MongoDB commands:

To see a list of all the databases in your MongoDB instance: show databases

To switch to an existing database or create a new one: use <database\_name>

To see the database selected: db

To drop database: db.dropDatabase()

To create a collection: db.createCollection(<collection\_name>)

To show all collections: show collections

To delete a collection and all its documents: db.<collection\_name>.drop()

To insert a single document into a collection: db.<collection\_name>.insertOne({ key: "value" })

To insert multiple documents at once: db.<collection\_name>.insertMany([{ key1: "value1" }, { key2: "value2" }])

To update a single document: db.<collection\_name>.updateOne(

{ <condition> },

{ $set: { <field>: <new\_value> } }

)

To update multiple documents: db.<collection\_name>.updateMany(

{ <condition> },

{ $set: { <field>: <new\_value> } }

)

Retrieve all documents in a collection: db.<collection\_name>.find()

To display results in a structured and more readable format: db.<collection\_name>.find().pretty()

To filter documents based on conditions: db.<collection\_name>.find({ <field>: <value> })

To retrieve a single document that matches a query: db.<collection\_name>.findOne({ <field>: <value> })

To retrieve only specific fields from documents: db.<collection\_name>.find({}, { <field1>: 1, <field2>: 1, \_id: 0 })

Count the total number of documents in a collection: db.<collection\_name>.countDocuments()

Count documents with a filter: db.<collection\_name>.countDocuments({ <field>: <value> })

To limit the number of documents returned: db.<collection\_name>.find().limit(<number>)

To skip a specific number of documents: db.<collection\_name>.find().skip(<number>)

To implement pagination: db.<collection\_name>.find().skip(<number>).limit(<number>)

Sort the documents based on a field:

db.<collection\_name>.find().sort({ <field>: 1 }) // Ascending order

db.<collection\_name>.find().sort({ <field>: -1 }) // Descending order

Retrieve all distinct values for a field: db.<collection\_name>.distinct("<field>")

If you have a text index, perform a text search: db.<collection\_name>.find({ $text: { $search: "<text>" } })

Query documents where a specific field exists or does not exist:

db.<collection\_name>.find({ <field>: { $exists: true } }) // Field exists

db.<collection\_name>.find({ <field>: { $exists: false } }) // Field does not exist

------------------------------------------------------------------------------------------------

Datatypes:

String – Stores text data - Ex: 'white box learning'

Number – Stores integers or floating-point numbers - Ex: 25, 3.14

Boolean – Stores true or false values - Ex: true, false

Date – Stores date and time in ISO 8601 format.

Ex: { "createdAt": ISODate("2023-01-01T00:00:00Z") }

Array – Stores an ordered list of values (can contain mixed types).

Ex: { "tags": ["mongodb", "database", "NoSQL"] }

Object/Embedded Document – Stores nested objects within a document.

Ex: {

"name": "Alice",

"address": {

"city": "New York",

"zip": "10001"

}

}

ObjectId – A unique 12-byte identifier for documents.

Binary Data – Stores binary files, like images or audio - Ex: { "file": BinData(0, "base64encodeddata") }

Null – Represents the absence of a value - Ex: null

---------------------------------------------------------------------------------------------------------

Comparison Operators in MongoDB:

$gt: Greater than

$gte: Greater than or equal to

$lt: Less than

$lte: Less than or equal to

$ne: Not equal to

$in: Matches any value in an array

$nin: Does not match any value in an array

---------------------------------------------------------------------------------------------------------

Logical Operators in MongoDB:

$and: Match all conditions

$or: Match any condition

$not: Negates a condition

$nor: Match none of the conditions

Queries:

Comparison Operators

1. $gt (Greater than): Find candidates whose age is greater than 30

db.Candidates.find({ age: { $gt: 30 } })

2. $lt (Less than): Find candidates whose age is less than 30

db.Candidates.find({ age: { $lt: 30 } })

3. $gte (Greater than or equal to): Find candidates who have 4 or more years of experience

db.Candidates.find({ experience\_years: { $gte: 4 } })

4. $lte (Less than or equal to): Find candidates who have 4 or fewer years of experience

db.Candidates.find({ experience\_years: { $lte: 4 } })

5. $ne (Not equal to): Find candidates whose location is not "New York"

db.Candidates.find({ location: { $ne: "New York" } })

6. $in (Matches any value in an array): Find candidates who are located either in "New York" or "Los Angeles"

db.Candidates.find({ location: { $in: ["New York", "Los Angeles"] } })

7. $nin (Does not match any value in an array): Find candidates whose location is neither "New York" nor "Los Angeles":

db.Candidates.find({ location: { $nin: ["New York", "Los Angeles"] } })

--------------------------------------------------------------------------------------------------------

Logical Operators:

1. $and (Match all conditions): The $and operator is used to match documents where all the conditions are true.

Find candidates who are older than 30 and have more than 3 years of experience

db.Candidates.find({

$and: [

{ age: { $gt: 30 } },

{ experience\_years: { $gt: 3 } }

]

})

2. $or (Match any condition): The $or operator is used to match documents where any of the conditions is true.

Find candidates who either have the skill "ReactJS" or live in "New York"

db.Candidates.find({

$or: [

{ skills: "ReactJS" },

{ location: "New York" }

]

})

3. $not (Negates a condition): The $not operator negates the condition, meaning it matches documents where the condition does not hold true.

Find candidates who do not have "ReactJS" as their skill

db.Candidates.find({

skills: { $not: { $eq: "ReactJS" } }

})

4. $nor (Match none of the conditions): The $nor operator is used to match documents where none of the conditions are true. It's the opposite of $or.

Find candidates who neither have "ReactJS" as their skill nor live in "New York"

db.Candidates.find({

$nor: [

{ skills: "ReactJS" },

{ location: "New York" }

]

})

---------------------------------------------------------------------------------------------------------

Aggregation Pipeline:

Aggregation Pipeline is a powerful framework for data processing that allows you to perform complex transformations and computations on your data. It processes data records in stages, where each stage applies an operation to the data and passes the result to the next stage.

Common Pipeline Stages:

Here are some of the most common aggregation pipeline stages:

$match: Filters the data to only include documents that match the specified criteria (like a query).

$group: Groups the documents by a specified field and applies aggregate functions (such as sum, average, min, max) to other fields.

$sort: Sorts the documents by one or more fields, in ascending or descending order.

$project: Reshapes the documents, specifying which fields to include, exclude, or compute.

$limit: Limits the number of documents passed to the next stage.

$skip: Skips a specified number of documents.

$unwind: Deconstructs an array field from the documents into multiple documents.

$addFields: Adds new fields or modifies existing fields.

$lookup: Joins documents from another collection (like SQL joins).

$count: Counts the number of documents.

---------------------------------------------------------------------------------------------------------

Aggregations: Aggregation in MongoDB is a powerful tool for processing data and transforming it into more meaningful results.

1. $match: This aggregation stage behaves like a find. It will filter documents that match the query provided.

Find Candidates with a Specific Skill (e.g., "Python")

db.candidates.aggregate([

{ $match: { skills: 'Python' } }

]);

using find:

db.candidates.find({ skills: 'Python' })

2. $sort: The $sort stage is used to sort documents in ascending or descending order.

Sort candidates by age in descending order

db.Candidates.aggregate([

{ $sort: { age: -1 } }

])

3. $project: Reshape Documents - The $project stage reshapes the documents, allowing you to include, exclude, or add new fields.

Include only name and age fields in the result

db.Candidates.aggregate([

{ $project: { name: 1, age: 1 } }

])

4. $limit: Limit the Number of Results - The $limit stage restricts the number of documents that are returned.

Limit the results to 5 candidates

db.Candidates.aggregate([

{ $limit: 5 }

])

5. $skip: Skip Documents: The $skip stage is used to skip a specified number of documents.

Skip the first 3 candidates and return the rest

db.Candidates.aggregate([

{ $skip: 3 }

])

6. $group: Groups documents by a specific identifier and performs aggregation operations like sum, average, count, etc.

db.Candidates.aggregate([

{ $group: { \_id: null, avgExperience: { $avg: "$experience\_years" } } }

])

7. $count: The $count operator returns the number of documents

Count the total number of candidates

db.Candidates.aggregate([

{ $count: "totalCandidates" }

])

8. $addFields: Add New Fields to Documents - The $addFields stage adds new fields to documents or modifies existing ones.

Add a new field isSenior based on the candidate's experience\_years

db.Candidates.aggregate([

{ $addFields: { isSenior: { $gte: ["$experience\_years", 5] } } }

])

---------------------------------------------------------------------------------------------------------

db.Candidates.aggregate([

{ $group: { \_id: null, maxExperience: { $max: "$experience\_years" } } }

])

db.Candidates.aggregate([

{ $group: { \_id: null, minExperience: { $min: "$experience\_years" } } }

])

db.Candidates.aggregate([

{ $group: { \_id: null, totalExperience: { $sum: "$experience\_years" } } }

])

db.Candidates.aggregate([

{ $group: { \_id: null, avgExperience: { $avg: "$experience\_years" } } }

])

------------------------------------------------------------------------------------------------------

Pipeline:

db.Candidates.aggregate([

`{

$match: { experience\_years: { $gt: 2 } }

},

{

$addFields: { isSenior: { $gte: ["$experience\_years", 3] } }

},

{

$group: {

\_id: "$location",

totalCandidates: { $sum: 1 },

avgExperience: { $avg: "$experience\_years" }

}

},

{

$sort: { totalCandidates: -1 }

},

{

$project: {

\_id: 1,

totalCandidates: 1,

avgExperience: 1,

isSenior: { $gte: ["$avgExperience", 3] }

}

}

])

-------------------------------------------------------------------------------------------------

$lookup:

The $lookup is a powerful tool in MongoDB that allows you to join two collections based on a common field, similar to SQL joins.

db.createCollection("Jobs"); // This creates the "jobs" collection if it doesn't exist.

db.Jobs.insertMany([

{

"job\_title": "Software Engineer",

"location": "New York",

"company": "TechCorp",

"salary": 120000,

"requirements": ["JavaScript", "ReactJS", "Node.js"]

},

{

"job\_title": "Backend Developer",

"location": "San Francisco",

"company": "DevSolutions",

"salary": 140000,

"requirements": ["Java", "Spring Boot", "Microservices"]

}

]);

db.Candidates.aggregate([

{

$lookup: {

from: "jobs", // The collection you are looking up from (e.g., "jobs")

localField: "location", // The field from the "candidates" collection (joining field)

foreignField: "location", // The field from the "jobs" collection (joining field)

as: "job\_info" // The name of the array where the matched results will be stored

}

}

])

------------------------------------------------------------------------------------------------------