**LAB MANUAL**

**AY: 2023 – 24 (EVEN SEMESTER)**

**SEMESTER: 4**

**DEPT(s): CSE-ALLIED BRANCHES**

**SUBJECT NAME: Mongo DB**

**SUBJECT CODE: BDS456B**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| MongoDB | | | Semester | 4 |
| Course Code | | BDS456B | CIE Marks | 50 |
| Teaching Hours/Week (L: T:P: S) | | 0:0:2:0 | SEE Marks | 50 |
| Total Hours of Pedagogy | | 24 | Total Marks | 100 |
| Credits | | 01 | | |
| **Course objectives:**   * Understand basic MongoDB functions, operators and types of operations in MongoDB. * Demonstrate the use of Indexing, Advanced Indexing in MongoDB. * Apply the aggregation and Map Reduction in MongoDB. * Demonstrate text searching on collections in MongoDB. | | | | |
| **Sl.NO** | **Experiments** | | | |
| 1 | 1. Illustration of Where Clause, AND,OR operations in MongoDB. 2. Execute the Commands of MongoDB and operations in MongoDB : Insert, Query, Update, Delete and Projection. (Note: use any collection)   [Refer: Book 1 chapter 4]. | | | |
| 2 | 1. Develop a MongoDB query to select certain fields and ignore some fields of the documents from any collection. 2. Develop a MongoDB query to display the first 5 documents from the results obtained in a.   [use of limit and find]  [Refe: Book1 Chapter 4, book 2: chapter 5] | | | |
| 3 | 1. Execute query selectors (comparison selectors, logical selectors ) and list out the results on any collection 2. Execute query selectors (Geospatial selectors, Bitwise selectors ) and list out the results on any collection   [Refer: Book 3 Chapter 13] | | | |
| 4 | Create and demonstrate how projection operators ($, $elematch and $slice) would be used in the MondoDB.  [Refer: Book 3 Chapter 14] | | | |
| 5 | Execute Aggregation operations ($avg, $min,$max, $push, $addToSet etc.). students encourage to execute several queries to demonstrate various aggregation operators)  [Refer: Book 3 Chapter 15] | | | |
| 6 | Execute Aggregation Pipeline and its operations (pipeline must contain $match, $group, $sort, $project, $skip etc. students encourage to execute several queries to demonstrate various aggregation operators)  [refer book 2: chapter 6 ] | | | |
| 7 | 1. Find all listings with listing\_url, name, address, host\_picture\_url in the listings And Reviews collection that have a host with a picture url 2. Using E-commerce collection write a query to display reviews summary.   [refer Book2: chapter 6] | | | |
| 8 | 1. Demonstrate creation of different types of indexes on collection (unique, sparse, compound and multikey indexes) 2. Demonstrate optimization of queries using indexes.   Refer: Book 2: Chapter 8 and Book 3: Chapter 12] | | | |
| 9 | 1. Develop a query to demonstrate Text search using catalog data collection for a given word 2. Develop queries to illustrate excluding documents with certain words and phrases   Refer: Book 2: Chapter 9] | | | |
| 10 | Develop an aggregation pipeline to illustrate Text search on Catalog data collection.  Refer: Book 2 :Chapter 9] | | | |
| **Course outcomes (Course Skill Set):**  At the end of the course the student will be able to:   1. Make use of MangoDB commands and queries. 2. Illustrate the role of aggregate pipelines to extract data. 3. Demonstrate optimization of queries by creating indexes. 4. Develop aggregate pipelines for text search in collections. | | | | |
| **Assessment Details (both CIE and SEE)**  The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together  **Continuous Internal Evaluation (CIE):**  CIE marks for the practical course are **50 Marks**.  The split-up of CIE marks for record/ journal and test are in the ratio **60:40**.   * Each experiment is to be evaluated for conduction with an observation sheet and record write-up. Rubrics for the evaluation of the journal/write-up for hardware/software experiments are designed by the faculty who is handling the laboratory session and are made known to students at the beginning of the practical session. * Record should contain all the specified experiments in the syllabus and each experiment write-up will be evaluated for 10 marks. * Total marks scored by the students are scaled down to **30 marks** (60% of maximum marks). * Weightage to be given for neatness and submission of record/write-up on time. * Department shall conduct a test of 100 marks after the completion of all the experiments listed in the syllabus. * In a test, test write-up, conduction of experiment, acceptable result, and procedural knowledge will carry a weightage of 60% and the rest 40% for viva-voce. * The suitable rubrics can be designed to evaluate each student’s performance and learning ability. * The marks scored shall be scaled down to **20 marks** (40% of the maximum marks).   The Sum of scaled-down marks scored in the report write-up/journal and marks of a test is the total CIE marks scored by the student. | | | | |
| **Semester End Evaluation (SEE):**   * SEE marks for the practical course are 50 Marks. * SEE shall be conducted jointly by the two examiners of the same institute, examiners are appointed by the Head of the Institute. * The examination schedule and names of examiners are informed to the university before the conduction of the examination. These practical examinations are to be conducted between the schedule mentioned in the academic calendar of the University. * All laboratory experiments are to be included for practical examination. * (Rubrics) Breakup of marks and the instructions printed on the cover page of the answer script to be strictly adhered to by the examiners. **OR** based on the course requirement evaluation rubrics shall be decided jointly by examiners. * Students can pick one question (experiment) from the questions lot prepared by the examiners jointly. * Evaluation of test write-up/ conduction procedure and result/viva will be conducted jointly by examiners. * General rubrics suggested for SEE are mentioned here, writeup-20%, Conduction procedure and result in -60%, Viva-voce 20% of maximum marks. SEE for practical shall be evaluated for 100 marks and scored marks shall be scaled down to 50 marks (however, based on course type, rubrics shall be decided by the examiners)   Change of experiment is allowed only once and 15% of Marks allotted to the procedure part are to be made zero.  The minimum duration of SEE is 02 hours | | | | |
| **Suggested Learning Resources:**   * **BOOK 1: “**MongoDB: The Definitive Guide”, Kristina chodorow, 2nd ed O’REILLY, 2013. * **BOOK 2: “***MongoDB in Action*” by KYLE BANKER et. al. 2nd ed, Manning publication, 2016 * **BOOK 3:**  “MongoDB Complete Guide” by Manu Sharma 1st ed, bpb publication, 2023. * **installation of MongoDB Video:** <https://www.youtube.com/watch?v=dEm2AS5amyA> * **video on Aggregation:** <https://www.youtube.com/watch?v=vx1C8EyTa7Y> * **MongoDB in action book Code download URL:** [**h**ttps://www.manning.com/downloads/529](https://www.manning.com/downloads/529) * **MongoDB Exercise URL:** <https://www.w3resource.com/mongodb-exercises/> | | | | |

## Experiment 1:

## a. Illustration of Where Clause, AND,OR operations in MongoDB.

## b. Execute the Commands of MongoDB and operations in MongoDB : Insert, Query, Update, Delete and Projection. (Note: use any collection)

## a. Where Clause, AND,OR operations in MongoDB

In MongoDB, the equivalent of SQL’s **WHERE** clause is achieved using query filters within the **find()** method. You can also combine multiple conditions using logical operators like **$and** and **$or**. Here’s how you can illustrate the usage of these features:

### Setting Up Example Data

First, let’s assume we have a collection named **ProgrammingBooks** with the following documents:

**ProgBooksDB>** use newDB

switched to db newDB

**newDB>**db.getName()

newDB

**newDB>**db.createCollection("ProgrammingBooks")

{ ok: 1 }

**newDB>**db.ProgrammingBooks.insertMany([

{ title: "Clean Code",

author: "Robert C. Martin",

category: "Software Development", year: 2008 },

{ title: "JavaScript: The Good Parts",

author: "Douglas Crockford",

category: "JavaScript", year: 2008 },

{ title: "Design Patterns",

author: "Erich Gamma",

category: "Software Design", year: 1994 },

{ title: "Introduction to Algorithms",

author: "Thomas H. Cormen",

category: "Algorithms", year: 2009 },

{ title: "Python Crash Course",

author: "Eric Matthes",

category: "Python", year: 2015 }]);

### Using the WHERE Clause Equivalent

To query documents with specific conditions, you can use the **find()** method with a filter object. For example, to find books published in the year 2008:

pretty() method is used to configure the cursor to display results in an easy-to-read format.

**newDB>** db.ProgrammingBooks.find({ year: 2008 }).pretty()

**Output:**

[

{

\_id: ObjectId('6651daad9edbdf91e12202e2'),

title: 'Clean Code',

author: 'Robert C. Martin',

category: 'Software Development',

year: 2008

},

{

\_id: ObjectId('6651daad9edbdf91e12202e3'),

title: 'JavaScript: The Good Parts',

author: 'Douglas Crockford',

category: 'JavaScript',

year: 2008

}

]

### Using the $and Operator

The **$and** operator is used to combine multiple conditions that must all be true. Here’s how to find books that are in the “Software Development” category and published in the year 2008:

**newDB>**db.ProgrammingBooks.find({ $and: [

{ category: "Software Development" },

{ year: 2008 }]}).pretty()

**Output:**

[

{

\_id: ObjectId('6651daad9edbdf91e12202e2'),

title: 'Clean Code',

author: 'Robert C. Martin',

category: 'Software Development',

year: 2008

}

]

* In this query both conditions must be met for a document to be included in the result.

### Using the $or Operator

The **$or** operator is used to combine multiple conditions where at least one must be true. Here’s how to find books that are either in the “JavaScript” category or published in the year 2015:

**newDB>** db.ProgrammingBooks.find({

$or: [

{ category: "JavaScript" },

{ year: 2015 }

]

}).pretty()

**Output:**

[

{

\_id: ObjectId('6651daad9edbdf91e12202e3'),

title: 'JavaScript: The Good Parts',

author: 'Douglas Crockford',

category: 'JavaScript',

year: 2008

},

{

\_id: ObjectId('6651daad9edbdf91e12202e6'),

title: 'Python Crash Course',

author: 'Eric Matthes',

category: 'Python',

year: 2015

}

]

In this query a document will be included in the result if it meets either condition.

### Combining $and and $or Operators

You can combine **$and** and **$or** operators for more complex queries. For example, to find books that are either in the “Software Development” category and published after 2007, or in the “Python” category:

**newDB>** db.ProgrammingBooks.find({

$or: [

{

$and: [

{ category: "Software Development" },

{ year: { $gt: 2007 } }

]

},

{ category: "Python" }

]

}).pretty()

**Output:**

[

{

\_id: ObjectId('6651daad9edbdf91e12202e2'),

title: 'Clean Code',

author: 'Robert C. Martin',

category: 'Software Development',

year: 2008

},

{

\_id: ObjectId('6651daad9edbdf91e12202e6'),

title: 'Python Crash Course',

author: 'Eric Matthes',

category: 'Python',

year: 2015

}

]

1. Execute the Commands of MongoDB and operations in MongoDB : Insert, Query, Update, Delete and Projection. (Note: use any collection)

## MongoDB Operations

### ****Switch to a Database****

If you want to use a specific database, switch to that database using the **use** command. If the database doesn’t exist, MongoDB will create it implicitly when you insert data into it:

**test>** use ProgBooksDB

switched to db ProgBooksDB

**ProgBooksDB>**

### ****Create the****ProgrammingBooks****Collection:****

To create the **ProgrammingBooks** collection, use the **createCollection()** method. This step is optional because MongoDB will automatically create the collection when you insert data into it, but you can explicitly create it if needed:

**ProgBooksDB>** db.createCollection("ProgrammingBooks")

### Insert operations

##### **Insert a Single Document into**ProgrammingBooks**:**

Use the **insertOne()** method to insert a new document into the **ProgrammingBooks** collection:

**ProgBooksDB>** db.ProgrammingBooks.insertOne({

title: "The Pragmatic Programmer: Your Journey to Mastery",

author: "David Thomas, Andrew Hunt",

category: "Software Development",

year: 1999

})

##### **Insert multiple Documents into the**ProgrammingBooks**Collection :**

Now, insert 5 documents representing programming books into the **ProgrammingBooks** collection using the **insertMany()** method:

ProgBooksDB> db.ProgrammingBooks.insertMany([

{

title: "Clean Code: A Handbook of Agile Software Craftsmanship",

author: "Robert C. Martin",

category: "Software Development",

year: 2008

},

{

title: "JavaScript: The Good Parts",

author: "Douglas Crockford",

category: "JavaScript",

year: 2008

},

{

title: "Design Patterns: Elements of Reusable Object-Oriented Software",

author: "Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides",

category: "Software Design",

year: 1994

},

{

title: "Introduction to Algorithms",

author: "Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein",

category: "Algorithms",

year: 1990

},

{

title: "Python Crash Course: A Hands-On, Project-Based Introduction to Programming",

author: "Eric Matthes",

category: "Python",

year: 2015

}

])

### Query operations

##### **Find All Documents**

To retrieve all documents from the **ProgrammingBooks** collection:

**ProgBooksDB>** db.ProgrammingBooks.find().pretty()

##### **Find Documents Matching a Condition**

To find books published after the year 2000:

**ProgBooksDB>** db.ProgrammingBooks.find({ year: { $gt: 2000 } }).pretty()

### Update Operations

##### **a. Update a Single Document**

To update a specific book (e.g., change the author of a book):

**ProgBooksDB>**db.ProgrammingBooks.updateOne(

{ title: "Clean Code: A Handbook of Agile Software Craftsmanship" },

{ $set: { author: "Robert C. Martin (Uncle Bob)" } }

)

//verify by displaying books published in year 2008

**ProgBooksDB>** db.ProgrammingBooks.find({ year: { $eq: 2008 } }).pretty()

##### **b. Update Multiple Documents**

To update multiple books (e.g., update the category of books published before 2010):

**ProgBooksDB>** db.ProgrammingBooks.updateMany(

{ year: { $lt: 2010 } },

{ $set: { category: "Classic Programming Books" } }

)

//verify the update operation by displaying books published before year 2010

**ProgBooksDB>** db.ProgrammingBooks.find({ year: { $lt: 2010 } }).pretty()

### Delete Operations

##### **Delete a Single Document**

To delete a specific book from the collection (e.g., delete a book by title):

ProgBooksDB> db.ProgrammingBooks.deleteOne({ title: "JavaScript: The Good Parts" })

{ acknowledged: true, deletedCount: 1 }

//Verify to see document is deleted

ProgBooksDB> db.ProgrammingBooks.find({ title: "JavaScript: The Good Parts" }).pretty()

##### **Delete Multiple Documents**

To delete multiple books based on a condition (e.g., delete all books published before 1995):

ProgBooksDB> db.ProgrammingBooks.deleteMany({ year: { $lt: 1995 } })

{ acknowledged: true, deletedCount: 2 }

##### **Delete All Documents in the Collection:**

To delete all documents in a collection (e.g., **ProgrammingBooks**), use the **deleteMany()** method with an empty filter ‘**{}’**:

**ProgBooksDB>** db.ProgrammingBooks.deleteMany({})

{ acknowledged: true, deletedCount: 3 }

//verify by displaying the collection

**ProgBooksDB>** db.ProgrammingBooks.find().pretty()

### Projection Operations

In MongoDB, a projection refers to the mechanism of specifying which fields (or columns) should be returned from a query result. When querying a collection, you can use projection to control the shape of the returned documents by specifying which fields to include or exclude.

In MongoDB, projection is typically specified as the second parameter to the **find()** method. The projection parameter takes an object where keys represent the fields to include or exclude, with values of **1** (include) or **0** (exclude).

**Include Specific Fields:**

Use **1** to include a field in the result:

**ProgBooksDB>** db.ProgrammingBooks.find({}, { title: 1, author: 1 } )

**Exclude Specific Fields:**

Use **0** to exclude a field from the result:

**ProgBooksDB>** db.ProgrammingBooks.find({}, {year: 0})

## Experiment 2:

1. Develop a MongoDB query to select certain fields and ignore some fields of the documents from any collection.
2. Develop a MongoDB query to display the first 5 documents from the results obtained in a collection.

[use of limit and find]

## a. Select and ignore fields

**Develop a MongoDB query to select certain fields and ignore some fields of the documents from any collection.**

To select certain fields and ignore others in MongoDB, you use projections in your queries. Projections allow you to specify which fields to include or exclude in the returned documents.

### Create database and create the Collection:

**test>** use MoviesDB

switched to db MoviesDB

**MoviesDB>** db.createCollection("Movies")

{ ok: 1 }

**MoviesDB>** db.Movies.insertMany([

{ title: "Inception", director: "Christopher Nolan",

genre: "Science Fiction", year: 2010, ratings:

{ imdb: 8.8, rottenTomatoes: 87 } },

{ title: "The Matrix", director: "Wachowskis",

genre: "Science Fiction", year: 1999, ratings:

{ imdb: 8.7, rottenTomatoes: 87 } },

{ title: "The Godfather", director: "Francis Ford Coppola",

genre: "Crime", year: 1972, ratings: { imdb: 9.2, rottenTomatoes: 97 } }]);

**Output:**

{

acknowledged: true,

insertedIds: {

'0': ObjectId('66523751d5449c3abf2202d8'),

'1': ObjectId('66523751d5449c3abf2202d9'),

'2': ObjectId('66523751d5449c3abf2202da')

}

}

### Basic Syntax for Projection

When using the **find()** method, the first parameter is the query filter, and the second parameter is the projection object. The projection object specifies the fields to include (using **1**) or exclude (using **0**).

### Including Specific Fields

To include specific fields, set the fields you want to include to **1**:

To select only the **title** and **director** fields from the **Movies** collection:

**MoviesDB>** db.Movies.find({}, { title: 1, director: 1 })

**Output:**

[

{

\_id: ObjectId('66523751d5449c3abf2202d8'),

title: 'Inception',

director: 'Christopher Nolan'

},

{

\_id: ObjectId('66523751d5449c3abf2202d9'),

title: 'The Matrix',

director: 'Wachowskis'

},

{

\_id: ObjectId('66523751d5449c3abf2202da'),

title: 'The Godfather',

director: 'Francis Ford Coppola'

}

]

**MoviesDB>** db.Movies.find({}, { title: 1, director: 1, \_id: 0 })

**Output:**

[

{ title: 'Inception', director: 'Christopher Nolan' },

{ title: 'The Matrix', director: 'Wachowskis' },

{ title: 'The Godfather', director: 'Francis Ford Coppola' }

]

In this query:

* The filter **{}** means we want to select all documents.
* The projection **{ title: 1, director: 1, \_id: 0 }** means we include the **title** and **director** fields, and exclude the **\_id** field (which is included by default unless explicitly excluded).

### Excluding Specific Fields

To exclude specific fields, set the fields you want to exclude to **0**:

To exclude the **ratings** field from the results:

**MoviesDB>** db.Movies.find({}, { ratings: 0 })

**Output:**

[

{

\_id: ObjectId('66523751d5449c3abf2202d8'),

title: 'Inception',

director: 'Christopher Nolan',

genre: 'Science Fiction',

year: 2010

},

{

\_id: ObjectId('66523751d5449c3abf2202d9'),

title: 'The Matrix',

director: 'Wachowskis',

genre: 'Science Fiction',

year: 1999

},

{

\_id: ObjectId('66523751d5449c3abf2202da'),

title: 'The Godfather',

director: 'Francis Ford Coppola',

genre: 'Crime',

year: 1972

}

]

In this query:

* The filter **{}** means we want to select all documents.
* The projection **{ ratings: 0 }** means we exclude the **ratings** field.

**Combining Filter and Projection**

* You can also combine a query filter with a projection. For example, to find movies directed by “Christopher Nolan” and include only the **title** and **year** fields:

**MoviesDB>** db.Movies.find({ director: "Christopher Nolan" }, { title: 1, year: 1, \_id: 0 })

Output:

Output:

[ { title: 'Inception', year: 2010 } ]

In this query:

* The filter **{ director: "Christopher Nolan" }** selects documents where the **director** is “Christopher Nolan”.
* The projection **{ title: 1, year: 1, \_id: 0 }** includes only the **title** and **year** fields and excludes the **\_id** field.

In MongoDB, projections are used to control which fields are included or excluded in the returned documents. This is useful for optimizing queries and reducing the amount of data transferred over the network. You specify projections as the second parameter in the **find()** method.

## b. Use of limit and find in MongoDB query

**Develop a MongoDB query to display the first 5 documents from the results obtained in a. (illustrate use of limit and find)**

To display the first 5 documents from a query result in MongoDB, you can use the **limit()** method in conjunction with the **find()** method. The **limit()** method restricts the number of documents returned by the query to the specified number.

### Example Scenario

Assume we have the **Movies** collection as described previously:

**test>** use MoviesDB

switched to db MoviesDB

**MoviesDB>** db.createCollection("Movies")

{ ok: 1 }

**MoviesDB>**db.Movies.insertMany([

{ title: "Inception", director: "Christopher Nolan", genre: "Science Fiction", year: 2010, ratings: { imdb: 8.8, rottenTomatoes: 87 } },

{ title: "The Matrix", director: "Wachowskis", genre: "Science Fiction", year: 1999, ratings: { imdb: 8.7, rottenTomatoes: 87 } },

{ title: "The Godfather", director: "Francis Ford Coppola", genre: "Crime", year: 1972, ratings: { imdb: 9.2, rottenTomatoes: 97 } },

{ title: "Pulp Fiction", director: "Quentin Tarantino", genre: "Crime", year: 1994, ratings: { imdb: 8.9, rottenTomatoes: 92 } },

{ title: "The Shawshank Redemption", director: "Frank Darabont", genre: "Drama", year: 1994, ratings: { imdb: 9.3, rottenTomatoes: 91 } },

{ title: "The Dark Knight", director: "Christopher Nolan", genre: "Action", year: 2008, ratings: { imdb: 9.0, rottenTomatoes: 94 } },

{ title: "Fight Club", director: "David Fincher", genre: "Drama", year: 1999, ratings: { imdb: 8.8, rottenTomatoes: 79 } }]);

### Query with Projection and Limit

Suppose you want to display the first 5 documents from the **Movies** collection, including only the **title**, **director**, and **year** fields. Here’s how you can do it:

**MoviesDB>** db.Movies.find({}, { title: 1, director: 1, year: 1, \_id: 0 }).limit(5)

**Output:**

[

{ "title": "Inception", "director": "Christopher Nolan", "year": 2010 },

{ "title": "The Matrix", "director": "Wachowskis", "year": 1999 },

{ "title": "The Godfather", "director": "Francis Ford Coppola", "year": 1972 },

{ "title": "Pulp Fiction", "director": "Quentin Tarantino", "year": 1994 },

{ "title": "The Shawshank Redemption", "director": "Frank Darabont", "year": 1994 }

]

### Explanation:

* **find({})**: This filter **{}** selects all documents in the collection.
* **{ title: 1, director: 1, year: 1, \_id: 0 }**: This projection includes the **title**, **director**, and **year** fields, and excludes the **\_id** field.
* **.limit(5)**: This method limits the query result to the first 5 documents.

By using the **find()** method with a projection and the **limit()** method, you can efficiently query and display a subset of documents from a MongoDB collection. This approach helps manage large datasets by retrieving only a specific number of documents, which is particularly useful for paginating results in applications.

## Experiment 3:

1. Execute query selectors (comparison selectors, logical selectors ) and list out the results on any collection.
2. Execute query selectors (Geospatial selectors, Bitwise selectors ) and list out the results on any collection

## a. Query selectors (comparison selectors, logical selectors )

**Execute query selectors (comparison selectors, logical selectors ) and list out the results on any collection**

Let’s create a new collection called **Employees** and insert some documents into it. Then, we’ll demonstrate the use of comparison selectors and logical selectors to query this collection.

### Create the Employees Collection and Insert Documents

First, we need to create the **Employees** collection and insert some sample documents.

**test>** use companyDB

**companyDB>** db.Employees.insertMany([

{ name: "Alice", age: 30, department: "HR", salary: 50000, joinDate: new Date("2015-01-15") },

{ name: "Bob", age: 24, department: "Engineering", salary: 70000, joinDate: new Date("2019-03-10") },

{ name: "Charlie", age: 29, department: "Engineering", salary: 75000, joinDate: new Date("2017-06-23") },

{ name: "David", age: 35, department: "Marketing", salary: 60000, joinDate: new Date("2014-11-01") },

{ name: "Eve", age: 28, department: "Finance", salary: 80000, joinDate: new Date("2018-08-19") }

])

**Output:**

{

acknowledged: true,

insertedIds: {

'0': ObjectId('665356cff5b334bcf92202d8'),

'1': ObjectId('665356cff5b334bcf92202d9'),

'2': ObjectId('665356cff5b334bcf92202da'),

'3': ObjectId('665356cff5b334bcf92202db'),

'4': ObjectId('665356cff5b334bcf92202dc')

}

}

### Queries Using Comparison Selectors

#### 1. $eq (Equal)

Find employees in the “Engineering” department.

**companyDB>** db.Employees.find({ department: { $eq: "Engineering" } }).pretty()

**Output:**

[

{

\_id: ObjectId('665356cff5b334bcf92202d9'),

name: 'Bob',

age: 24,

department: 'Engineering',

salary: 70000,

joinDate: ISODate('2019-03-10T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202da'),

name: 'Charlie',

age: 29,

department: 'Engineering',

salary: 75000,

joinDate: ISODate('2017-06-23T00:00:00.000Z')

}

]

#### 2. $ne (Not Equal)

Find employees who are not in the “HR” department.

**companyDB>** db.Employees.find({ department: { $ne: "HR" } }).pretty()

**Output:**

[

{

\_id: ObjectId('665356cff5b334bcf92202d9'),

name: 'Bob',

age: 24,

department: 'Engineering',

salary: 70000,

joinDate: ISODate('2019-03-10T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202da'),

name: 'Charlie',

age: 29,

department: 'Engineering',

salary: 75000,

joinDate: ISODate('2017-06-23T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202db'),

name: 'David',

age: 35,

department: 'Marketing',

salary: 60000,

joinDate: ISODate('2014-11-01T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202dc'),

name: 'Eve',

age: 28,

department: 'Finance',

salary: 80000,

joinDate: ISODate('2018-08-19T00:00:00.000Z')

}

]

#### 3. $gt (Greater Than)

Find employees who are older than 30.

**companyDB>** db.Employees.find({ age: { $gt: 30 } }).pretty()

**Output:**

[

{

\_id: ObjectId('665356cff5b334bcf92202db'),

name: 'David',

age: 35,

department: 'Marketing',

salary: 60000,

joinDate: ISODate('2014-11-01T00:00:00.000Z')

}

]

#### 4. $lt (Less Than)

Find employees with a salary less than 70000.

**companyDB>** db.Employees.find({ salary: { $lt: 70000 } }).pretty()

**Output:**

[

{

\_id: ObjectId('665356cff5b334bcf92202d8'),

name: 'Alice',

age: 30,

department: 'HR',

salary: 50000,

joinDate: ISODate('2015-01-15T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202db'),

name: 'David',

age: 35,

department: 'Marketing',

salary: 60000,

joinDate: ISODate('2014-11-01T00:00:00.000Z')

}

]

#### 5. $gte (Greater Than or Equal)

Find employees who joined on or after January 1, 2018.

**companyDB>** db.Employees.find({ joinDate: { $gte: new Date("2018-01-01") } }).pretty()

**Output:**

[

{

\_id: ObjectId('665356cff5b334bcf92202d9'),

name: 'Bob',

age: 24,

department: 'Engineering',

salary: 70000,

joinDate: ISODate('2019-03-10T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202dc'),

name: 'Eve',

age: 28,

department: 'Finance',

salary: 80000,

joinDate: ISODate('2018-08-19T00:00:00.000Z')

}

]

#### 6. $lte (Less Than or Equal)

Find employees who are 28 years old or younger.

**companyDB>** db.Employees.find({ age: { $lte: 28 } }).pretty()

**Output:**

[

{

\_id: ObjectId('665356cff5b334bcf92202d9'),

name: 'Bob',

age: 24,

department: 'Engineering',

salary: 70000,

joinDate: ISODate('2019-03-10T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202dc'),

name: 'Eve',

age: 28,

department: 'Finance',

salary: 80000,

joinDate: ISODate('2018-08-19T00:00:00.000Z')

}

]

### Queries Using Logical Selectors

#### 1. $and (Logical AND)

Find employees who are in the “Engineering” department and have a salary greater than 70000.

**companyDB>** db.Employees.find({

$and: [

{ department: "Engineering" },

{ salary: { $gt: 70000 } }

]

}).pretty()

**Output:**

[

{

\_id: ObjectId('665356cff5b334bcf92202da'),

name: 'Charlie',

age: 29,

department: 'Engineering',

salary: 75000,

joinDate: ISODate('2017-06-23T00:00:00.000Z')

}

]

#### 2. $or (Logical OR)

Find employees who are either in the “HR” department or have a salary less than 60000.

**companyDB>** db.Employees.find({

$or: [

{ department: "HR" },

{ salary: { $lt: 60000 } }

]

}).pretty()

**Output:**

[

{

\_id: ObjectId('665356cff5b334bcf92202d8'),

name: 'Alice',

age: 30,

department: 'HR',

salary: 50000,

joinDate: ISODate('2015-01-15T00:00:00.000Z')

}

]

#### 3. $not (Logical NOT)

Find employees who are not in the “Engineering” department.

**companyDB>** db.Employees.find({

department: {

$not: { $eq: "Engineering" }

}

}).pretty()

**Output:**

[

{

\_id: ObjectId('665356cff5b334bcf92202d8'),

name: 'Alice',

age: 30,

department: 'HR',

salary: 50000,

joinDate: ISODate('2015-01-15T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202db'),

name: 'David',

age: 35,

department: 'Marketing',

salary: 60000,

joinDate: ISODate('2014-11-01T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202dc'),

name: 'Eve',

age: 28,

department: 'Finance',

salary: 80000,

joinDate: ISODate('2018-08-19T00:00:00.000Z')

}

]

#### 4. $nor (Logical NOR)

Find employees who are neither in the “HR” department nor have a salary greater than 75000.

**companyDB>** db.Employees.find({

$nor: [

{ department: "HR" },

{ salary: { $gt: 75000 } }

]

}).pretty()

**Output:**

[

{

\_id: ObjectId('665356cff5b334bcf92202d9'),

name: 'Bob',

age: 24,

department: 'Engineering',

salary: 70000,

joinDate: ISODate('2019-03-10T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202da'),

name: 'Charlie',

age: 29,

department: 'Engineering',

salary: 75000,

joinDate: ISODate('2017-06-23T00:00:00.000Z')

},

{

\_id: ObjectId('665356cff5b334bcf92202db'),

name: 'David',

age: 35,

department: 'Marketing',

salary: 60000,

joinDate: ISODate('2014-11-01T00:00:00.000Z')

}

]

## b. Query selectors (Geospatial selectors, Bitwise selectors )

**Execute query selectors (Geospatial selectors, Bitwise selectors ) and list out the results on any collection**

Let’s extend our MongoDB examples to include queries using geospatial selectors and bitwise selectors. We will create a new collection called **Places** for geospatial queries and a collection called Devices for bitwise queries.

**test>**use geoDatabase

**geoDatabase>**db.Places.insertMany([

{ name: "Central Park", location: { type: "Point", coordinates: [-73.9654, 40.7829] } },

{ name: "Times Square", location: { type: "Point", coordinates: [-73.9851, 40.7580] } },

{ name: "Brooklyn Bridge", location: { type: "Point", coordinates: [-73.9969, 40.7061] } },

{ name: "Empire State Building", location: { type: "Point", coordinates: [-73.9857, 40.7488] } },

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

])

##### 1. $near (Find places near a certain point)

Find places near a specific coordinate, for example, near Times Square.

**geoDatabase>**db.Places.find({

location: {

$near: {

$geometry: {

type: "Point",

coordinates: [-73.9851, 40.7580]

},

$maxDistance: 5000 // distance in meters

}

}

}).pretty()

**Output:**

[

{

\_id: ObjectId('66536a9799cad9cd2b2202d9'),

name: 'Times Square',

location: { type: 'Point', coordinates: [ -73.9851, 40.758 ] }

},

{

\_id: ObjectId('66536a9799cad9cd2b2202db'),

name: 'Empire State Building',

location: { type: 'Point', coordinates: [ -73.9857, 40.7488 ] }

},

{

\_id: ObjectId('66536a9799cad9cd2b2202d8'),

name: 'Central Park',

location: { type: 'Point', coordinates: [ -73.9654, 40.7829 ] }

}

]

##### 2. ‘$geoWithin’ (Find places within a specific area)

Find places within a specific polygon, for example, an area covering part of Manhattan.

**geoDatabase>**db.Places.find({

location: {

$geoWithin: {

$geometry: {

type: "Polygon",

coordinates: [

[

[-70.016, 35.715],

[-74.014, 40.717],

[-73.990, 40.730],

[-73.990, 40.715],

[-70.016, 35.715]

]

]

}

}

}

}).pretty()

**Output:**

[

{

\_id: ObjectId('66536a9799cad9cd2b2202da'),

name: 'Brooklyn Bridge',

location: { type: 'Point', coordinates: [ -73.9969, 40.7061 ] }

}

]

### Bitwise Selectors

Next, let’s create a **Devices** collection for bitwise operations.

#### Create the Devices Collection and Insert Documents

**test>**use techDB

**techDB>**db.Devices.insertMany([

{ name: "Device A", status: 5 }, // Binary: 0101

{ name: "Device B", status: 3 }, // Binary: 0011

{ name: "Device C", status: 12 }, // Binary: 1100

{ name: "Device D", status: 10 }, // Binary: 1010

{ name: "Device E", status: 7 } // Binary: 0111

])

1. ‘$bitsAllSet’ (Find documents where all bits are set)

Find devices where the binary status has both the 1st and 3rd bits set (binary mask 0101, or decimal 5).

**techDB>**db.Devices.find({

status: { $bitsAllSet: [0, 2] }

}).pretty()

**Output:**

[

{

\_id: ObjectId('6653703d4e38f292e52202d8'),

name: 'Device A',

status: 5

},

{

\_id: ObjectId('6653703d4e38f292e52202dc'),

name: 'Device E',

status: 7

}

]

2. $bitsAnySet (Find documents where any of the bits are set)

Find devices where the binary status has at least the 2nd bit set (binary mask 0010, or decimal 2).

**techDB>**db.Devices.find({

status: { $bitsAnySet: [1] }

}).pretty()

**Output:**

[

{

\_id: ObjectId('6653703d4e38f292e52202d9'),

name: 'Device B',

status: 3

},

{

\_id: ObjectId('6653703d4e38f292e52202db'),

name: 'Device D',

status: 10

},

{

\_id: ObjectId('6653703d4e38f292e52202dc'),

name: 'Device E',

status: 7

}

]

3. ‘$bitsAllClear’ (Find documents where all bits are clear)

Find devices where the binary status has both the 2nd and 4th bits clear (binary mask 1010, or decimal 10).

**techDB>**b.Devices.find({

status: { $bitsAllClear: [1, 3] }

}).pretty()

**Output:**

[

{

\_id: ObjectId('6653703d4e38f292e52202d8'),

name: 'Device A',

status: 5

}

]

4. $bitsAnyClear (Find documents where any of the bits are clear)

Find devices where the binary status has at least the 1st bit clear (binary mask 0001, or decimal 1).

**techDB>**db.Devices.find({

status: { $bitsAnyClear: [0] }

}).pretty()

**Output:**

[

{

\_id: ObjectId('6653703d4e38f292e52202da'),

name: 'Device C',

status: 12

},

{

\_id: ObjectId('6653703d4e38f292e52202db'),

name: 'Device D',

status: 10

}

]

**Experiment 4:**

**Projection Operators**

**Create and demonstrate how projection operators ($, $elematch and $slice) would be used in the MondoDB.**

To demonstrate the use of projection operators ($, $elemMatch, and $slice) in MongoDB, let’s create a Products collection. We’ll insert documents that include arrays, which will allow us to showcase these operators effectively.

### Create the Products Collection and Insert Documents

**test>** use retailDB

**retailDB>** db.Products.insertMany([{

name: "Laptop",

brand: "BrandA",

features: [

{ name: "Processor", value: "Intel i7" },

{ name: "RAM", value: "16GB" },

{ name: "Storage", value: "512GB SSD" }

],

reviews: [

{ user: "Alice", rating: 5, comment: "Excellent!" },

{ user: "Bob", rating: 4, comment: "Very good" },

{ user: "Charlie", rating: 3, comment: "Average" }

]

},

{

name: "Smartphone",

brand: "BrandB",

features: [

{ name: "Processor", value: "Snapdragon 888" },

{ name: "RAM", value: "8GB" },

{ name: "Storage", value: "256GB" }

],

reviews: [

{ user: "Dave", rating: 4, comment: "Good phone" },

{ user: "Eve", rating: 2, comment: "Not satisfied" }

]

}

])

### Use Projection Operators

#### 1. The $ Projection Operator

The $ operator is used to project the first matching element from an array of embedded documents.

**Example:** Find the product named “Laptop” and project the review from the user “Alice”.

**retailDB**> db.Products.find(

{ name: "Laptop", "reviews.user": "Alice" },

{ "reviews.$": 1 }

).pretty()

**Output**

{ "\_id": ObjectId("..."),

"reviews": [{ "user": "Alice", "rating": 5, "comment": "Excellent!" }]}

#### 2. The $elemMatch Projection Operator

The $elemMatch operator is used to project the first matching element from an array based on specified criteria.

**Example:** Find the product named “Laptop” and project the review where the rating is greater than 4.

**retailDB>** db.Products.find(

{ name: "Laptop" },

{ reviews: { $elemMatch: { rating: { $gt: 4 } } } }

).pretty()

**Output:**

{

"\_id": ObjectId("..."),

"reviews": [

{ "user": "Alice", "rating": 5, "comment": "Excellent!" }

]

}

#### 3. The $slice Projection Operator

The $slice operator is used to include a subset of the array field.

**Example:** Find the product named “Smartphone” and project the first review.

**retailDB>** db.Products.find(

{ name: "Smartphone" },

{ reviews: { $slice: 1 } }

).pretty()

**Output:**

{

"\_id": ObjectId("..."),

"reviews": [

{ "user": "Dave", "rating": 4, "comment": "Good phone" }

]

}

### Additional Example with Multiple Projection Operators

**Example:** Find the product named “Laptop” and project the name, the first two features, and the review with the highest rating.

**retailDB>** db.Products.find(

{ name: "Laptop" },

{

name: 1,

features: { $slice: 2 },

reviews: { $elemMatch: { rating: 5 } }

}

).pretty()

**Output:**

{

"\_id": ObjectId("..."),

"name": "Laptop",

"features": [

{ "name": "Processor", "value": "Intel i7" },

{ "name": "RAM", "value": "16GB" } ],"reviews": [ { "user": "Alice", "rating": 5, "comment": "Excellent!" }]}

Using projection operators in MongoDB, you can fine-tune the data returned by your queries:

* The **$** operator is useful for projecting the first matching element from an array.
* The **$elemMatch** operator allows you to project the first array element that matches specified criteria.
* The **$slice** operator lets you project a subset of an array, such as the first **n** elements or a specific range.

**Experiment 5:**

Execute Aggregation operations ($avg, $min,$max, $push, $addToSet etc.). students encourage to execute several queries to demonstrate various aggregation operators)

## Aggregation operations

Execute Aggregation operations (𝑎𝑣𝑔,min,𝑚𝑎𝑥,push, $addToSet etc.). students encourage to execute several queries to demonstrate various aggregation operators)

To demonstrate aggregation operations such as $avg, $min, $max, $push, and $addToSet in MongoDB, we will use a Sales collection. This collection will contain documents representing sales transactions.

### Create the Sales Collection and Insert Documents

First, we’ll create the Sales collection and insert sample documents.

test> use salesDB

salesDB> db.Sales.insertMany([

{ date: new Date("2024-01-01"), product: "Laptop", price: 1200, quantity: 1, customer: "Amar" },

{ date: new Date("2024-01-02"), product: "Laptop", price: 1200, quantity: 2, customer: "Babu" },

{ date: new Date("2024-01-03"), product: "Mouse", price: 25, quantity: 5, customer: "Chandra" },

{ date: new Date("2024-01-04"), product: "Keyboard", price: 45, quantity: 3, customer: "Amar" },

{ date: new Date("2024-01-05"), product: "Monitor", price: 300, quantity: 1, customer: "Babu" },

{ date: new Date("2024-01-06"), product: "Laptop", price: 1200, quantity: 1, customer: "Deva" }

])

### Execute Aggregation Operations

#### 1. $avg (Average)

Calculate the average price of each product.

**salesDB>** db.Sales.aggregate([

{

$group: {

\_id: "$product",

averagePrice: { $avg: "$price" }

}

}

]).pretty()

**Output:**

[

{ "\_id": "Laptop", "averagePrice": 1200 },

{ "\_id": "Mouse", "averagePrice": 25 },

{ "\_id": "Keyboard", "averagePrice": 45 },

{ "\_id": "Monitor", "averagePrice": 300 }

]

#### 2. $min (Minimum)

Find the minimum price of each product.

**salesDB>** db.Sales.aggregate([

{

$group: {

\_id: "$product",

minPrice: { $min: "$price" }

}

}

]).pretty()

**Output:**

[

{ "\_id": "Laptop", "minPrice": 1200 },

{ "\_id": "Mouse", "minPrice": 25 },

{ "\_id": "Keyboard", "minPrice": 45 },

{ "\_id": "Monitor", "minPrice": 300 }

]

#### 3. $max (Maximum)

Find the maximum price of each product.

**salesDB>** db.Sales.aggregate([

{

$group: {

\_id: "$product",

maxPrice: { $max: "$price" }

}

}

]).pretty()

**Output:**

[

{ "\_id": "Laptop", "maxPrice": 1200 },

{ "\_id": "Mouse", "maxPrice": 25 },

{ "\_id": "Keyboard", "maxPrice": 45 },

{ "\_id": "Monitor", "maxPrice": 300 }

]

#### 4. $push (Push Values to an Array)

Group sales by customer and push each purchased product into an array.

**salesDB>** db.Sales.aggregate([

{

$group: {

\_id: "$customer",

products: { $push: "$product" }

}

}

]).pretty()

**Output:**

[

{ "\_id": "Amar", "products": ["Laptop", "Keyboard"] },

{ "\_id": "Babu", "products": ["Laptop", "Monitor"] },

{ "\_id": "Chandra", "products": ["Mouse"] },

{ "\_id": "Deva", "products": ["Laptop"] }

]

#### 5. $addToSet (Add Unique Values to an Array)

Group sales by customer and add each unique purchased product to an array.

**salesDB>** db.Sales.aggregate([

{

$group: {

\_id: "$customer",

uniqueProducts: { $addToSet: "$product" }

}

}

]).pretty()

**Output:**

[

{ "\_id": "Amar", "uniqueProducts": ["Laptop", "Keyboard"] },

{ "\_id": "Babu", "uniqueProducts": ["Laptop", "Monitor"] },

{ "\_id": "Chandra", "uniqueProducts": ["Mouse"] },

{ "\_id": "Deva", "uniqueProducts": ["Laptop"] }

]

### Combining Aggregation Operations

Let’s combine several aggregation operations to get a comprehensive report.

**Example:** Calculate the total quantity and total sales amount for each product, and list all customers who purchased each product.

**salesDB>** db.Sales.aggregate([

{

$group: {

\_id: "$product",

totalQuantity: { $sum: "$quantity" },

totalSales: { $sum: { $multiply: ["$price", "$quantity"] } },

customers: { $addToSet: "$customer" }

}

}

]).pretty()

**Output:**

[

{

"\_id": "Laptop",

"totalQuantity": 4,

"totalSales": 4800,

"customers": ["Amar", "Babu", "Deva"]

},

{

"\_id": "Mouse",

"totalQuantity": 5,

"totalSales": 125,

"customers": ["Chandra"]

},

{

"\_id": "Keyboard",

"totalQuantity": 3,

"totalSales": 135,

"customers": ["Amar"]

},

{

"\_id": "Monitor",

"totalQuantity": 1,

"totalSales": 300,

"customers": ["Babu"]

}

]

By using aggregation operations such as $avg, $min, $max, $push, and $addToSet, you can perform complex data analysis and transformations on MongoDB collections. These operations enable you to calculate averages, find minimum and maximum values, push values into arrays, and create sets of unique values. The examples provided show how to use these operators to analyze a Sales collection.

**Experiment 6:**

## Aggregation Pipeline and its operations

**Execute Aggregation Pipeline and its operations (pipeline must contain 𝑚𝑎𝑡𝑐ℎ,group, 𝑠𝑜𝑟𝑡,project, $skip etc.)**

Let’s consider a scenario involving a restaurantDB database with a restaurants collection. Each document in the restaurants collection contains details about a restaurant, including its name, cuisine, location, and an array of reviews. Each review includes a rating and a comment. After creating the restaurantDB database and insert sample documents into the restaurants collection we will create an aggregation pipeline as shown below.

use restaurantDB

db.restaurants.insertMany([

{

name: "Biryani House",

cuisine: "Indian",

location: "Jayanagar",

reviews: [

{ user: "Aarav", rating: 5, comment: "Amazing biryani!" },

{ user: "Bhavana", rating: 4, comment: "Great place!" }

]

},

{

name: "Burger Joint",

cuisine: "American",

location: "Koramangala",

reviews: [

{ user: "Chirag", rating: 3, comment: "Average burger" },

{ user: "Devika", rating: 4, comment: "Good value" }

]

},

{

name: "Pasta House",

cuisine: "Italian",

location: "Rajajinagar",

reviews: [

{ user: "Esha", rating: 5, comment: "Delicious pasta!" },

{ user: "Farhan", rating: 4, comment: "Nice ambiance" }

]

},

{

name: "Curry Palace",

cuisine: "Indian",

location: "Jayanagar",

reviews: [

{ user: "Gaurav", rating: 4, comment: "Spicy and tasty!" },

{ user: "Harini", rating: 5, comment: "Best curry in town!" }

]

},

{

name: "Taco Stand",

cuisine: "Mexican",

location: "Jayanagar",

reviews: [

{ user: "Ishaan", rating: 5, comment: "Fantastic tacos!" },

{ user: "Jaya", rating: 4, comment: "Very authentic" }

]

}

])

Now, let’s execute an aggregation pipeline that includes the $match, $unwind, $group, $sort, $project, and $skip stages.

#### Aggregation Pipeline Explanation

1. **$match**: Filter restaurants by cuisine ("Jayanagar" location).
2. **$unwind**: Deconstruct the reviews array from each document to output a document for each review.
3. **$group**: Group the documents by restaurant name and calculate the average rating and total number of reviews.
4. **$sort**: Sort the results by average rating in descending order.
5. **$project**: Restructure the output to include only the restaurant name, average rating, and total reviews.
6. **$skip**: Skip the first document.

db.restaurants.aggregate([

{

$match: {

location: "Jayanagar"

}

},

{

$unwind: "$reviews"

},

{

$group: {

\_id: "$name",

averageRating: { $avg: "$reviews.rating" },

totalReviews: { $sum: 1 }

}

},

{

$sort: {

averageRating: -1

}

},

{

$project: {

\_id: 0,

restaurant: "$\_id",

averageRating: 1,

totalReviews: 1

}

},

{

$skip: 1

}

]).pretty()

**Experiment 7:**

1. Find all listings with listing\_url, name, address, host\_picture\_url in the listings And Reviews collection that have a host with a picture url
2. Using E-commerce collection write a query to display reviews summary.

## a. Find all listings

**Find all listings with listing\_url, name, address, host\_picture\_url in the listings And Reviews collection that have a host with a picture url**

To find all listings with listing\_url, name, address, and host\_picture\_url in the listingsAndReviews collection where the host has a picture URL, let is create appropriate databases and queries as follows.

### Create the Database

First, switch to or create the database you want to use. For this example, let’s call the database vacationRentals.

test> use vacationRentals

switched to db vacationRentals

vacationRentals>

### Create the listingsAndReviews Collection and Insert Documents

Next, create the listingsAndReviews collection and insert sample documents. Here are a few example documents to illustrate the structure:

vacationRentals> db.listingsAndReviews.insertMany([

{

listing\_url: "http://www.example.com/listing/123456",

name: "Beautiful Apartment",

address: {

street: "123 Main Street",

suburb: "Central",

city: "Metropolis",

country: "Wonderland"

},

host: {

name: "Alice",

picture\_url: "http://www.example.com/images/host/host123.jpg"

}

},

{

listing\_url: "http://www.example.com/listing/654321",

name: "Cozy Cottage",

address: {

street: "456 Another St",

suburb: "North",

city: "Smallville",

country: "Wonderland"

},

host: {

name: "Bob",

picture\_url: ""

}

},

{

listing\_url: "http://www.example.com/listing/789012",

name: "Modern Condo",

address: {

street: "789 Side Road",

suburb: "East",

city: "Gotham",

country: "Wonderland"

},

host: {

name: "Charlie",

picture\_url: "http://www.example.com/images/host/host789.jpg"

}

}

])

### Query to Find Listings with Host Picture URLs

Now that the collection is set up, you can run the query to find all listings with listing\_url, name, address, and host\_picture\_url where the host has a picture URL.

db.listingsAndReviews.find(

{

"host.picture\_url": { $exists: true, $ne: "" }

},

{

listing\_url: 1,

name: 1,

address: 1,

"host.picture\_url": 1

}

).pretty()

### Explanation:

* **Query Filter:**
  + "host.picture\_url": { $exists: true, $ne: "" }: This part of the query ensures that only documents where the host.picture\_url field exists and is not an empty string are selected.
* **Projection:**
  + { listing\_url: 1, name: 1, address: 1, "host.picture\_url": 1 }: This part of the query specifies the fields to include in the output. The 1 indicates that these fields should be included.

### Expected Result

The query should return documents where the host has a picture URL. Based on the inserted documents, the result should look something like this:

{

"\_id": ObjectId("..."),

"listing\_url": "http://www.example.com/listing/123456",

"name": "Beautiful Apartment",

"address": {

"street": "123 Main Street",

"suburb": "Central",

"city": "Metropolis",

"country": "Wonderland"

},

"host": {

"picture\_url": "http://www.example.com/images/host/host123.jpg"

}

}

{

"\_id": ObjectId("..."),

"listing\_url": "http://www.example.com/listing/789012",

"name": "Modern Condo",

"address": {

"street": "789 Side Road",

"suburb": "East",

"city": "Gotham",

"country": "Wonderland"

},

"host": {

"picture\_url": "http://www.example.com/images/host/host789.jpg"

}

}

## b. E-commerce collection

**Using E-commerce collection write a query to display reviews summary.**

To display a summary of reviews in an e-commerce collection, we can assume the ecommerce database contains a products collection with documents structured to include reviews. Each product document could have a reviews array with review details such as rating, comment, and user.

use ecommerce

db.products.insertMany([

{

product\_id: 1,

name: "Laptop",

category: "Electronics",

price: 1200,

reviews: [

{ user: "Alice", rating: 5, comment: "Excellent!" },

{ user: "Bob", rating: 4, comment: "Very good" },

{ user: "Charlie", rating: 3, comment: "Average" }

]

},

{

product\_id: 2,

name: "Smartphone",

category: "Electronics",

price: 800,

reviews: [

{ user: "Dave", rating: 4, comment: "Good phone" },

{ user: "Eve", rating: 2, comment: "Not satisfied" },

{ user: "Frank", rating: 5, comment: "Amazing!" }

]

},

{

product\_id: 3,

name: "Headphones",

category: "Accessories",

price: 150,

reviews: [

{ user: "Grace", rating: 5, comment: "Great sound" },

{ user: "Heidi", rating: 3, comment: "Okay" }

]

}

])

This script will set up the ecommerce database, populate the products collection with sample data, and execute an aggregation query to summarize the reviews.

### Explanation:

1. **$unwind:** Deconstructs the reviews array from each document to output a document for each element.
2. **$group:** Groups the documents by product name, and calculates:
   * totalReviews: The total number of reviews for each product.
   * averageRating: The average rating of the reviews for each product.
   * comments: An array of all review comments for each product.
3. **$project:** Restructures the output documents to include the product name, total reviews, average rating, and comments.

db.products.aggregate([

{

$unwind: "$reviews"

},

{

$group: {

\_id: "$name",

totalReviews: { $sum: 1 },

averageRating: { $avg: "$reviews.rating" },

comments: { $push: "$reviews.comment" }

}

},

{

$project: {

\_id: 0,

product: "$\_id",

totalReviews: 1,

averageRating: 1,

comments: 1

}

}

]).pretty()

### Sample Output:

The query will return a summary for each product in the collection:

[

{

"product": "Laptop",

"totalReviews": 3,

"averageRating": 4,

"comments": [

"Excellent!",

"Very good",

"Average"

]

},

{

"product": "Smartphone",

"totalReviews": 3,

"averageRating": 3.6666666666666665,

"comments": [

"Good phone",

"Not satisfied",

"Amazing!"

]

},

{

"product": "Headphones",

"totalReviews": 2,

"averageRating": 4,

"comments": [

"Great sound",

"Okay"

]

}

]

**Experiment 8:**

1. Demonstrate creation of different types of indexes on collection (unique, sparse, compound and multikey indexes)
2. Demonstrate optimization of queries using indexes.

## a. Demonstrate different types of indexes

**Demonstrate creation of different types of indexes on collection (unique, sparse, compound and multikey indexes)**

Let’s demonstrate the creation of various types of indexes on a restaurants collection in the restaurantDB  database. We’ll cover unique, sparse, compound, and multikey indexes.

### Step 1: Create the database and Collection

First, let’s set up the restaurantDB database and insert sample documents into the restaurants collection.

test> use restaurantDB

switched to db restaurantDB

restaurantDB > db.restaurants.insertMany([

{name: "Biryani House",cuisine: "Indian",location: "Downtown",

reviews: [{ user: "Aarav", rating: 5, comment: "Amazing biryani!" },

{ user: "Bhavana", rating: 4, comment: "Great place!"}],

contact: { phone: "1234567890", email: "contact@biryanihouse.com" }},

{name: "Curry Palace",cuisine: "Indian",location: "Downtown",

reviews: [{user: "Gaurav", rating: 4, comment: "Spicy and tasty!" },

{ user: "Harini", rating: 5, comment: "Best curry in town!"}],

contact: { phone: "0987654321", email: "contact@currypalace.com" }},

{name: "Taco Stand",cuisine: "Mexican",location: "Downtown",

reviews: [{ user: "Ishaan", rating: 5, comment: "Fantastic tacos!" },

{ user: "Jaya", rating: 4, comment: "Very authentic" }],

contact: { phone: "1122334455", email: "contact@tacostand.com" }}])

Output:

{

acknowledged: true,

insertedIds: {

'0': ObjectId('669f7a61d8bffa229fcdcdf6'),

'1': ObjectId('669f7a61d8bffa229fcdcdf7'),

'2': ObjectId('669f7a61d8bffa229fcdcdf8')

}

}

### Step 2: Create Various Indexes

#### 1. Unique Index

A unique index ensures that the indexed field does not contain duplicate values.

// Create a unique index on the contact.email field

db.restaurants.createIndex({ "contact.email": 1 }, { unique: true })

#### 2. Sparse Index

A sparse index only includes documents that contain the indexed field, ignoring documents where the field is missing.

// Create a sparse index on the location field

db.restaurants.createIndex({ location: 1 }, { sparse: true })

#### 3. Compound Index

A compound index indexes multiple fields within a single index.

// Create a compound index on the name and location fields

db.restaurants.createIndex({ name: 1, location: 1 })

#### 4. Multikey Index

A multikey index is created on an array field, indexing each element of the array.

// Create a multikey index on the reviews field

db.restaurants.createIndex({ reviews: 1 })

### Step 3: Verify Indexes

To verify the created indexes, you can use the getIndexes method.

// Verify the created indexes

restaurantDB> db.restaurants.getIndexes()

**Output:**

[

{ v: 2, key: { \_id: 1 }, name: '\_id\_' },

{

v: 2,

key: { 'contact.email': 1 },

name: 'contact.email\_1',

unique: true

},

{ v: 2, key: { location: 1 }, name: 'location\_1', sparse: true },

{ v: 2, key: { name: 1, location: 1 }, name: 'name\_1\_location\_1' },

{ v: 2, key: { reviews: 1 }, name: 'reviews\_1' }

]

This experiment sets up the restaurantDB database, populates the restaurants collection with sample data, and demonstrates the creation of unique, sparse, compound, and multikey indexes. The getIndexes method at the end allows you to verify the indexes created on the collection.

## b. Demonstrate optimization of queries using indexes.

To demonstrate the optimization of queries using indexes, we have to use a fairly large database where query execution times are longer. For the following examples, we’ll use a dataset of daily NASDAQ summaries. To follow along, you’ll need this data locally.

Step 1: Create the Database and Collection

First, download the archive from here . Then, unzip the file to a temporary folder.

$ unzip stocks.zip

Archive: stocks.zip

creating: dump/stocks/

inflating: dump/stocks/system.indexes.bson

inflating: dump/stocks/values.bson

Now its time to import this stocks database into the MongoDB using the mongorestore command. After that switch to the stocks database.

$mongorestore -d stocks dump/stocks

$ mongosh

test>use stocks

switched to db stocks

Lets have a look at the structure of this database to

find the various fields using the following command.

The stocks database has a value collection that contains,

for a certain subset of the NASDAQ stock exchange’s symbols,

there’s a document for each day’s high, low, close, and volume

for a 25-year period beginning in 1983.

stocks>show collections

stocks>keys = db.values.findOne()

{ \_id: ObjectId('4d094f58c96767d7a0099d49'),

exchange: 'NASDAQ', stock\_symbol: 'AACC',

date: '2008-03-07',open: 8.4,

high: 8.75,

low: 8.08,close: 8.55,

volume: 275800,

'adj close': 8.55}

stocks>db.values.countDocuments()

output: 4308303

You can also see that this database has more than four million

records/documents and a huge amount of information in it.

Queries run on such databases usually take more time to execute.

For example if we want to find out the first occurrence of

Google’s stock price, we issue the following query.

stocks>db.values.find({"stock\_symbol": "GOOG"}).sort({date: -1}).limit(1)

[

{

\_id: ObjectId('4d094f7ec96767d7a02a0af6'),

exchange: 'NASDAQ',

stock\_symbol: 'GOOG',

date: '2008-03-07',

open: 428.88,

high: 440,

low: 426.24,

close: 433.35,

volume: 8071800,

'adj close': 433.35

}

]

We observe that this query takes time to run. Lets see if we can

actually measure the time taken. Fortunately

we have a explain() method. MongoDB’s explain command

provides detailed information about a given query execution

details. It provides even more detail with the executionStats

parameter. Let us see an example

Step 2: Issue a slow query

Let us develop a query to finding the highest closing price in the data set.

stocks> db.values.find({}).sort({close: -1}).limit(1)

[

{

\_id: ObjectId('4d094f69c96767d7a01a110d'),

exchange: 'NASDAQ',

stock\_symbol: 'BORD',

date: '2000-09-25',

open: 7500,

high: 7500,

low: 7500,

close: 7500,

volume: 0,

'adj close': 6679.94

}

]

Step 3: Execution statistics

To obtain execution statistics we need to append the explain

method. To get more details pass executionStats to the explain

method.

stocks>db.values.find({}).sort({close: -1}).limit(1).explain()

stocks>db.values.find({}).sort({close: -1}).limit(1).explain("executionStats")

stocks>db.values.find({}).sort({close:1}).limit(1).explain("executionStats").executionStats.totalDocsExamined

4308303

stocks>db.values.find({}).sort({close:1}).limit(1).explain("executionStats").executionStats.executionTimeMillis

1831

Step 2: Create a Index and optimize query performance

We will add an index on the close field.

stocks> db.values.createIndex({close: 1})

close\_1

We will rerun the queries now and examine the query performance.

stocks>db.values.find({}).sort({close:1}).limit(1).explain("executionStats").executionStats.totalDocsExamined

1

stocks>db.values.find({}).sort({close:1}).limit(1).explain("executionStats").executionStats.executionTimeMillis

57

Now after the database is indexed we see a huge improvement in

performance. The no of records scanned is just 1 and query took

only 57 milliseconds to execute.

**Experiment 9:**

1. Develop a query to demonstrate Text search using catalog data collection for a given word.
2. Develop queries to illustrate excluding documents with certain words and phrases

# Solution:

1. Develop a query to demonstrate Text search using catalog data collection for a given word. To demonstrate text search in MongoDB using a catalog collection, we’ll follow these steps: Create the catalog collection and insert sample documents.

Create a text index on the relevant fields.

Perform a text search query to find documents containing a specific word.

For this example let us consider a movie database that has been imported from a CSV file. We can import data from the CSV file using the mongoimport utility as follows:

The CSV file kan\_movies.csv is provided below for your reference.

**Step 1:** Importing from CSV file using **mongoimport** into the catalog Collection mongoimport --db=kannadaMoviesDB --collection=catalog

--file=c:\users\Admin\Downloads\kan\_movies.csv --type=csv

--fields="name","year","duration","rating","genre","lang"

–db parameter is used to specify the database into which data is to be imported.

–file parameter is used to specify the file from which data is to be imported

–type parameter is used to specify the file type (csv, json,…..)

–collection parameter is used to specify the collection into which data is to be imported.

–fields parameter is used to specify a list of strings that are field names in the collection. Now launch MongoDB and choose the newly created database as follows:

test> use kannadaMoviesDB switched to db kannadaMoviesDB kannadaMoviesDB> show collections catalog

kannadaMoviesDB> db.catalog.countDocuments() 701

Alternatively you can create a catalog collection by adding documents using insertMany query as done in previous exercises. **Step 2:** Create a Text Index

Next, create a text index on the name and genre fields to enable text search.

// Create a text index on the name and genre fields db.catalog.createIndex({name: "text", genre: "text"}) **Step 3:** Perform a Text Search Query

Now, let’s perform a text search to find documents containing a specific word. For example, let’s search for the word “maga”.

// Perform a text search query to find documents containing the word "maga" db.catalog.find({$text: {$search: "maga"}})

// Perform a text search query to find documents containing the word "raju" db.catalog.find({$text: {$search: "raju"}})

**Step 4:** Perform a Text Search Query for a phrase

Now, let’s perform a text search to find documents containing a specific phrase. For example, let’s search for the phrase “tappida Maga”.

// Perform a text search query to find documents containing the phrase "maga" db.catalog.find({$text: {$search: "tappida Maga"}})

# Explanation

1. Inserting Documents: We insert several documents into the

catalog collection with fields name,year,duration,rating,genre, and lang.

1. Creating a Text Index: We create a text index on the name and description fields to enable text search.
2. Performing a Text Search: We use the $text operator with the **$search** parameter to find documents that contain the word “raju” in either the name or genre fields.

This script sets up the catalog collection, creates a text index, and demonstrates a text search query to find documents containing a specific word.

1. Develop queries to illustrate excluding documents with certain words and phrases

To exclude documents containing certain words or phrases in MongoDB, you can use the $text operator combined with the

$search parameter and the negation (-) operator.

This allows you to perform text searches that exclude specific terms. A negated term is a term that is prefixed by a minus sign -.

If you negate a term, the $text operator excludes the documents that contain those terms from the results.

# Step-by-Step Process

1. Set up the catalog collection: Insert sample documents.
2. Create a text index: Enable text search.
3. Perform queries to exclude documents: Use the $text operator with negation.

**Step 1:** Create a catalog collection

For this we will use the same catalog collection from our previous example. You can follow the same steps as earlier to create the collection.

**Step 2:** Create a Text Index

Create a text index on the name and description fields.

// Create a text index on the name and description fields db.catalog.createIndex({ name: "text", description: "text", genre: "text" }) **Step 3:** Perform Queries to Exclude Documents

Use the $text operator with negation to exclude documents containing specific words or phrases.

**Example 1:** Exclude Documents Containing the Word “action”

Suppose we want to list movies that belong to crime or romance (or both) genre but not belonging to the action genre. Since this will yield too many results we will restrict the search to the year 2021.

// Exclude documents containing the word "action"

db.catalog.find({ $text: { $search: "crime romance -action" }, year:2021 } )

**Example 2:** Exclude Documents Containing the Phrase “da maga”

We display those documents that have the word maga but not the phrase da maga.

// Exclude documents containing the phrase "da maga" db.catalog.find({$text: {$search: "maga -"da maga""}})

**Note:**The negated word excludes documents that contain the negated word from the result set. When passed a string that only contains negated words, $text does not match any documents.

**Experiment 10:**

Develop an aggregation pipeline to illustrate Text search on Catalog data collection.

This section shows aggregation pipeline examples that use the following pizza orders collection:

|  |
| --- |
| db.orders.insertMany( [ |
| { \_id: 0, name: **"Pepperoni"**, size: **"small"**, price: 19, |
| quantity: 10, date: ISODate( **"2021-03-13T08:14:30Z"** ) }, |
| { \_id: 1, name: **"Pepperoni"**, size: **"medium"**, price: 20, |
| quantity: 20, date : ISODate( **"2021-03-13T09:13:24Z"** ) }, |
| { \_id: 2, name: **"Pepperoni"**, size: **"large"**, price: 21, |
| quantity: 30, date : ISODate( **"2021-03-17T09:22:12Z"** ) }, |
| { \_id: 3, name: **"Cheese"**, size: **"small"**, price: 12, |
| quantity: 15, date : ISODate( **"2021-03-13T11:21:39.736Z"** ) }, |
| { \_id: 4, name: **"Cheese"**, size: **"medium"**, price: 13, |
| quantity:50, date : ISODate( **"2022-01-12T21:23:13.331Z"** ) }, |
| { \_id: 5, name: **"Cheese"**, size: **"large"**, price: 14, |
| quantity: 10, date : ISODate( **"2022-01-12T05:08:13Z"** ) }, |
| { \_id: 6, name: **"Vegan"**, size: **"small"**, price: 17, |
| quantity: 10, date : ISODate( **"2021-01-13T05:08:13Z"** ) }, |
| { \_id: 7, name: **"Vegan"**, size: **"medium"**, price: 18, |
| quantity: 10, date : ISODate( **"2021-01-13T05:10:13Z"** ) } |
| ] ) |

### Calculate Total Order Quantity

The following aggregation pipeline example contains two [stages](https://www.mongodb.com/docs/manual/reference/operator/aggregation-pipeline/#std-label-aggregation-pipeline-operator-reference) and returns the total order quantity of medium size pizzas grouped by pizza name:

|  |
| --- |
| db.orders.aggregate( [ |
|  |
| *// Stage 1: Filter pizza order documents by pizza size* |
| { |
| $match: { size: **"medium"** } |
| }, |
|  |
| *// Stage 2: Group remaining documents by pizza name and calculate total quantity* |
| { |
| $group: { \_id: **"$name"**, totalQuantity: { $sum: **"$quantity"** } } |
| } |
|  |
| ] ) |

The [$match](https://www.mongodb.com/docs/manual/reference/operator/aggregation/match/#mongodb-pipeline-pipe.-match) stage:

* Filters the pizza order documents to pizzas with a size of medium.
* Passes the remaining documents to the [$group](https://www.mongodb.com/docs/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) stage.

The [$group](https://www.mongodb.com/docs/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) stage:

* Groups the remaining documents by pizza name.
* Uses [$sum](https://www.mongodb.com/docs/manual/reference/operator/aggregation/sum/#mongodb-group-grp.-sum) to calculate the total order quantity for each pizza name. The total is stored in the totalQuantity field returned by the aggregation pipeline.

Example output:

|  |
| --- |
| [ |
| { \_id: **'Cheese'**, totalQuantity: 50 }, |
| { \_id: **'Vegan'**, totalQuantity: 10 }, |
| { \_id: **'Pepperoni'**, totalQuantity: 20 } |
| ] |

### Calculate Total Order Value and Average Order Quantity

* The following example calculates the total pizza order value and average order quantity between two dates:

|  |
| --- |
| db.orders.aggregate( [ |
|  |
| *// Stage 1: Filter pizza order documents by date range* |
| { |
| $match: |
| { |
| **"date"**: { $gte: new ISODate( **"2020-01-30"** ), $lt: new ISODate( **"2022-01-30"** ) } |
| } |
| }, |
|  |
| *// Stage 2: Group remaining documents by date and calculate results* |
| { |
| $group: |
| { |
| \_id: { $dateToString: { format: **"%Y-%m-%d"**, date: **"$date"** } }, |
| totalOrderValue: { $sum: { $multiply: [ **"$price"**, **"$quantity"** ] } }, |
| averageOrderQuantity: { $avg: **"$quantity"** } |
| } |
| }, |
|  |
| *// Stage 3: Sort documents by totalOrderValue in descending order* |
| { |
| $sort: { totalOrderValue: -1 } |
| } |
|  |
| ] ) |

The [$match](https://www.mongodb.com/docs/manual/reference/operator/aggregation/match/#mongodb-pipeline-pipe.-match) stage:

* Filters the pizza order documents to those in a date range specified using [$gte](https://www.mongodb.com/docs/manual/reference/operator/aggregation/gte/#mongodb-expression-exp.-gte) and [$lt.](https://www.mongodb.com/docs/manual/reference/operator/aggregation/lt/#mongodb-expression-exp.-lt)
* Passes the remaining documents to the [$group](https://www.mongodb.com/docs/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) stage.

The [$group](https://www.mongodb.com/docs/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) stage:

* Groups the documents by date using [$dateToString.](https://www.mongodb.com/docs/manual/reference/operator/aggregation/dateToString/#mongodb-expression-exp.-dateToString)
* For each group, calculates:
  + Total order value using [$sum](https://www.mongodb.com/docs/manual/reference/operator/aggregation/sum/#mongodb-group-grp.-sum) and [$multiply.](https://www.mongodb.com/docs/manual/reference/operator/aggregation/multiply/#mongodb-expression-exp.-multiply)
  + Average order quantity using [$avg.](https://www.mongodb.com/docs/manual/reference/operator/aggregation/avg/#mongodb-group-grp.-avg)
* Passes the grouped documents to the [$sort](https://www.mongodb.com/docs/manual/reference/operator/aggregation/sort/#mongodb-pipeline-pipe.-sort) stage.

The [$sort](https://www.mongodb.com/docs/manual/reference/operator/aggregation/sort/#mongodb-pipeline-pipe.-sort) stage:

* Sorts the documents by the total order value for each group in descending order (-1).
* Returns the sorted documents.

Example output:

|  |
| --- |
| [ |
| { \_id: **'2022-01-12'**, totalOrderValue: 790, averageOrderQuantity: 30 }, |
| { \_id: **'2021-03-13'**, totalOrderValue: 770, averageOrderQuantity: 15 }, |
| { \_id: **'2021-03-17'**, totalOrderValue: 630, averageOrderQuantity: 30 }, |
| { \_id: **'2021-01-13'**, totalOrderValue: 350, averageOrderQuantity: 10 } |
| ] |