1a. Illustration of Where Clause, AND,OR operations in MongoDB.

Create a MongoDB collection named students with the following documents:

[

{ "\_id": 1, "name": "Alice", "age": 20, "grade": "A", "city": "New York" },

{ "\_id": 2, "name": "Bob", "age": 22, "grade": "B", "city": "San Francisco" },

{ "\_id": 3, "name": "Charlie", "age": 23, "grade": "A", "city": "New York" },

{ "\_id": 4, "name": "David", "age": 21, "grade": "C", "city": "Los Angeles" },

{ "\_id": 5, "name": "Eve", "age": 20, "grade": "B", "city": "New York" }

]

WHERE CLAUSE

db.students.find({"$where":"this.age>25"})

**AND Operation**

The **AND** operation is implied when you provide multiple fields in the query document.

**Example: Find all students in New York who have grade "A"**

db.students.find({ "city": "New York", "grade": "A" })

OR operation

The **OR** operation is performed using the **$or** operator.

**Example: Find all students who are either in New York or have grade "A"**

db.students.find({

$or: [

{ "city": "New York" },

{ "grade": "A" }

]

})

1b Execute the Commands of MongoDB and operations in MongoDB : Insert, Query, Update, Delete

and Projection.

* **Insert**: Use **insertOne()** or **insertMany()** to add documents.
* **Query**: Use **find()** to retrieve documents based on specified criteria.
* **Update**: Use **updateOne()** or **updateMany()** to modify existing documents.
* **Delete**: Use **deleteOne()** or **deleteMany()** to remove documents.
* **Projection**: Use the projection parameter in **find()** to control which fields are returned in the result set
  1. Insert Documents

To insert documents into a collection, you can use the insertOne() or insertMany() methods.

db.students.insertOne({ "name": "Alice", "age": 25, "grade": "A" });

db.students.insertMany([ { "\_id": 1, "name": "Alice", "age": 20, "grade": "A", "city": "New York" },

{ "\_id": 2, "name": "Bob", "age": 22, "grade": "B", "city": "San Francisco" },

{ "\_id": 3, "name": "Charlie", "age": 23, "grade": "A", "city": "New York" },

{ "\_id": 4, "name": "David", "age": 21, "grade": "C", "city": "Los Angeles" },

{ "\_id": 5, "name": "Eve", "age": 20, "grade": "B", "city": "New York" }

]);

**2. Query Documents**

To query documents from a collection, you can use the **find()** method.

db.students.find({ "age": { $gt: 20 } });

3. Update Documents

To update documents in a collection, you can use the updateOne() or updateMany() methods.

db.students.updateOne({ "name": "Alice" }, { $set: { "age": 26 } });

db.students.updateMany(

{ "city": "New York" },

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

);

**4. Delete Documents**

To delete documents from a collection, you can use the **deleteOne()** or **deleteMany()** methods.

db.students.deleteOne({ "name": "Alice" });

db.students.deleteMany({ "grade": "C" });

**5. Projection**

To project specific fields in the result set, you can use the second parameter of the **find()** method.

db.students.find({}, { "name": 1, "age": 1, "\_id": 0 });

2a. Develop a MongoDB query to select certain fields and ignore some fields of the documents from

any collection.

Create a collection of movie document

[

{

"title": "The Dark Knight",

"year": 2008,

"director": "Christopher Nolan",

"genre": ["Action", "Crime", "Drama"],

"actors": ["Christian Bale", "Heath Ledger", "Aaron Eckhart"],

"rating": 9.0

},

{

"title": "Inception",

"year": 2010,

"director": "Christopher Nolan",

"genre": ["Action", "Adventure", "Sci-Fi"],

"actors": ["Leonardo DiCaprio", "Joseph Gordon-Levitt", "Ellen Page"],

"rating": 8.8

},

{

"title": "Interstellar",

"year": 2014,

"director": "Christopher Nolan",

"genre": ["Adventure", "Drama", "Sci-Fi"],

"actors": ["Matthew McConaughey", "Anne Hathaway", "Jessica Chastain"],

"rating": 8.6

},

{

"title": "Parasite",

"year": 2019,

"director": "Bong Joon-ho",

"genre": ["Comedy", "Drama", "Thriller"],

"actors": ["Song Kang-ho", "Lee Sun-kyun", "Cho Yeo-jeong"],

"rating": 8.6

},

{

"title": "The Shawshank Redemption",

"year": 1994,

"director": "Frank Darabont",

"genre": ["Drama"],

"actors": ["Tim Robbins", "Morgan Freeman", "Bob Gunton"],

"rating": 9.3

}

]

In MongoDB, the projection object uses a value of 1 to include a field and 0 to exclude a field. So when you see { field: 1 }, it means include that field, and { field: 0 } means exclude that field.

db.collectionName.find({}, { title: 1, year: 1, \_id: 0 })

To exclude specific fields from the result set using a projection in MongoDB, you would specify those fields with a value of 0 in the projection object. Here's how you would do it

db.movies.find({}, { title: 0, director: 0, \_id: 0 })

2b. Develop a MongoDB query to display the first 5 documents from the results obtained in a.

[use of limit and find]

db.collectionName.find({}, { title: 1, year: 1, director: 1, \_id: 0 }).limit(5)

3a Execute query selectors (comparison selectors, logical selectors ) and list out the results on any

collection

Comparison selector:

$gt (greater than)

$eq: Matches values that are equal to a specified value.

$ne: Matches values that are not equal to a specified value.

$gt: Matches values that are greater than a specified value.

$gte: Matches values that are greater than or equal to a specified value.

$lt: Matches values that are less than a specified value.

$lte: Matches values that are less than or equal to a specified value.

$in: Matches any of the values specified in an array.

db.students.find({

$or: [

{ grade: 11 },

{ grade: 12 }

],

age: { $gt: 20 }

})

Logical selector:

$and: Performs a logical AND operation on an array of expressions.

$or: Performs a logical OR operation on an array of expressions.

$nor: Performs a logical NOR operation on an array of expressions.

$not: Performs a logical NOT operation on an expression.

db.students.find({

age: { $gt: 20 }, // Age greater than 20

city: "New York", // City is New York

grade: { $or: [11, 12] } // Grade is either 11 or 12

})

db.products.find({ price: { $gt: 50 } })

3b Execute query selectors (Geospatial selectors, Bitwise selectors ) and list out the results on any

collection

[

{

"\_id": 1,

"name": "Coffee Shop A",

"location": {

"type": "Point",

"coordinates": [-73.985150, 40.758700]

}

},

{

"\_id": 2,

"name": "Restaurant B",

"location": {

"type": "Point",

"coordinates": [-73.985200, 40.758800]

}

},

{

"\_id": 3,

"name": "Park C",

"location": {

"type": "Point",

"coordinates": [-73.984900, 40.759000]

}

}

]

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

db.place.find({location: { $near: { $geometry: {type: "Point",coordinates: [-73.985130, 40.758896] },$maxDistance: 1000 } }});

b. Execute query selectors (Geospatial selectors, Bitwise selectors ) and list out the results on any

collection

[Refer: Book 3 Chapter 13]

Collection

[

{ "name": "Document A", "status": 6 },

{ "name": "Document B", "status": 3 },

{ "name": "Document C", "status": 7 },

{ "name": "Document D", "status": 8 }

]

Queries

db.flags.find({

status: { $bitsAllClear: 2 } // Matches documents with status 3 and 8

});

db.flags.find({

status: { $bitsAllClear: 2 } // Matches documents with status 3 and 8

});

db.flags.find({

status: { $bitsAnyClear: 8 } // Matches documents with status 6, 3, and 7

});

db.flags.find({

status: { $bitsAnySet: 1 } // Matches documents with status 3 and 7

});

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

MondoDB.

[Refer: Book 3 Chapter 14]

 **$**: The positional operator, used to return the first matching element in an array.

 **$elemMatch**: Projects the first element in an array that matches the specified query conditions.

 **$slice**: Limits the number of elements returned from an array.

[

{

"\_id": 1,

"orderId": 1,

"items": [

{ "itemName": "Laptop", "quantity": 10 },

{ "itemName": "Mouse", "quantity": 5 },

{ "itemName": "Keyboard", "quantity": 15 }

]

},

{

"\_id": 2,

"orderId": 2,

"items": [

{ "itemName": "Monitor", "quantity": 8 },

{ "itemName": "Mouse", "quantity": 20 },

{ "itemName": "USB Cable", "quantity": 13 }

]

}

]

### Using the $ Projection Operator

The $ projection operator is used to return only the first matching element from an array that matches the query condition.

db.orders.find(

{ "items.itemName": "Mouse" },

{ "items.$": 1 }

);

### Using the $elemMatch Projection Operator

The $elemMatch projection operator is used to project the first element in an array that matches the specified $elemMatch condition.

db.orders.find(

{ orderId: 2 },

{ items: { $elemMatch: { itemName: "Mouse" } } }

);

### Using the $slice Projection Operator

The $slice projection operator is used to return a specified number of elements from an array. You can specify a positive number to return the first N elements, a negative number to return the last N elements, or use a combination of two numbers to return a specific range.

db.orders.find(

{ orderId: 1 },

{ items: { $slice: 2 } }

);

db.orders.find(

{ orderId: 1 },

{ items: { $slice: -2 } }

);

db.orders.find(

{ orderId: 1 },

{ items: { $slice: [1, 2] } }

);

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]

[

{ "\_id": 1, "name": "Alice", "age": 24, "score": 85 },

{ "\_id": 2, "name": "Bob", "age": 22, "score": 90 },

{ "\_id": 3, "name": "Charlie", "age": 23, "score": 82 },

{ "\_id": 4, "name": "David", "age": 25, "score": 88 },

{ "\_id": 5, "name": "Eve", "age": 24, "score": 91 }

]

db.students.aggregate([

{

$group: {

\_id: null,

averageScore: { $avg: "$score" }

}

}

])

db.students.aggregate([

{

$group: {

\_id: null,

minAge: { $min: "$age" }

}

}

])

db.students.aggregate([

{

$group: {

\_id: null,

maxScore: { $max: "$score" }

}

}

])

db.students.aggregate([

{

$group: {

\_id: null,

allNames: { $push: "$name" }

}

}

])

db.students.aggregate([

{

$group: {

\_id: null,

uniqueAges: { $addToSet: "$age" }

}

}

])

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 ]

[

{ "\_id": 1, "name": "Alice", "age": 20, "score": 85 },

{ "\_id": 2, "name": "Bob", "age": 22, "score": 90 },

{ "\_id": 3, "name": "Charlie", "age": 21, "score": 88 },

{ "\_id": 4, "name": "David", "age": 23, "score": 82 },

{ "\_id": 5, "name": "Eve", "age": 22, "score": 87 }

]

QUERY

db.students.aggregate([

{ $match: { age: { $gte: 21 } } }, // Match documents where age is 21 or greater

{ $group: { \_id: "$age", avgScore: { $avg: "$score" } } }, // Group by age and calculate average score

{ $sort: { \_id: 1 } }, // Sort by age in ascending order

{ $project: { \_id: 0, age: "$\_id", avgScore: 1 } }, // Project to show age and average score only

{ $skip: 1 } // Skip the first result

])

### Explanation of Each Stage:

* **$match**: Filters documents where the age field is greater than or equal to 21.
* **$group**: Groups documents by age and calculates the average score for each group.
* **$sort**: Sorts the results by age in ascending order.
* **$project**: Projects the results to show only age and avgScore, while excluding \_id.
* **$skip**: Skips the first result (after sorting).

7

a. 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

[

{

"\_id": 1,

"listing\_url": "http://example.com/listing/1",

"name": "Cozy Cottage",

"address": { "country": "USA", "city": "New York" },

"host": { "picture\_url": "http://example.com/host/1.jpg" }

},

{

"\_id": 2,

"listing\_url": "http://example.com/listing/2",

"name": "Urban Apartment",

"address": { "country": "USA", "city": "San Francisco" },

"host": { "picture\_url": "http://example.com/host/2.jpg" }

},

{

"\_id": 3,

"listing\_url": "http://example.com/listing/3",

"name": "Beach House",

"address": { "country": "USA", "city": "Miami" },

"host": { "picture\_url": "" }

}

]

QUERY

db.getCollection('listingsAndReviews').aggregate([

{

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

},

{

$project: {

listing\_url: 1,

name: 1,

address: 1,

host\_picture\_url: "$host.picture\_url"

}

}

]);

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

[refer Book2: chapter 6]

{

"\_id": 1,

"product\_id": "P001",

"product\_name": "Product A",

"reviews": [

{ "review\_id": "R001", "rating": 5, "comment": "Excellent product!", "review\_date": "2023-01-01" },

{ "review\_id": "R002", "rating": 4, "comment": "Very good!", "review\_date": "2023-01-15" }

]

}

QUERY

db.getCollection('ecommerce').aggregate([

{

$unwind: "$reviews"

},

{

$group: {

\_id: "$product\_id",

product\_name: { $first: "$product\_name" },

totalReviews: { $sum: 1 },

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

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

}

},

{

$project: {

\_id: 0,

product\_id: "$\_id",

product\_name: 1,

totalReviews: 1,

averageRating: 1,

allComments: 1

}

}

]);

8

a. Demonstrate creation of different types of indexes on collection (unique, sparse, compound and

multikey indexes)

COLLECTION

[

{

"\_id": 1,

"username": "user1",

"email": "user1@example.com",

"firstName": "John",

"lastName": "Doe",

"tags": ["admin", "premium"]

},

{

"\_id": 2,

"username": "user2",

"email": "user2@example.com",

"firstName": "Jane",

"lastName": "Doe",

"tags": ["user", "basic"]

},

{

"\_id": 3,

"username": "user3",

"email": null,

"firstName": "Jim",

"lastName": "Beam",

"tags": ["user", "guest"]

}

]

 **Open MongoDB Compass** and connect to your MongoDB instance.

  **Navigate to the Collection**:

*  Click on the database and then the collection where you want to create the index.

 **Create Index**:

*  Go to the "Indexes" tab.
* Click on "CREATE INDEX".

 **Fill Index Details**:

*  **Unique Index**:
  + Field: username
  + Type: Ascending (1)
  + Options: Check "Unique"
* **Sparse Index**:
  + Field: email
  + Type: Ascending (1)
  + Options: Check "Sparse"
* **Compound Index**:
  + Fields: lastName (Ascending 1), firstName (Ascending 1)
* **Multikey Index**:
  + Field: tags
  + Type: Ascending (1)

 **Create Index**:

*  Click on "Create Index" to save the index.

b. Demonstrate optimization of queries using indexes.

**Demonstrate Query Optimization**

**Query 1: Use of Unique Index**

db.users.find({ username: "user1" });

The query will use the unique index on the username field to quickly locate the document. The index allows MongoDB to efficiently look up the username without scanning the entire collection.

**Query 2: Use of Sparse Index**

db.users.find({ email: "user2@example.com" });

The query will use the sparse index on the email field. Since the index is sparse, it only includes documents where the email field exists, making the query efficient for cases where not all documents have an email.

**Query 3: Use of Compound Index**

db.users.find({ lastName: "Doe", firstName: "Jane" });

The query will use the compound index on lastName and firstName. The compound index is particularly useful for queries that match on both fields, allowing MongoDB to quickly find documents that satisfy both conditions.

**Query 4: Use of Multikey Index**

db.users.find({ tags: "admin" });

* **Expected Behavior:** The query will use the multikey index on the tags field. Multikey indexes are used to index fields that hold arrays, allowing efficient querying of array elements.

**Checking Index Usage**

To see how MongoDB uses the indexes, you can use the explain() method:

db.users.find({ username: "user1" }).explain("executionStats");

9

1. Develop a query to demonstrate Text search using catalog data collection for a given word

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

Document

[

{

"title": "Introduction to MongoDB",

"description": "A beginner's guide to MongoDB",

"tags": ["database", "NoSQL"]

},

{

"title": "Advanced MongoDB Techniques",

"description": "In-depth coverage of advanced MongoDB topics",

"tags": ["database", "NoSQL", "advanced"]

},

{

"title": "MongoDB Performance Tuning",

"description": "Learn how to optimize MongoDB performance",

"tags": ["database", "performance"]

},

{

"title": "Getting Started with NoSQL",

"description": "An overview of NoSQL databases",

"tags": ["database", "NoSQL", "beginner"]

}

]

**Query**

### Create a Text Index

Create a text index on the fields you want to search. In this example, let's create a text index on the title and description fields.

db.catalog.createIndex({ title: "text", description: "text" });

### Perform a Text Search

Use the $text operator to search for a word, e.g., "MongoDB".

db.catalog.find({ $text: { $search: "MongoDB" } });

b.

**Query to Exclude Documents Containing the Word "Advanced"**

db.catalog.find({

$or: [

{ title: { $not: /Advanced/ } },

{ description: { $not: /Advanced/ } }

]

});

10

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

Documents

[

{

"title": "How to Bake the Perfect Chocolate Cake",

"description": "In this video, we'll show you step-by-step how to bake a delicious chocolate cake from scratch.",

"tags": ["baking", "chocolate cake", "cooking"]

},

{

"title": "Top 10 Python Programming Tips and Tricks",

"description": "Learn the top 10 Python programming tips and tricks to improve your coding skills.",

"tags": ["programming", "Python", "tips and tricks"]

},

{

"title": "Exploring the Wonders of the Universe",

"description": "A documentary exploring the mysteries and wonders of the universe and space exploration.",

"tags": ["science", "universe", "documentary"]

},

{

"title": "How to Start a Successful YouTube Channel",

"description": "Tips and strategies for starting and growing a successful YouTube channel.",

"tags": ["YouTube", "content creation", "marketing"]

},

{

"title": "The Ultimate Guide to Digital Marketing",

"description": "A comprehensive guide to digital marketing strategies and techniques for businesses.",

"tags": ["marketing", "digital marketing", "business"]

},

{

"title": "Meditation for Beginners: A Simple Guide",

"description": "Learn the basics of meditation with this simple guide designed for beginners.",

"tags": ["meditation", "mindfulness", "beginners"]

},

{

"title": "The Future of Artificial Intelligence",

"description": "An insightful discussion on the future developments and impacts of artificial intelligence.",

"tags": ["AI", "technology", "future"]

},

{

"title": "DIY Home Improvement Projects",

"description": "Easy and affordable DIY home improvement projects you can do yourself.",

"tags": ["DIY", "home improvement", "projects"]

}

]

Query

db.videos.createIndex({ title: "text", description: "text" });

db.videos.aggregate([

// Stage 1: Perform Text Search

{

$match: {

$text: { $search: "guide" }

}

},

// Stage 2: Sort by Text Search Score

{

$sort: {

score: { $meta: "textScore" }

}

},

// Stage 3: Project Relevant Fields and Text Search Score

{

$project: {

title: 1,

description: 1,

tags: 1,

score: { $meta: "textScore" }

}

}

]);

Note:

In MongoDB's aggregation pipeline, the score field that is being projected with { $meta: "textScore" } represents the relevance score of each document based on the text search query.