products.findOne({name: "Laptop"}).\_id], total\_amount: 1200, order\_date: new Date("2023-09-01")},

{customer\_id: db.customers.findOne({name: "Bob Smith"}).\_id, product\_ids: [db.products.findOne({name: "Smartphone"}).\_id], total\_amount: 800, order\_date: new Date("2023-09-02")}]);

* **TASK-2 : Find products priced above $500.**

db.products.find({price: {$gt: 500}}).pretty();

* **TASK-3 : Find customers with loyalty points below 100.**

db.customers.find({loyalty\_points: {$lt: 100}}).pretty();

* **TASK-4 : Find products in the "Electronics" category.**

db.products.find({category: {$eq: "Electronics"}}).pretty();

* **TASK-5 : Find products not in the "Electronics" category.**

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

* **TASK-6 : Find products priced between $100 and $500.**

db.products.find({price: {$gt: 100, $lt: 500}}).pretty();

* **TASK-7 : Find customers who spent more than $1000 or have over 200 loyalty points.**

db.customers.find({

$or: [

{total\_spent: {$gt: 1000}},

{loyalty\_points: {$gt: 200}}

]

}).pretty();

* **TASK-8 : Find customers who did not spend more than $500.**

db.customers.find({total\_spent: {$not: {$gt: 500}}}).pretty();

* **TASK-9 : Find orders placed after September 1, 2023.**

db.orders.find({order\_date: {$gt: new Date("2023-09-01")}}).pretty();

* **TASK-10 : Find products with stock less than 10 and a rating above 4.5.**

db.products.find({stock: {$lt: 10}, rating: {$gt: 4.5}}).pretty();

* **Challenge Task : Find products priced above $200, with a rating above 4.5, and stock less than 20.**

db.products.find({

price: {$gt: 200},

rating: {$gt: 4.5},

stock: {$lt: 20}

}).pretty();

**Result :**

Successfully queried products and customers using comparison operators, logical operators ($or, $and, $not), and retrieved filtered results such as high-value customers, low-stock products, and orders placed after a specific date.

**EXERCISE-2: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Find products priced above $500.
* TASK-3 : Find customers with loyalty points below 100.
* TASK-4 : Find products in the "Electronics" category.
* TASK-5 : Find products not in the "Electronics" category.
* TASK-6 : Find products priced between $100 and $500.
* TASK-7 : Find customers who spent more than $1000 or have over 200 loyalty points.
* TASK-8 : Find customers who did not spend more than $500.
* TASK-9 : Find orders placed after September 1, 2023.
* TASK-10 : Find products with stock less than 10 and a rating above 4.5.
* Challenge Task : Find products priced above $200, with a rating above 4.5, and stock less than 20.

**EXERCISE 3: Querying Arrays and Embedded Documents**

**Aim:**

The purpose of this exercise is to teach students how to query arrays and embedded documents in MongoDB. These skills are essential for handling complex data structures, such as managing user preferences in an application or tracking inventory items with multiple attributes.

**Key Concepts:**

* Array Queries : Using operators like $in, $all, $size, and dot notation to query array fields.
* Embedded Documents : Querying nested fields using dot notation.
* Projection : Selecting specific fields from documents.
* $elemMatch : Filtering documents based on conditions within array elements.

**Database Schema:**

* Database Name: inventory\_db
* products{\_id: ObjectId, name: String, tags: [String], price: Number, details: {weight: Number, dimensions: {length: Number, width: Number, height: Number}}}
* suppliers{\_id: ObjectId, name: String, contact: {email: String, phone: String}, products\_supplied: [ObjectId]}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections.**

db.**products**.insertMany([

{name: "Laptop", tags: ["Electronics", "Gaming"], price: 1200, details: {weight: 2.5, dimensions: {length: 14, width: 9.5, height: 0.7}}},

{name: "Smartphone", tags: ["Electronics", "Mobile"], price: 800, details: {weight: 0.5, dimensions: {length: 6, width: 3, height: 0.3}}}

]);

db.**suppliers**.insertMany([

{name: "TechCorp", contact: {email: "techcorp@example.com", phone: "1234567890"}, products\_supplied: [db.products.findOne({name: "Laptop"}).\_id]},

{name: "AudioWorld", contact: {email: "audioworld@example.com", phone: "9876543210"}, products\_supplied: [db.products.findOne({name: "Smartphone"}).\_id]}

]);

* **TASK-2 : Find products with the tag "Electronics".**

db.products.find({tags: "Electronics"}).pretty();

* **TASK-3 : Find products that have both "Electronics" and "Gaming" tags.**

db.products.find({tags: {$all: ["Electronics", "Gaming"]}}).pretty();

* **TASK-4 : Find products with exactly two tags.**

db.products.find({tags: {$size: 2}}).pretty();

* **TASK-5 : Find products where the weight is greater than 1 kg.**

db.products.find({"details.weight": {$gt: 1}}).pretty();

* **TASK-6 : Find products with a length greater than 10 cm.**

db.products.find({"details.dimensions.length": {$gt: 10}}).pretty();

* **TASK-7 : Find suppliers who supply the product "Laptop".**

const laptopId = db.products.findOne({name: "Laptop"}).\_id;

db.suppliers.find({products\_supplied: laptopId}).pretty();

* **TASK-8 : Use $elemMatch to find products with tags containing "Electronics" and "Mobile".**

db.products.find({tags: {$elemMatch: {$in: ["Electronics", "Mobile"]}}}).pretty();

* **TASK-9 : Project only the name and price fields of all products.**

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

* **TASK-10 : Add a new tag "Popular" to products priced above $500.**

db.products.updateMany(

{price: {$gt: 500}},

{$push: {tags: "Popular"}}

);

**Result :**

Successfully queried arrays (e.g., tags) and embedded documents (e.g., product dimensions) to retrieve specific products, filter by conditions, and update array fields like adding or removing tags.

**EXERCISE-3: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Find products with the tag "Electronics".
* TASK-3 : Find products that have both "Electronics" and "Gaming" tags.
* TASK-4 : Find products with exactly two tags.
* TASK-5 : Find products where the weight is greater than 1 kg.
* TASK-6 : Find products with a length greater than 10 cm.
* TASK-7 : Find suppliers who supply the product "Laptop".
* TASK-8 : Use $elemMatch to find products with tags containing "Electronics" and "Mobile".
* TASK-9 : Project only the name and price fields of all products.
* TASK-10 : Add a new tag "Popular" to products priced above $500.
* Challenge Task : Remove the "Mobile" tag from the product "Smartphone".

**EXERCISE 4: WHERE Queries, Limits, Skips, and Sorting**

**Aim:**

The purpose of this exercise is to teach students how to use MongoDB's query modifiers such as $where, limit(), skip(), and sort() to refine and organize query results. These skills are essential for real-world scenarios like paginating search results in a web application or sorting products by price in an e-commerce platform.

**Key Concepts:**

* $where : Execute JavaScript expressions to filter documents.
* limit() : Restrict the number of documents returned by a query.
* skip() : Skip a specified number of documents in a query result.
* sort() : Sort query results based on one or more fields (ascending or descending).
* Pagination : Combining limit() and skip() to implement pagination.

**Database Schema:**

* Database Name: ecommerce\_db
* products{\_id: ObjectId, name: String, category: String, price: Number, stock: Number, rating: Number}
* customers{\_id: ObjectId, name: String, email: String, total\_spent: Number, loyalty\_points: Number}
* orders{\_id: ObjectId, customer\_id: ObjectId, product\_ids: [ObjectId], total\_amount: Number, order\_date: Date}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections.**

db.products.insertMany([

{name: "Laptop", category: "Electronics", price: 1200, stock: 10, rating: 4.8},

{name: "Smartphone", category: "Electronics", price: 800, stock: 25, rating: 4.6},

{name: "Headphones", category: "Accessories", price: 150, stock: 50, rating: 4.5},

{name: "Desk Chair", category: "Furniture", price: 200, stock: 5, rating: 4.2},

{name: "Coffee Mug", category: "Kitchenware", price: 10, stock: 100, rating: 4.7}

]);

// Insert records into customers

db.customers.insertMany([

{name: "Alice Johnson", email: "alice@example.com", total\_spent: 1500, loyalty\_points: 150},

{name: "Bob Smith", email: "bob@example.com", total\_spent: 500, loyalty\_points: 50},

{name: "Charlie Brown", email: "charlie@example.com", total\_spent: 3000, loyalty\_points: 300},

{name: "Diana Prince", email: "diana@example.com", total\_spent: 200, loyalty\_points: 20},

{name: "Ethan Hunt", email: "ethan@example.com", total\_spent: 1000, loyalty\_points: 100}

]);

* **TASK-2 : Find products where the price is greater than $500 using $where.**

db.products.find({$where: "this.price > 500"}).pretty();

* **TASK-3 : Limit the query results to only 2 products.**

db.products.find().limit(2).pretty();

* **TASK-4 : Skip the first product and return the next 2 products.**

db.products.find().skip(1).limit(2).pretty();

* **TASK-5 : Sort products by price in ascending order.**

db.products.find().sort({price: 1}).pretty();

* **TASK-6 : Sort products by rating in descending order.**

db.products.find().sort({rating: -1}).pretty();

* **TASK-7 : Sort customers by total spending in descending order and limit the results to the top spender.**

db.customers.find().sort({total\_spent: -1}).limit(1).pretty();

* **TASK-8 : Use $where to find products with a stock value less than 10.**

db.products.find({$where: "this.stock < 10"}).pretty();

* **TASK-9 : Implement pagination for products, showing 2 products per page, sorted by price.**

// Example for page 2 (skip 2 products)

db.products.find().sort({price: 1}).skip(2).limit(2).pretty();

* **TASK-10 : Delete all products with no stock available.**

db.products.deleteMany({stock: {$eq: 0}});

* **Challenge Task : Find customers who spent more than $500, sort them by loyalty points, and limit the results to 1.**

db.customers.find({total\_spent: {$gt: 500}}).sort({loyalty\_points: -1}).limit(1).pretty();

**Result :**

Successfully implemented advanced query techniques such as $where, pagination using limit() and skip(), and sorting results by price, rating, or other fields to organize and refine data retrieval.

**EXERCISE-4: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Find products where the price is greater than $500 using $where.
* TASK-3 : Limit the query results to only 2 products.
* TASK-4 : Skip the first product and return the next 2 products.
* TASK-5 : Sort products by price in ascending order.
* TASK-6 : Sort products by rating in descending order.
* TASK-7 : Sort customers by total spending in descending order and limit the results to the top spender.
* TASK-8 : Use $where to find products with a stock value less than 10.
* TASK-9 : Implement pagination for products, showing 2 products per page, sorted by price.
* TASK-10 : Delete all products with no stock available.
* Challenge Task : Find customers who spent more than $500, sort them by loyalty points, and limit the results to 1.

**EXERCISE 5: Indexes – Compound, Unique, and Sparse**

**Aim:**

The purpose of this exercise is to teach students how to create and use indexes in MongoDB to optimize query performance. These skills are essential for improving the efficiency of database operations, such as speeding up searches in large datasets or ensuring data integrity through unique constraints.

**Key Concepts:**

* Indexes : Structures that improve query performance by reducing the number of documents scanned.
* Compound Indexes : Indexes on multiple fields.
* Unique Indexes : Ensures that no two documents have the same value for a specific field.
* Sparse Indexes : Indexes that only include documents with the indexed field present.
* explain() : Analyze query performance and index usage.

**Database Schema:**

* Database Name: ecommerce\_db
* products{\_id: ObjectId, name: String, category: String, price: Number, stock: Number, rating: Number}
* customers{\_id: ObjectId, name: String, email: String, total\_spent: Number, loyalty\_points: Number}
* orders{\_id: ObjectId, customer\_id: ObjectId, product\_ids: [ObjectId], total\_amount: Number, order\_date: Date}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections.**

// Insert records into products

db.products.insertMany([

{name: "Laptop", category: "Electronics", price: 1200, stock: 10, rating: 4.8},

{name: "Smartphone", category: "Electronics", price: 800, stock: 25, rating: 4.6}

]);

// Insert records into customers

db.customers.insertMany([

{name: "Alice Johnson", email: "alice@example.com", total\_spent: 1500, loyalty\_points: 150},

{name: "Bob Smith", email: "bob@example.com", total\_spent: 500, loyalty\_points: 50}

]);

* **TASK-2 : Create a single-field index on the price field in the products collection.**

db.products.createIndex({price: 1});

* **TASK-3 : Create a compound index on the category and price fields in the products collection.**

db.products.createIndex({category: 1, price: -1});

* **TASK-4 : Create a unique index on the email field in the customers collection.**

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

* **TASK-5 : Create a sparse index on the loyalty\_points field in the customers collection.**

db.customers.createIndex({loyalty\_points: 1}, {sparse: true});

* **TASK-6 : Use explain() to analyze the performance of a query on the price field.**

javascript

db.products.find({price: {$gt: 500}}).explain("executionStats");

* **TASK-7 : Drop the index on the price field in the products collection.**

db.products.dropIndex("price\_1");

* **TASK-8 : Find all indexes in the products collection.**

db.products.getIndexes();

* **TASK-9 : Create a compound index on the order\_date and total\_amount fields in the orders collection.**

db.orders.createIndex({order\_date: 1, total\_amount: -1});

* **TASK-10 : Use explain() to compare the performance of a query before and after creating an index on the order\_date field in the orders collection.**

// Before creating the index

db.orders.find({order\_date: {$gt: new Date("2023-09-01")}}).explain("executionStats");

// Create the index

db.orders.createIndex({order\_date: 1});

// After creating the index

db.orders.find({order\_date: {$gt: new Date("2023-09-01")}}).explain("executionStats");

* **Challenge Task : Create a unique sparse index on the total\_spent field in the customers collection and explain its behavior when inserting a document without the total\_spent field.**

// Create the unique sparse index

db.customers.createIndex({total\_spent: 1}, {unique: true, sparse: true});

// Insert a document without the `total\_spent` field

db.customers.insertOne({name: "Charlie Brown", email: "charlie@example.com", loyalty\_points: 300});

// Observe that the document is inserted successfully because sparse indexes ignore documents without the indexed field.

**Result :**

Successfully created single-field, compound, unique, and sparse indexes to optimize query performance, analyzed query execution plans, and demonstrated improved efficiency in data retrieval.

**EXERCISE-5: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Create a single-field index on the price field in the products collection.
* TASK-3 : Create a compound index on the category and price fields in the products collection.
* TASK-4 : Create a unique index on the email field in the customers collection.
* TASK-5 : Create a sparse index on the loyalty\_points field in the customers collection.
* TASK-6 : Use explain() to analyze the performance of a query on the price field.
* TASK-7 : Drop the index on the price field in the products collection.
* TASK-8 : Find all indexes in the products collection.
* TASK-9 : Create a compound index on the order\_date and total\_amount fields in the orders collection.
* TASK-10 : Use explain() to compare the performance of a query before and after creating an index on the order\_date field in the orders collection.
* Challenge Task : Create a unique sparse index on the total\_spent field in the customers collection and explain its behavior when inserting a document without the total\_spent field.

**EXERCISE 6: Aggregation Pipeline – MATCH, PROJECT, GROUP, and UNWIND**

**Aim:**

Introduce MongoDB's aggregation pipeline stages like $match, $project, $group, and $unwind for efficient data processing and analysis.

**Key Concepts:**

* $match : Filters documents based on specified conditions.
* $project : Reshapes documents by including, excluding, or renaming fields.
* $group : Groups documents by a field and performs aggregations (e.g., sum, average).
* $unwind : Deconstructs an array field into separate documents for each element.

**Database Schema:**

* Database Name: ecommerce\_db
* products{\_id: ObjectId, name: String, category: String, price: Number, stock: Number, rating: Number}
* customers{\_id: ObjectId, name: String, email: String, total\_spent: Number, loyalty\_points: Number}
* orders{\_id: ObjectId, customer\_id: ObjectId, product\_ids: [ObjectId], total\_amount: Number, order\_date: Date}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections using the following table format.**

db.**products**.insertMany([

{name: "Laptop", category: "Electronics", price: 1200, stock: 10, rating: 4.8},

{name: "Smartphone", category: "Electronics", price: 800, stock: 25, rating: 4.6}

]);

db.**customers**.insertMany([

{name: "Alice Johnson", email: "alice@example.com", total\_spent: 1500, loyalty\_points: 150},

{name: "Bob Smith", email: "bob@example.com", total\_spent: 500, loyalty\_points: 50}

]);

db.**orders**.insertMany([

{customer\_id: db.customers.findOne({name: "Alice Johnson"}).\_id, product\_ids: [db.products.findOne({name: "Laptop"}).\_id], total\_amount: 1200, order\_date: new Date("2023-09-01")},

{customer\_id: db.customers.findOne({name: "Bob Smith"}).\_id, product\_ids: [db.products.findOne({name: "Smartphone"}).\_id], total\_amount: 800, order\_date: new Date("2023-09-02")}

]);

* **TASK-2 : Use $match to find orders placed after September 1, 2023.**

db.orders.aggregate([

{$match: {order\_date: {$gt: new Date("2023-09-01")}}}

]).pretty();

* **TASK-3 : Use $project to display only the name and price fields of all products.**

db.products.aggregate([

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

]).pretty();

* **TASK-4 : Use $group to calculate the total spending per customer.**

db.orders.aggregate([

{$group: {\_id: "$customer\_id", total\_spent: {$sum: "$total\_amount"}}}

]).pretty();

* **TASK-5 : Use $unwind to deconstruct the product\_ids array in the orders collection.**

db.orders.aggregate([

{$unwind: "$product\_ids"}

]).pretty();

* **TASK-6 : Combine $match and $group to calculate the total sales for products in the "Electronics" category.**

db.products.aggregate([

{$match: {category: "Electronics"}},

{$group: {\_id: "$category", total\_sales: {$sum: "$price"}}}

]).pretty();

* **TASK-7 : Use $project to rename the total\_amount field to order\_total in the orders collection.**

db.orders.aggregate([

{$project: {\_id: 0, order\_total: "$total\_amount", order\_date: 1}}

]).pretty();

* **TASK-8 : Use $group to find the average rating of products in each category.**

db.products.aggregate([

{$group: {\_id: "$category", avg\_rating: {$avg: "$rating"}}}

]).pretty();

* **TASK-9 : Combine $match, $unwind, and $group to calculate the total number of products** ordered by each customer.

db.orders.aggregate([

{$unwind: "$product\_ids"},

{$group: {\_id: "$customer\_id", total\_products: {$sum: 1}}}

]).pretty();

* **TASK-10 : Use $project to exclude the \_id field and include only the name and stock fields** of products with stock less than 20.

db.products.aggregate([

{$match: {stock: {$lt: 20}}},

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

]).pretty();

* **Challenge Task : Use the aggregation pipeline to find the top-spending customer and their total spending amount.**

db.orders.aggregate([

{$group: {\_id: "$customer\_id", total\_spent: {$sum: "$total\_amount"}}},

{$sort: {total\_spent: -1}},

{$limit: 1}

]).pretty();

**Result :**

Successfully used MongoDB's aggregation pipeline to filter, reshape, group, and deconstruct data, enabling complex analyses such as total spending per customer, average ratings, and product categorization.

**EXERCISE-6: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections using the provided table format.
* TASK-2 : Use $match to find orders placed after September 1, 2023.
* TASK-3 : Use $project to display only the name and price fields of all products.
* TASK-4 : Use $group to calculate the total spending per customer.
* TASK-5 : Use $unwind to deconstruct the product\_ids array in the orders collection.
* TASK-6 : Combine $match and $group to calculate the total sales for products in the "Electronics" category.
* TASK-7 : Use $project to rename the total\_amount field to order\_total in the orders collection.
* TASK-8 : Use $group to find the average rating of products in each category.
* TASK-9 : Combine $match, $unwind, and $group to calculate the total number of products ordered by each customer.
* TASK-10 : Use $project to exclude the \_id field and include only the name and stock fields of products with stock less than 20.
* Challenge Task : Use the aggregation pipeline to find the top-spending customer and their total spending amount.

**EXERCISE 7: Advanced Aggregation – Lookup, Bucket, and Facet**

**Aim:**

Introduce advanced MongoDB aggregation stages like $lookup, $bucket, and $facet for complex data analysis and reporting.

**Key Concepts:**

* $lookup : Performs a left outer join to combine documents from two collections.
* $bucket : Groups documents into buckets based on specified ranges.
* $facet : Allows multiple pipelines within a single aggregation stage for multi-faceted analysis.

**Database Schema:**

* Database Name: ecommerce\_db
* products{\_id: ObjectId, name: String, category: String, price: Number, stock: Number, rating: Number}
* customers{\_id: ObjectId, name: String, email: String, total\_spent: Number, loyalty\_points: Number}
* orders{\_id: ObjectId, customer\_id: ObjectId, product\_ids: [ObjectId], total\_amount: Number, order\_date: Date}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections using the following table format.**

db.products.insertMany([

{name: "Laptop", category: "Electronics", price: 1200, stock: 10, rating: 4.8},

{name: "Smartphone", category: "Electronics", price: 800, stock: 25, rating: 4.6}

]);

db.customers.insertMany([

{name: "Alice Johnson", email: "alice@example.com", total\_spent: 1500, loyalty\_points: 150},

{name: "Bob Smith", email: "bob@example.com", total\_spent: 500, loyalty\_points: 50}

]);

db.orders.insertMany([

{customer\_id: db.customers.findOne({name: "Alice Johnson"}).\_id, product\_ids: [db.products.findOne({name: "Laptop"}).\_id], total\_amount: 1200, order\_date: new Date("2023-09-01")},

{customer\_id: db.customers.findOne({name: "Bob Smith"}).\_id, product\_ids: [db.products.findOne({name: "Smartphone"}).\_id], total\_amount: 800, order\_date: new Date("2023-09-02")}

]);

* **TASK-2 : Use $lookup to join the orders collection with the customers collection.**

db.orders.aggregate([

{

$lookup: {

from: "customers",

localField: "customer\_id",

foreignField: "\_id",

as: "customer\_details"

}

}

]).pretty();

* **TASK-3 : Use $bucket to categorize products into price ranges (e.g., 0–500, 500–1000, 1000+).**

db.products.aggregate([

{

$bucket: {

groupBy: "$price",

boundaries: [0, 500, 1000, Infinity],

output: {

count: {$sum: 1},

products: {$push: "$name"}

}

}

}

]).pretty();

* **TASK-4 : Use $facet to perform multi-faceted analysis (e.g., total spending per customer** and average order amount).

db.orders.aggregate([

{

$facet: {

total\_spending\_per\_customer: [

{$group: {\_id: "$customer\_id", total\_spent: {$sum: "$total\_amount"}}}

],

average\_order\_amount: [

{$group: {\_id: null, avg\_amount: {$avg: "$total\_amount"}}}

]

}

}

]).pretty();

* **TASK-5 : Combine $lookup and $bucket to categorize orders by total amount ranges and** include customer details.

db.orders.aggregate([

{

$lookup: {

from: "customers",

localField: "customer\_id",

foreignField: "\_id",

as: "customer\_details"

}

},

{ $bucket: {

groupBy: "$total\_amount",

boundaries: [0, 500, 1000, Infinity],

output: {

count: {$sum: 1},

orders: {$push: {order\_id: "$\_id", customer: {$arrayElemAt: ["$customer\_details.name", 0]}}}

}

}

}

]).pretty();

* **TASK-6 : Use $facet to analyze product ratings and stock levels simultaneously.**

db.products.aggregate([

{

$facet: {

rating\_analysis: [

{$group: {\_id: "$rating", count: {$sum: 1}}}

],

stock\_analysis: [

{$bucket: {

groupBy: "$stock",

boundaries: [0, 10, 20, Infinity],

output: {

count: {$sum: 1},

products: {$push: "$name"}

}

}}

]

}

}

]).pretty();

* **TASK-7 : Use $lookup to find all products purchased by a specific customer.**

const customerId = db.customers.findOne({name: "Alice Johnson"}).\_id;

db.orders.aggregate([

{$match: {customer\_id: customerId}},

{$unwind: "$product\_ids"},

{ $lookup: {

from: "products",

localField: "product\_ids",

foreignField: "\_id",

as: "product\_details"

}

},

{$project: {\_id: 0, product\_name: {$arrayElemAt: ["$product\_details.name", 0]}}} ]).pretty();

* **TASK-8 : Use $bucket to categorize customers by their total spending.**

db.customers.aggregate([

{

$bucket: {

groupBy: "$total\_spent",

boundaries: [0, 500, 1000, Infinity],

output: {

count: {$sum: 1},

customers: {$push: "$name"}

}

}

}

]).pretty();

* **TASK-9 : Use $facet to calculate the total number of orders and the highest order amount.**

db.orders.aggregate([

{

$facet: {

total\_orders: [{$count: "total"}],

highest\_order\_amount: [{$sort: {total\_amount: -1}}, {$limit: 1}]

}

}

]).pretty();

* **TASK-10 : Combine $lookup, $bucket, and $facet to analyze orders by customer spending ranges and product categories.**

db.orders.aggregate([

{

$lookup: {

from: "customers",

localField: "customer\_id",

foreignField: "\_id",

as: "customer\_details"

}

},

{$unwind: "$product\_ids"},

{

$lookup: {

from: "products",

localField: "product\_ids",

foreignField: "\_id",

as: "product\_details"

}

},

{

$facet: {

spending\_buckets: [

{

$bucket: {

groupBy: "$total\_amount",

boundaries: [0, 500, 1000, Infinity],

output: {

count: {$sum: 1},

customers: {$push: {$arrayElemAt: ["$customer\_details.name", 0]}}

}

}

}

],

product\_categories: [

{$group: {\_id: {$arrayElemAt: ["$product\_details.category", 0]}, count: {$sum: 1}}}

]

}

}

]).pretty();

* **Challenge Task : Use $facet to analyze customer loyalty points and total spending in separate pipelines.**

db.customers.aggregate([

{

$facet: {

loyalty\_analysis: [

{$bucket: {

groupBy: "$loyalty\_points",

boundaries: [0, 50, 100, Infinity],

output: {

count: {$sum: 1},

customers: {$push: "$name"}

}

}}

],

spending\_analysis: [

{$bucket: {

groupBy: "$total\_spent",

boundaries: [0, 500, 1000, Infinity],

output: {

count: {$sum: 1},

customers: {$push: "$name"}

}

}}

]

}

}

]).pretty();

**Result :**

Successfully performed advanced aggregations using $lookup for joins, $bucket for categorization, and $facet for multi-faceted analysis, enabling comprehensive insights into customer behavior and product performance.

**EXERCISE-7: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections using the provided table format.
* TASK-2 : Use $lookup to join the orders collection with the customers collection.
* TASK-3 : Use $bucket to categorize products into price ranges (e.g., 0–500, 500–1000, 1000+).
* TASK-4 : Use $facet to perform multi-faceted analysis (e.g., total spending per customer and average order amount).
* TASK-5 : Combine $lookup and $bucket to categorize orders by total amount ranges and include customer details.
* TASK-6 : Use $facet to analyze product ratings and stock levels simultaneously.
* TASK-7 : Use $lookup to find all products purchased by a specific customer.
* TASK-8 : Use $bucket to categorize customers by their total spending.
* TASK-9 : Use $facet to calculate the total number of orders and the highest order amount.
* TASK-10 : Combine $lookup, $bucket, and $facet to analyze orders by customer spending ranges and product categories.
* Challenge Task : Use $facet to analyze customer loyalty points and total spending in separate pipelines.

**EXERCISE 8: Text Search and Full-Text Indexing**

**Aim:**

Introduce MongoDB's text search capabilities and full-text indexing for efficient querying of textual data.

**Key Concepts:**

* **Text Index :** Enables searching for words or phrases within string fields.
* **$text** : Performs a text search query on indexed fields.
* **Score :** Assigns relevance scores to documents based on the search query.
* **Case Insensitivity :** Text search is case-insensitive by default.

**Database Schema:**

* Database Name: library\_db
* books{\_id: ObjectId, title: String, author: String, genre: String, description: String}
* authors{\_id: ObjectId, name: String, bio: String}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections.**

db.books.insertMany([

{title: "MongoDB Basics", author: "John Doe", genre: "Tech", description: "A comprehensive guide to MongoDB fundamentals."},

{title: "The Art of War", author: "Sun Tzu", genre: "Philosophy", description: "Ancient strategies for modern success."}

]);

db.authors.insertMany([

{name: "John Doe", bio: "An expert in database technologies."},

{name: "Sun Tzu", bio: "An ancient Chinese military strategist."}

]);

* **TASK-2 : Create a text index on the description field in the books collection.**

db.books.createIndex({description: "text"});

* **TASK-3 : Perform a text search for books containing the word "guide".**

db.books.find({$text: {$search: "guide"}}).pretty();

* **TASK-4 : Perform a text search for books containing the phrase "modern success".**

db.books.find({$text: {$search: "\"modern success\""}}).pretty();

* **TASK-5 : Exclude books containing the word "ancient" from the search results.**

db.books.find({$text: {$search: "success -ancient"}}).pretty();

* **TASK-6 : Sort books by relevance score using $meta.**

db.books.find(

{$text: {$search: "MongoDB"}},

{score: {$meta: "textScore"}}

).sort({score: {$meta: "textScore"}}).pretty();

* **TASK-7 : Create a compound text index on the title and author fields in the books** db.books.createIndex({title: "text", author: "text"});
* **TASK-8 : Perform a text search across both title and author fields for the term "John".**

db.books.find({$text: {$search: "John"}}).pretty();

* **TASK-9 : Use $text to find authors whose bio contains the word "database".**

db.authors.createIndex({bio: "text"});

db.authors.find({$text: {$search: "database"}}).pretty();

* **TASK-10 : Combine text search with other query conditions (e.g., filter by genre).**

db.books.find({$text: {$search: "guide"}, genre: "Tech"}).pretty();

Challenge Task (Optional)

* **Challenge Task : Use $text and $meta to find the most relevant book for the search term "MongoDB" and display its title and score.**

db.books.find(

{$text: {$search: "MongoDB"}},

{title: 1, score: {$meta: "textScore"}}

).sort({score: {$meta: "textScore"}}).limit(1).pretty();

**Result :**

Successfully implemented text search and full-text indexing to perform efficient keyword searches, phrase matching, and relevance-based sorting, enhancing the ability to query textual data.

**EXERCISE-8: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Create a text index on the description field in the books collection.
* TASK-3 : Perform a text search for books containing the word "guide".
* TASK-4 : Perform a text search for books containing the phrase "modern success".
* TASK-5 : Exclude books containing the word "ancient" from the search results.
* TASK-6 : Sort books by relevance score using $meta.
* TASK-7 : Create a compound text index on the title and author fields in the books collection.
* TASK-8 : Perform a text search across both title and author fields for the term "John".
* TASK-9 : Use $text to find authors whose bio contains the word "database".
* TASK-10 : Combine text search with other query conditions (e.g., filter by genre).
* Challenge Task : Use $text and $meta to find the most relevant book for the search term "MongoDB" and display its title and score.

**EXERCISE 9: Geospatial Queries and Indexing**

**Aim:**

Introduce MongoDB's geospatial querying capabilities and indexing for location-based data. These skills are essential for applications like finding nearby restaurants, tracking delivery routes, or analyzing geographic trends.

**Key Concepts:**

* Geospatial Index : Enables efficient querying of location-based data using 2dsphere indexes.
* $near : Finds documents closest to a specified point.
* $geoWithin : Finds documents within a specified area (e.g., a polygon or circle).
* $geoIntersects : Finds documents that intersect with a specified geometry.

**Database Schema:**

* Database Name: location\_db
* places{\_id: ObjectId, name: String, location: {type: "Point", coordinates: [longitude, latitude]}}
* users{\_id: ObjectId, name: String, last\_location: {type: "Point", coordinates: [longitude, latitude]}}

**Procedure with Code:**

* TASK-1 : Insert sample records into the collections.

db.**places**.insertMany([

{name: "Central Park", location: {type: "Point", coordinates: [-73.9712, 40.7831]}},

{name: "Empire State Building", location: {type: "Point", coordinates: [-73.9857, 40.7484]}},

{name: "Statue of Liberty", location: {type: "Point", coordinates: [-74.0445, 40.6892]}}

]);

db.**users**.insertMany([

{name: "Alice", last\_location: {type: "Point", coordinates: [-73.9924, 40.7500]}},

{name: "Bob", last\_location: {type: "Point", coordinates: [-73.9750, 40.7600]}}

]);

* **TASK-2 : Create a 2dsphere index on the location field in the places collection.**

db.places.createIndex({location: "2dsphere"});

* **TASK-3 : Find places near the coordinates [-73.9924, 40.7500] (Alice's last location).**

db.places.find({

location: {

$near: {

$geometry: {

type: "Point",

coordinates: [-73.9924, 40.7500]

}

}

}

}).pretty();

* **TASK-4 : Find places within a 2-kilometer radius of Alice's last location.**

db.places.find({

location: {

$geoWithin: {

$centerSphere: [[-73.9924, 40.7500], 2 / 6378.1] // Radius in radians (2 km)

}

}

}).pretty();

* **TASK-5 : Create a 2dsphere index on the last\_location field in the users collection.**

db.users.createIndex({last\_location: "2dsphere"});

* **TASK-6 : Find users whose last location is within a bounding box around Central Park.**

db.users.find({

last\_location: {

$geoWithin: {

$box: [

[-73.9812, 40.7681], // Bottom-left corner

[-73.9612, 40.8031] // Top-right corner

]

}

}

}).pretty();

* **TASK-7 : Use $geoIntersects to find places that intersect with a polygon around Times** Square.

db.places.find({

location: {

$geoIntersects: {

$geometry: {

type: "Polygon",

coordinates: [[

[-73.9900, 40.7500],

[-73.9800, 40.7500],

[-73.9800, 40.7600],

[-73.9900, 40.7600],

[-73.9900, 40.7500]

]]

}

}

}

}).pretty();

* **TASK-8 : Combine $near with additional query conditions (e.g., filter by name).**

db.places.find({

location: {

$near: {

$geometry: {

type: "Point",

coordinates: [-73.9924, 40.7500]

}

}

},

name: "Central Park"

}).pretty();

* **TASK-9 : Drop the 2dsphere index on the location field in the places collection.**

db.places.dropIndex("location\_2dsphere");

* **TASK-10 : Use $geoWithin to find places within a circular area around Bob's last location.**

db.places.find({

location: {

$geoWithin: {

$centerSphere: [[-73.9750, 40.7600], 3 / 6378.1] // Radius in radians (3 km)

}

}

}).pretty();

* **Challenge Task : Use $near to find the nearest place to Bob's last location and display its name and distance.**

db.places.aggregate([

{

$geoNear: {

near: {type: "Point", coordinates: [-73.9750, 40.7600]},

distanceField: "distance",

spherical: true

}

},

{$sort: {distance: 1}},

{$limit: 1},

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

]).pretty();

**Result :**

Successfully performed geospatial queries using $near, $geoWithin, and $geoIntersects, enabling location-based analyses such as finding nearby places, points within a radius, and intersections with geographic shapes.

**EXERCISE-9: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Create a 2dsphere index on the location field in the places collection.
* TASK-3 : Find places near the coordinates [-73.9924, 40.7500] (Alice's last location).
* TASK-4 : Find places within a 2-kilometer radius of Alice's last location.
* TASK-5 : Create a 2dsphere index on the last\_location field in the users collection.
* TASK-6 : Find users whose last location is within a bounding box around Central Park.
* TASK-7 : Use $geoIntersects to find places that intersect with a polygon around Times Square.
* TASK-8 : Combine $near with additional query conditions (e.g., filter by name).
* TASK-9 : Drop the 2dsphere index on the location field in the places collection.
* TASK-10 : Use $geoWithin to find places within a circular area around Bob's last location.
* Challenge Task : Use $near to find the nearest place to Bob's last location and display its name and distance.

This structure ensures progressive learning while reinforcing MongoDB's geospatial querying and indexing concepts

**EXERCISE 10: Backup, Restore, and Performance Optimization**

**Aim:**

Teach students how to back up and restore MongoDB databases using mongodump and mongorestore, as well as optimize query performance using indexes and explain plans.

**Key Concepts:**

* mongodump : A utility for creating backups of MongoDB databases.
* mongorestore : A utility for restoring data from backups.
* Indexes : Improve query performance by reducing the number of documents scanned.
* explain() : Analyze query performance and identify bottlenecks.
* Performance Optimization : Techniques like indexing, query refinement, and limiting result sets.

**Database Schema:**

* Database Name: ecommerce\_db
* products{\_id: ObjectId, name: String, category: String, price: Number, stock: Number, rating: Number}
* customers{\_id: ObjectId, name: String, email: String, total\_spent: Number, loyalty\_points: Number}
* orders{\_id: ObjectId, customer\_id: ObjectId, product\_ids: [ObjectId], total\_amount: Number, order\_date: Date}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections.**

db.products.insertMany([

{name: "Laptop", category: "Electronics", price: 1200, stock: 10, rating: 4.8},

{name: "Smartphone", category: "Electronics", price: 800, stock: 25, rating: 4.6}

]);

db.customers.insertMany([

{name: "Alice Johnson", email: "alice@example.com", total\_spent: 1500, loyalty\_points: 150},

{name: "Bob Smith", email: "bob@example.com", total\_spent: 500, loyalty\_points: 50}

]);

db.orders.insertMany([

{customer\_id: db.customers.findOne({name: "Alice Johnson"}).\_id, product\_ids: [db.products.findOne({name: "Laptop"}).\_id], total\_amount: 1200, order\_date: new Date("2023-09-01")},

{customer\_id: db.customers.findOne({name: "Bob Smith"}).\_id, product\_ids: [db.products.findOne({name: "Smartphone"}).\_id], total\_amount: 800, order\_date: new Date("2023-09-02")}

]);

* **TASK-2 : Use mongodump to create a backup of the ecommerce\_db database.**

mongodump --db ecommerce\_db --out /path/to/backup/

* **TASK-3 : Drop the ecommerce\_db database to simulate data loss.**

use ecommerce\_db;

db.dropDatabase();

* **TASK-4 : Use mongorestore to restore the ecommerce\_db database from the backup.**

mongorestore --db ecommerce\_db /path/to/backup/ecommerce\_db/

* **TASK-5 : Create an index on the price field in the products collection.**

db.products.createIndex({price: 1});

* **TASK-6 : Use explain() to analyze the performance of a query on the price field.**

db.products.find({price: {$gt: 500}}).explain("executionStats");

* **TASK-7 : Optimize a query by adding a compound index on category and price.**

db.products.createIndex({category: 1, price: -1});

* **TASK-8 : Compare query performance before and after adding the compound index.**

// Before adding the compound index

db.products.find({category: "Electronics", price: {$gt: 500}}).explain("executionStats");

// After adding the compound index

db.products.find({category: "Electronics", price: {$gt: 500}}).explain("executionStats");

* **TASK-9 : Use hint() to force MongoDB to use a specific index for a query.**

db.products.find({price: {$gt: 500}}).hint({price: 1}).explain("executionStats");

* **TASK-10 : Remove unused indexes to improve write performance.**

db.products.dropIndex("price\_1");

* **Challenge Task : Use mongostat or mongotop to monitor database performance and identify slow queries.**

mongostat --host <hostname> --port <port>

mongotop --host <hostname> --port <port>

**Result :**

Successfully backed up and restored a MongoDB database using mongodump and mongorestore, optimized query performance with indexes, and analyzed query execution plans to identify bottlenecks.

**EXERCISE-10: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Use mongodump to create a backup of the ecommerce\_db database.
* TASK-3 : Drop the ecommerce\_db database to simulate data loss.
* TASK-4 : Use mongorestore to restore the ecommerce\_db database from the backup.
* TASK-5 : Create an index on the price field in the products collection.
* TASK-6 : Use explain() to analyze the performance of a query on the price field.
* TASK-7 : Optimize a query by adding a compound index on category and price.
* TASK-8 : Compare query performance before and after adding the compound index.
* TASK-9 : Use hint() to force MongoDB to use a specific index for a query.
* TASK-10 : Remove unused indexes to improve write performance.
* Challenge Task : Use mongostat or mongotop to monitor database performance and identify slow queries.

**EXERCISE 11: Domain-Specific Task – Education Database**

**Aim:**

Teach students how to design and query a MongoDB database tailored to an education system. This exercise focuses on managing student records, courses, and grades, which are essential for applications like school management systems or learning platforms.

**Key Concepts:**

* Schema Design : Structuring collections to represent real-world entities like students, courses, and enrollments.
* CRUD Operations : Inserting, querying, updating, and deleting data related to students and courses.
* Aggregation : Analyzing student performance, course enrollment statistics, and grade distributions.
* Indexes : Optimizing queries for frequently accessed fields like student IDs and course names.

**Database Schema:**

* Database Name: education\_db
* students{\_id: ObjectId, name: String, email: String, age: Number, major: String}
* courses{\_id: ObjectId, course\_name: String, instructor: String, credits: Number}
* enrollments{\_id: ObjectId, student\_id: ObjectId, course\_id: ObjectId, grade: String, enrollment\_date: Date}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections.**

// Insert records into students

db.students.insertMany([

{name: "Alice Johnson", email: "alice@example.com", age: 20, major: "Computer Science"},

{name: "Bob Smith", email: "bob@example.com", age: 22, major: "Mathematics"}

]);

// Insert records into courses

db.courses.insertMany([

{course\_name: "Introduction to Programming", instructor: "Dr. Jane Doe", credits: 3},

{course\_name: "Calculus I", instructor: "Dr. John Smith", credits: 4}

]);

// Insert records into enrollments

db.enrollments.insertMany([

{student\_id: db.students.findOne({name: "Alice Johnson"}).\_id, course\_id: db.courses.findOne({course\_name: "Introduction to Programming"}).\_id, grade: "A", enrollment\_date: new Date("2023-09-01")},

{student\_id: db.students.findOne({name: "Bob Smith"}).\_id, course\_id: db.courses.findOne({course\_name: "Calculus I"}).\_id, grade: "B", enrollment\_date: new Date("2023-09-02")}

]);

* **TASK-2 : Find all students majoring in "Computer Science".**

db.students.find({major: "Computer Science"}).pretty();

* **TASK-3 : List all courses taught by "Dr. Jane Doe".**

db.courses.find({instructor: "Dr. Jane Doe"}).pretty();

* **TASK-4 : Find the enrollment details of "Alice Johnson".**

const aliceId = db.students.findOne({name: "Alice Johnson"}).\_id;

db.enrollments.find({student\_id: aliceId}).pretty();

* **TASK-5 : Update the grade of "Bob Smith" in "Calculus I" to "A".**

const bobId = db.students.findOne({name: "Bob Smith"}).\_id;

const calculusId = db.courses.findOne({course\_name: "Calculus I"}).\_id;

db.enrollments.updateOne(

{student\_id: bobId, course\_id: calculusId},

{$set: {grade: "A"}}

);

* **TASK-6 : Calculate the total number of credits each student is enrolled in.**

db.enrollments.aggregate([

{

$lookup: {

from: "courses",

localField: "course\_id",

foreignField: "\_id",

as: "course\_details"

}

},

{

$group: {

\_id: "$student\_id",

total\_credits: {$sum: {$arrayElemAt: ["$course\_details.credits", 0]}}

}

}

]).pretty();

* **TASK-7 : Find the average grade of students in "Introduction to Programming".**

const programmingId = db.courses.findOne({course\_name: "Introduction to Programming"}).\_id;

db.enrollments.aggregate([

{$match: {course\_id: programmingId}},

{$group: {\_id: null, avg\_grade: {$avg: "$grade"}}}

]).pretty();

* **TASK-8 : Delete all enrollments older than September 1, 2023.**

db.enrollments.deleteMany({enrollment\_date: {$lt: new Date("2023-09-01")}});

* **TASK-9 : Use $lookup to join enrollments with students and display student names along with their grades.**

db.enrollments.aggregate([

{

$lookup: {

from: "students",

localField: "student\_id",

foreignField: "\_id",

as: "student\_details"

}

},

{$project: {\_id: 0, student\_name: {$arrayElemAt: ["$student\_details.name", 0]}, grade: 1}}]).pretty();

* **TASK-10 : Create an index on the course\_name field in the courses collection.**

db.courses.createIndex({course\_name: 1});

* **Challenge Task : Use aggregation to find the most popular course (the course with the highest number of enrollments).**

db.enrollments.aggregate([

{$group: {\_id: "$course\_id", enrollment\_count: {$sum: 1}}},

{$sort: {enrollment\_count: -1}},

{$limit: 1},

{

$lookup: {

from: "courses",

localField: "\_id",

foreignField: "\_id",

as: "course\_details"

}

},

{$project: {\_id: 0, course\_name: {$arrayElemAt: ["$course\_details.course\_name", 0]}, enrollment\_count: 1}}

]).pretty();

**Result :**

Successfully designed and queried an education database, managing student records, course enrollments, and grades, enabling tasks such as calculating total credits, analyzing performance, and updating grades.

**EXERCISE-11: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Find all students majoring in "Computer Science".
* TASK-3 : List all courses taught by "Dr. Jane Doe".
* TASK-4 : Find the enrollment details of "Alice Johnson".
* TASK-5 : Update the grade of "Bob Smith" in "Calculus I" to "A".
* TASK-6 : Calculate the total number of credits each student is enrolled in.
* TASK-7 : Find the average grade of students in "Introduction to Programming".
* TASK-8 : Delete all enrollments older than September 1, 2023.
* TASK-9 : Use $lookup to join enrollments with students and display student names along with their grades.
* TASK-10 : Create an index on the course\_name field in the courses collection.
* Challenge Task : Use aggregation to find the most popular course (the course with the highest number of enrollments).

**EXERCISE 12: Domain-Specific Task – Office Management System**

**Aim:**

Teach students how to design and query a MongoDB database tailored to an office management system. This exercise focuses on managing employees, departments, and tasks, which are essential for applications like workforce management or task tracking systems.

**Key Concepts:**

* Schema Design : Structuring collections to represent real-world entities like employees, departments, and tasks.
* CRUD Operations : Inserting, querying, updating, and deleting data related to employees and tasks.
* Aggregation : Analyzing employee performance, department statistics, and task completion rates.
* Indexes : Optimizing queries for frequently accessed fields like employee IDs and task statuses.

**Database Schema:**

* Database Name: office\_db
* employees{\_id: ObjectId, name: String, email: String, department: String, role: String, hire\_date: Date}
* departments{\_id: ObjectId, department\_name: String, manager\_id: ObjectId}
* tasks{\_id: ObjectId, task\_name: String, assigned\_to: ObjectId, status: String, due\_date: Date}

**Procedure with Code:**

* **TASK-1 : Insert sample records into the collections.**

// Insert records into employees

db.employees.insertMany([

{name: "Alice Johnson", email: "alice@example.com", department: "HR", role: "Manager", hire\_date: new Date("2020-06-15")},

{name: "Bob Smith", email: "bob@example.com", department: "IT", role: "Developer", hire\_date: new Date("2021-08-20")}

]);

// Insert records into departments

db.departments.insertMany([

{department\_name: "HR", manager\_id: db.employees.findOne({name: "Alice Johnson"}).\_id},

{department\_name: "IT", manager\_id: db.employees.findOne({name: "Bob Smith"}).\_id}

]);

// Insert records into tasks

db.tasks.insertMany([

{task\_name: "Onboard New Employees", assigned\_to: db.employees.findOne({name: "Alice Johnson"}).\_id, status: "Completed", due\_date: new Date("2023-09-10")},

{task\_name: "Develop New Feature", assigned\_to: db.employees.findOne({name: "Bob Smith"}).\_id, status: "In Progress", due\_date: new Date("2023-09-25")}

]);

* **TASK-2 : Find all employees in the "IT" department.**

db.employees.find({department: "IT"}).pretty();

* **TASK-3 : List all tasks assigned to "Bob Smith".**

const bobId = db.employees.findOne({name: "Bob Smith"}).\_id;

db.tasks.find({assigned\_to: bobId}).pretty();

* **TASK-4 : Update the status of the task "Develop New Feature" to "Completed".**

db.tasks.updateOne(

{task\_name: "Develop New Feature"},

{$set: {status: "Completed"}}

);

* **TASK-5 : Calculate the total number of tasks assigned to each employee.**

db.tasks.aggregate([

{$group: {\_id: "$assigned\_to", total\_tasks: {$sum: 1}}}

]).pretty();

* **TASK-6 : Find all overdue tasks (tasks with a due\_date earlier than today).**

db.tasks.find({due\_date: {$lt: new Date()}}).pretty();

* **TASK-7 : Use $lookup to join tasks with employees and display employee names along with their tasks.**

db.tasks.aggregate([

{

$lookup: {

from: "employees",

localField: "assigned\_to",

foreignField: "\_id",

as: "employee\_details"

}

},

{$project: {\_id: 0, task\_name: 1, employee\_name: {$arrayElemAt: ["$employee\_details.name", 0]}, status: 1}}

]).pretty();

* **TASK-8 : Find the department managed by "Alice Johnson".**

const aliceId = db.employees.findOne({name: "Alice Johnson"}).\_id;

db.departments.find({manager\_id: aliceId}).pretty();

* **TASK-9 : Delete all completed tasks older than September 1, 2023.**

db.tasks.deleteMany({status: "Completed", due\_date: {$lt: new Date("2023-09-01")}});

* **TASK-10 : Create an index on the status field in the tasks collection.**

db.tasks.createIndex({status: 1});

**Result :**

Successfully managed an office database, handling employee records, task assignments, and departmental relationships, enabling tasks such as tracking task statuses, calculating workloads, and managing overdue tasks.

**EXERCISE-12: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Find all employees in the "IT" department.
* TASK-3 : List all tasks assigned to "Bob Smith".
* TASK-4 : Update the status of the task "Develop New Feature" to "Completed".
* TASK-5 : Calculate the total number of tasks assigned to each employee.
* TASK-6 : Find all overdue tasks (tasks with a due\_date earlier than today).
* TASK-7 : Use $lookup to join tasks with employees and display employee names along with their tasks.
* TASK-8 : Find the department managed by "Alice Johnson".
* TASK-9 : Delete all completed tasks older than September 1, 2023.
* TASK-10 : Create an index on the status field in the tasks collection.
* Challenge Task : Use aggregation to find the employee with the most tasks assigned.

**EXERCISE 13: Domain-Specific Task – E-commerce Platform**

**Aim:**

Teach students how to design and query a MongoDB database tailored to an e-commerce platform. This exercise focuses on managing products, customers, orders, and reviews, which are essential for applications like online shopping systems.

**Key Concepts:**

* Schema Design : Structuring collections to represent real-world entities like products, customers, orders, and reviews.
* CRUD Operations : Inserting, querying, updating, and deleting data related to products and orders.
* Aggregation : Analyzing sales data, customer behavior, and product performance.
* Indexes : Optimizing queries for frequently accessed fields like product IDs, order dates, and ratings.

**Database Schema:**

* Database Name: ecommerce\_db
* products{\_id: ObjectId, name: String, category: String, price: Number, stock: Number, rating: Number}
* customers{\_id: ObjectId, name: String, email: String, total\_spent: Number, loyalty\_points: Number}
* orders{\_id: ObjectId, customer\_id: ObjectId, product\_ids: [ObjectId], total\_amount: Number, order\_date: Date}
* reviews{\_id: ObjectId, product\_id: ObjectId, customer\_id: ObjectId, rating: Number, review\_text: String, review\_date: Date}

**Procedure with Code**

* **TASK-1 : Insert sample records into the collections.**

// Insert records into products

db.**products**.insertMany([

{name: "Laptop", category: "Electronics", price: 1200, stock: 10, rating: 4.8},

{name: "Smartphone", category: "Electronics", price: 800, stock: 25, rating: 4.6}

]);

// Insert records into customers

db.**customers**.insertMany([

{name: "Alice Johnson", email: "alice@example.com", total\_spent: 1500, loyalty\_points: 150},

{name: "Bob Smith", email: "bob@example.com", total\_spent: 500, loyalty\_points: 50}

]);

// Insert records into orders

db.**orders**.insertMany([

{customer\_id: db.customers.findOne({name: "Alice Johnson"}).\_id, product\_ids: [db.products.findOne({name: "Laptop"}).\_id], total\_amount: 1200, order\_date: new Date("2023-09-01")},

{customer\_id: db.customers.findOne({name: "Bob Smith"}).\_id, product\_ids: [db.products.findOne({name: "Smartphone"}).\_id], total\_amount: 800, order\_date: new Date("2023-09-02")}

]);

// Insert records into reviews

db.**reviews**.insertMany([

{product\_id: db.products.findOne({name: "Laptop"}).\_id, customer\_id: db.customers.findOne({name: "Alice Johnson"}).\_id, rating: 5, review\_text: "Great product!", review\_date: new Date("2023-09-03")},

{product\_id: db.products.findOne({name: "Smartphone"}).\_id, customer\_id: db.customers.findOne({name: "Bob Smith"}).\_id, rating: 4, review\_text: "Good phone, but battery life could be better.", review\_date: new Date("2023-09-04")}

]);

* **TASK-2 : Find all products in the "Electronics" category.**

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

* **TASK-3 : List all orders placed by "Alice Johnson".**

const aliceId = db.customers.findOne({name: "Alice Johnson"}).\_id;

db.orders.find({customer\_id: aliceId}).pretty();

* **TASK-4 : Update the stock of the product "Laptop" to 5.**

db.products.updateOne(

{name: "Laptop"},

{$set: {stock: 5}}

);

* **TASK-5 : Calculate the total revenue generated from all orders.**

db.orders.aggregate([

{$group: {\_id: null, total\_revenue: {$sum: "$total\_amount"}}}

]).pretty();

* **TASK-6 : Find all reviews with a rating greater than 4.**

db.reviews.find({rating: {$gt: 4}}).pretty();

* **TASK-7 : Use $lookup to join orders with customers and display customer names along with** their orders.

db.orders.aggregate([

{

$lookup: {

from: "customers",

localField: "customer\_id",

foreignField: "\_id",

as: "customer\_details"

}

},

{$project: {\_id: 0, customer\_name: {$arrayElemAt: ["$customer\_details.name", 0]}, total\_amount: 1, order\_date: 1}}

]).pretty();

* **TASK-8 : Find the average rating of the product "Smartphone".**

const smartphoneId = db.products.findOne({name: "Smartphone"}).\_id;

db.reviews.aggregate([

{$match: {product\_id: smartphoneId}},

{$group: {\_id: null, avg\_rating: {$avg: "$rating"}}}

]).pretty();

* **TASK-9 : Delete all orders older than September 1, 2023.**

db.orders.deleteMany({order\_date: {$lt: new Date("2023-09-01")}});

* **TASK-10 : Create an index on the review\_date field in the reviews collection.**

db.reviews.createIndex({review\_date: 1});

* **Challenge Task : Use aggregation to find the top-rated product (the product with the highest average rating).**

db.reviews.aggregate([

{$group: {\_id: "$product\_id", avg\_rating: {$avg: "$rating"}}},

{$sort: {avg\_rating: -1}},

{$limit: 1},

{

$lookup: {

from: "products",

localField: "\_id",

foreignField: "\_id",

as: "product\_details"

}

},

{$project: {\_id: 0, product\_name: {$arrayElemAt: ["$product\_details.name", 0]}, avg\_rating: 1}}

]).pretty();

Result :

Successfully designed and queried an e-commerce database, managing products, customers, orders, and reviews, enabling tasks such as filtering products, analyzing sales data, and optimizing inventory management.

**EXERCISE-13: Student Task Section**

**Tasks:**

* TASK-1 : Insert sample records into the collections.
* TASK-2 : Find all products in the "Electronics" category.
* TASK-3 : List all orders placed by "Alice Johnson".
* TASK-4 : Update the stock of the product "Laptop" to 5.
* TASK-5 : Calculate the total revenue generated from all orders.
* TASK-6 : Find all reviews with a rating greater than 4.
* TASK-7 : Use $lookup to join orders with customers and display customer names along with their orders.
* TASK-8 : Find the average rating of the product "Smartphone".
* TASK-9 : Delete all orders older than September 1, 2023.
* TASK-10 : Create an index on the review\_date field in the reviews collection.
* Challenge Task : Use aggregation to find the top-rated product (the product with the highest average rating).

# Exercise 14: CRUD Operations in Neo4j

**Aim**  
 To perform CRUD (Create, Read, Update, Delete) operations in Neo4j using Cypher queries.

## Database Schema

* **Database Name:** college\_db
* **Collections:**
  + students { student\_id: Number(PK), name: String, major: String, year: String }
  + courses { course\_id: Number(PK), title: String, credits: Number }
  + enrollments { student\_id: Number(FK), course\_id: Number(FK), grade: String }

## Procedure with Code

* **Task 1: Create students and courses nodes.**

CREATE (s1:Student {student\_id: 1, name: "Alice", major: "Computer Science", year: "Sophomore"}),

(s2:Student {student\_id: 2, name: "Bob", major: "Mathematics", year: "Junior"}),

(c1:Course {course\_id: 101, title: "Database Systems", credits: 3}),

(c2:Course {course\_id: 102, title: "Data Structures", credits: 4});

* **Task 2: Create relationships between students and courses.**

MATCH (s:Student {student\_id: 1}), (c:Course {course\_id: 101})

CREATE (s)-[:ENROLLED\_IN {grade: "A"}]->(c);

MATCH (s:Student {student\_id: 2}), (c:Course {course\_id: 102})

CREATE (s)-[:ENROLLED\_IN {grade: "B"}]->(c);

* **Task 3: Read all students and their majors.**

MATCH (s:Student)

RETURN s.name, s.major;

* **Task 4: Read all courses and their credit hours.**

MATCH (c:Course)

RETURN c.title, c.credits;

* **Task 5: Update a student's major.**

MATCH (s:Student {student\_id: 1})

SET s.major = "Software Engineering";

* **Task 6: Delete a course node.**

MATCH (c:Course {course\_id: 102})

DELETE c;

* **Task 7: Find all courses a specific student is enrolled in.**

MATCH (s:Student {student\_id: 1})-[:ENROLLED\_IN]->(c:Course)

RETURN c.title;

* **Task 8: Count the number of students in each major.**

MATCH (s:Student)

RETURN s.major, COUNT(s) AS student\_count

ORDER BY student\_count DESC;

* **Task 9: Retrieve all students enrolled in a specific course.**

MATCH (c:Course {course\_id: 101})<-[:ENROLLED\_IN]-(s:Student)

RETURN s.name;

* **Task 10: List all unique majors in the student collection.**

MATCH (s:Student)

RETURN DISTINCT s.major;

* **Task 11: Find the average grade of students in a specific course.**

MATCH (c:Course {course\_id: 101})<-[:ENROLLED\_IN]-(s:Student)

RETURN AVG(s.grade) AS average\_grade;

* **Task 12: Find students who are not enrolled in any course.**

MATCH (s:Student)

WHERE NOT (s)-[:ENROLLED\_IN]->()

RETURN s.name;

* **Task 13: Retrieve students along with their grades in each course.**

MATCH (s:Student)-[r:ENROLLED\_IN]->(c:Course)

RETURN s.name, c.title, r.grade;

* **Task 14: Get all students and their respective courses.**

MATCH (s:Student)-[:ENROLLED\_IN]->(c:Course)

RETURN s.name, COLLECT(c.title) AS courses;

* **Task 15: Create an index on the student\_id property for performance.**

CREATE INDEX ON :Student(student\_id);

**Result**  
 Successfully performed CRUD operations in Neo4j, created relationships, and utilized aggregation functions.

**EXERCISE-14: Student Task Section**

## Database Schema

* **Database Name:** college\_db
* **Collections:**
  + students { student\_id: Number(PK), name: String, major: String, year: String }
  + courses { course\_id: Number(PK), title: String, credits: Number }
  + enrollments { student\_id: Number(FK), course\_id: Number(FK), grade: String }

## Tasks

* 1. Create a professors collection with professor\_id, name, and department.
  2. Link professors to courses they teach using a TEACHES relationship.
  3. Query to find all courses taught by a specific professor.
  4. Find students who are in their final year.
  5. Update the credit hours for a specific course.
  6. Delete a specific student from the database.
  7. List all students with their corresponding courses using a specific pattern.
  8. Create a relationship to indicate which student is a mentor for another.
  9. Count how many students are in each year.
  10. Retrieve students enrolled in more than one course.
  11. Find courses that have a specific number of credits.
  12. Identify the most popular major based on student enrollment.
  13. Retrieve all courses along with their professor.
  14. List students by their grade in a specific course.
  15. Create a unique index on the professor\_id for the professors collection.

# Exercise 15: Graph Data Modeling in Neo4j

**Aim**  
 To model a simple graph for a social network using Neo4j.

## Database Schema

* **Database Name:** social\_network\_db
* **Collections:**
  + users { user\_id: Number(PK), name: String, age: Number, city: String }
  + posts { post\_id: Number(PK), content: String, created\_at: Date }
  + friends { user\_id1: Number(FK), user\_id2: Number(FK) }
  + likes { user\_id: Number(FK), post\_id: Number(FK) }

## Procedure with Code

* **Task 1: Create users and posts nodes.**

CREATE (u1:User {user\_id: 1, name: "Alice", age: 25, city: "Chennai"}),

(u2:User {user\_id: 2, name: "Bob", age: 30, city: "Mumbai"}),

(p1:Post {post\_id: 1, content: "Hello World!", created\_at: date()}),

(p2:Post {post\_id: 2, content: "Learning Neo4j!", created\_at: date()});

* **Task 2: Create friendships between users.**

MATCH (u1:User {user\_id: 1}), (u2:User {user\_id: 2})

CREATE (u1)-[:FRIENDS\_WITH]->(u2);

* **Task 3: User creates a post.**

MATCH (u:User {user\_id: 1}), (p:Post {post\_id: 1})

CREATE (u)-[:CREATED]->(p);

* **Task 4: Find all friends of a specific user.**

MATCH (u:User {user\_id: 1})-[:FRIENDS\_WITH]->(friends)

RETURN friends.name;

* **Task 5: Count the number of posts created by each user.**

MATCH (u:User)-[:CREATED]->(p:Post)

RETURN u.name, COUNT(p) AS post\_count

ORDER BY post\_count DESC;

* **Task 6: Like a post by a user.**

MATCH (u:User {user\_id: 1}), (p:Post {post\_id: 1})

CREATE (u)-[:LIKES]->(p);

* **Task 7: Find all posts liked by a specific user.**

MATCH (u:User {user\_id: 1})-[:LIKES]->(p:Post)

RETURN p.content;

* **Task 8: Retrieve users living in a specific city.**

MATCH (u:User {city: "Chennai"})

RETURN u.name;

* **Task 9: List all users along with their number of friends.**

MATCH (u:User)

RETURN u.name, SIZE((u)-[:FRIENDS\_WITH]->()) AS friend\_count;

* **Task 10: Find posts created by users from a specific city.**

MATCH (u:User {city: "Chennai"})-[:CREATED]->(p:Post)

RETURN p.content;

* **Task 11: Create a relationship indicating users following each other.**

MATCH (u1:User {user\_id: 1}), (u2:User {user\_id: 2})

CREATE (u1)-[:FOLLOWS]->(u2);

* **Task 12: Count how many users a specific user follows.**

MATCH (u:User {user\_id: 1})-[:FOLLOWS]->(followed)

RETURN COUNT(followed) AS follows\_count;

* **Task 13: Find mutual friends between two users.**

MATCH (u1:User {user\_id: 1})-[:FRIENDS\_WITH]-(mutual:User)<-[:FRIENDS\_WITH]-(u2:User {user\_id: 2})

RETURN mutual.name;

* **Task 14: List all posts and their corresponding likers.**

MATCH (p:Post)<-[:LIKES]-(u:User)

RETURN p.content, COLLECT(u.name) AS likers;

* **Task 15: Create an index on user\_id property for faster lookup.**

CREATE INDEX ON :User(user\_id);

**Result**  
 Successfully performed CRUD operations, established relationships, and retrieved information from a social network graph.

**EXERCISE-14: Student Task Section**

## Database Schema

* **Database Name:** social\_network\_db
* **Collections:**
  + users { user\_id: Number(PK), name: String, age: Number, city: String }
  + posts { post\_id: Number(PK), content: String, created\_at: Date }
  + friends { user\_id1: Number(FK), user\_id2: Number(FK) }
  + likes { user\_id: Number(FK), post\_id: Number(FK) }

## Tasks

* 1. Create a comments collection with comment\_id, content, and created\_at.
  2. Link comments to posts and users using a COMMENTED\_ON relationship.
  3. Query to find all comments made on a specific post.
  4. Identify users above a certain age.
  5. Update the content of a specific post.
  6. Delete a specific user from the database.
  7. Find users who have not made any posts.
  8. Count the number of likes for each post.
  9. Retrieve posts by a specific user and their corresponding likes.
  10. List users who have mutual friends with a given user.
  11. Create a relationship to indicate which user recommended another user.
  12. Retrieve all users along with their posts.
  13. Find the most active user based on the number of posts.
  14. Identify users who have the same age.
  15. Create a unique index on the post\_id for the posts collection.