Introduction to MongoDB

MongoDB is an open-source, cross-platform, and distributed document-based database designed for ease of application development and scaling. It is a NoSQL database developed by MongoDB Inc.

MongoDB database is built to store a huge amount of data and also perform fast.

MongoDB is not a Relational Database Management System (RDBMS). It's called a "NoSQL" database. It is opposite to SQL based databases where it does not normalize data under schemas and tables where every table has a fixed structure. Instead, it stores data in the collections as JSON based documents and does not enforce schemas. It does not have tables, rows, and columns as other SQL (RDBMS) databases. It is one of the most popular and widely used NoSQL databases.

A NoSQL database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases.

RDBMS vs MongoDB:

RDBMS has a typical schema design that shows number of tables and the relationship between these tables whereas MongoDB is document-oriented. There is no concept of schema or relationship.

Complex transactions are not supported in MongoDB because complex join operations are not available.

MongoDB allows a highly flexible and scalable document structure. For example, one data document of a collection in MongoDB can have two fields whereas the other document in the same collection can have four.

MongoDB is faster as compared to RDBMS due to efficient indexing and storage techniques.

There are a few terms that are related in both databases. What's called Table in RDBMS is called a Collection in MongoDB. Similarly, a Tuple is called a Document and A Column is called a Field. MongoDB provides a default '_id' (if not provided explicitly) which is a 12-byte hexadecimal number that assures the uniqueness of every document. It is similar to the Primary key in RDBMS.

Where do we use MongoDB?

MongoDB is preferred over RDBMS in the following scenarios:

Big Data: If you have huge amount of data to be stored in tables, think of MongoDB before RDBMS databases. MongoDB has built-in solution for partitioning and sharing your database.

Unstable Schema: Adding a new column in RDBMS is hard whereas MongoDB is schema-less. Adding a new field does not effect old documents and will be very easy.

Distributed data: Since multiple copies of data are stored across different servers, recovery of data is instant and safe even if there is a hardware failure.

Who's using MongoDB?

MongoDB has been adopted as backend software by a number of major websites and services including EA, Cisco, Shutterfly, Adobe, Ericsson, Craigslist, eBay, and Foursquare.

Advantages of MongoDB:

It is a schema-less NoSQL database. You need not to design the schema of the database when you are working with MongoDB.

- It does not support join operation.
- It provides great flexibility to the fields in the documents.
- It contains heterogeneous data.
- It provides high performance, availability, scalability.
- It supports Geospatial efficiently.
- It is a document-oriented database and the data is stored in BSON documents.
- It also supports multiple document ACID transition (string from MongoDB 4.0).
- It provides drivers to store and fetch data from different applications developed in different technologies such as C#, Java, Python, Node.js, etc.
- It does not require any SQL injection.
- It is easily integrated with Big Data Hadoop
- It allows us to split data across multiple servers.
- It provides tools to manage MongoDB databases.

Disadvantages of MongoDB:

- It uses high memory for data storage.
- You are not allowed to store more than 16MB data in the documents.
- The nesting of data in BSON is also limited you are not allowed to nest data more than 100 levels.

Installing MongoDB:

- http://www.mongodb.org/downloads and select your operating system out of Windows, Linux, Mac OS X and Solaris.
- A detailed explanation about the installation of MongoDB is given on their site.

  https://www.mongodb.com/docs/manual/installation/

To install a free MongoDB database server on our local Windows machine. So, click on the Product menu -> Community Edition->MongoDB Community Server Download->Select package->Click on Download.

InstallingMongoDB Shell

MongoDB Shell is the quickest way to connect, configure, query, and work with your MongoDB database. It acts as a command-line client of the MongoDB server.

Note: MongoDB Shell is an open source (Apache 2.0), standalone product developed separately from the MongoDB Server.

MongoDB Shell is already installed with MongoDB. You can find it in the installation directory where you installed MongoDB. By default, it is "C:\Program Files\MongoDB\Server". Open the installation folder and appropriate version folder and go to the "bin" folder. Here, "mongo.exe" is MongoDB shell. Click on it to open the MongoDB shell, as shown below.

Click on the Product menu -> Community Edition->Tools->MongoDB Shell Download-> click on Download.

Click on the Download button to download the installer file.

Now, click on the downloaded installer file to start the installation wizar.

Once installation completes, click the Finish button to close the wizard.

Execute MongoDB Commands

Open a new command prompt on Windows and write **mongosh** and press Enter. It will open the same MongoDB shell.

You can execute MongoB commands for CRUD operations on MongoDB shell (mongo or mongosh). For example, execute the "shows dbs" command to see all the databases on the connected MongoDB server.

```
> show dbs
admin 41 kB
config 111 kB
local 41 kB
```

Use the "db" command to check the current database.

```
>db
test
```

Run the .editor command to execute multi-line commands. Press Ctrl + d to run a command or Ctrl + c to cancel.

MongoDB Shell Commands

MongoDB Shell is the quickest way to connect, configure, query, and work with your MongoDB database. It acts as a command-line client of the MongoDB server.

You can start MongoDB Shell by executing mongo or mongosh command on the command prompt/terminal. mongosh is the new MongoDB shell with some more features than the old mongo shell.

mongosh<commands>

The --help command display all the commands which you can use with mongo or mongosh

The **show dbs** command will display all the databases on the connected server.

The **show collections** commands shows collections in current database admin.

Use the **db.help()** command to get the help on db command.

Use **db.test.help()** command to get the help on collection related commands.

Connect to MongoDB Database

By default, the mongosh or mongo command connects with the local MongoDB database on the localhost:27017. So, mongosh and mongosh "mongodb://localhost:27017" are the same that connect to a database on the localhost at port 27017.

To connect with the local database on a different port, use the --port option:

C:\>mongosh --port 23023

The following connects to the remote database on mymongodb.example.com at port 23023.

C:\>mongosh "mongodb://mymongodb.example.com:23023"

Or, use the --host and --port options:

mongosh --host mongodb0.example.com --port 28015

Use the --username and --authenticationDatabase command-line options to connect with the database that requires authentication.

mongosh "mongodb://mymongodb.example.com:23023" --username steve --authenticationDatabase admin

CRUD Operations in MongoDB

CRUD operations create, read, update, and delete documents.

You can execute MongoDB commands for CRUD operations on MongoDB shell (mongo or mongosh).

Create Operations:

Creating a database

Syntax: use <DBname>

This command will create a DB but then it is actually created only if we add a collection to it.

To see all databases:

Execute the "shows dbs" command to see all the databases on the connected MongoDB server.

Drop a database:

To drop a database in MongoDB, you can use the db.dropDatabase() method.

Step1: Switch to the database that you want to drop. You can use the use command:

use your_database_name

Step2: Once you are inside the database, run the following command to drop the database:

db.dropDatabase()

Note: Be cautious while using the dropDatabase command, as it irreversibly removes the entire database and its data. Double-check that you are targeting the correct database before executing the command

Collections in MongoDB

In MongoDB, a collection is a group of MongoDB documents. It is the equivalent of an RDBMS (Relational Database Management System) table. Collections do not enforce a schema, which means that documents within a collection can have different fields and structures.

Here are some key points about MongoDB collections:

Documents: A collection is made up of MongoDB documents, which are JSON-like BSON (Binary JSON) documents. Each document within a collection can have its own unique structure. However, it is common to have documents with a similar structure within a collection.

Schema-less: MongoDB is often referred to as a schema-less database because collections do not enforce a rigid schema. This flexibility allows you to insert documents with different fields into the same collection.

Indexing: Collections can have indexes to improve query performance. Indexes in MongoDB are similar to indexes in relational databases and are used to optimize queries and improve the speed of data retrieval.

Atomic Operations: MongoDB supports atomic operations at the document level. This means that a single document can be updated atomically, and the changes are visible to other operations once the update is complete

Dynamic Creation: Collections are created dynamically when the first document is inserted into them. You don't need to define the structure of the collection explicitly; MongoDB will create the collection on the fly.

CRUD Operations: Collections support CRUD (Create, Read, Update, Delete) operations. You can insert documents into a collection, query documents, update existing documents, and delete documents.

Creating a collection:

Syntax: db.createCollection(<collection-name>);

Example: db.createCollection(“kmitstudents”);

Also, we don't need to create collection. MongoDB creates collection automatically, when you insert some document

Example: db.kmit.insert({"name" : “kmitstudent"})

A collection by name kmit will be created to which a document with the data will be added.

Drop a collection:

Step1: Switch to the database that contains the collection you want to delete.

You can use the use command:

use your_database_name

Step2: Once you are inside the database, run the following command to drop the your collection :

db.COLLECTION_NAME.drop();

For example, if you want to delete a collection named "myCollection" in the "myDatabase" database, the command would be:

use myDatabase

```
db.myCollection.drop()
```

Note: Be careful when using the drop() method, as it permanently removes the collection and its data. Ensure that you are targeting the correct collection before executing the command, and consider creating a backup if necessary

Insert a document to collection:

To insert a document into a MongoDB collection, you can use the insertOne() method or insertMany() method or insert() method, depending on whether you want to insert a single document or multiple documents

insert():

It has been deprecated in recent versions of MongoDB in favor of the more specific insertOne() and insertMany() methods. These newer methods provide more clarity and better align with the operations they perform.

Syntax: >db.COLLECTION_NAME.insert(document)

Example: >db.kmit.insert({"name" : “kmitstudent"})

Note: In the inserted document, if we don't specify the _id parameter, then MongoDB assigns a unique ObjectId for this document

If we want to add the unique id as well

Example: >db.kmit.insert({ _id: 10, name : “kmitstudent1"})

insertOne():

Syntax: >db.COLLECTION_NAME.insertOne(document)

Example: >db.kmit.insertOne({"name" : “kmitstudent"})

insertMany():

Syntax: >db.COLLECTION_NAME.insertMany([{document 1},{document 2},…])

Example:

```
db.empDetails.insertMany(
        [
                {
                        First_Name: "Radhika",
                        Last_Name: "Sharma",
                        Age: "26",
                        e_mail: "radhika_sharma.123@gmail.com",
                        phone: "9000012345"
```

```
},
{
        First_Name: "Rachel",
        Last_Name: "Christopher",
        Age: "27",
        e_mail: "Rachel_Christopher.123@gmail.com",
        phone: "9000054321"
},
{
        First_Name: "Fathima",
        Last_Name: "Sheik",
        Age: "24",
        e_mail: "Fathima_Sheik.123@gmail.com",
        phone: "9000054321"
}])
```

Read Operations (Query documents from a collection):

Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection**:**

db.collection.find()

The find() method in MongoDB is used to query documents from a collection. It allows you to retrieve documents that match a specified condition or criteria. The find() method returns a cursor, which you can iterate over to access the matching documents.

Syntax: db.Collection_name.find({selection_criteria}, {projection})

selection_criteria: It specifies selection criteria. To return all documents in a collection use empty document({}). The type of this parameter is document.

projection: It specifies the fields to return in the documents that match the selection criteria. To return all fields in the matching documents, remove this parameter.

Ex: db.books.find() or db.student.find({})

db.student.find({age:18})

Projection - selecting only the necessary data rather than selecting whole of the data of a document.

If a document has 5 fields and we need to show only 3, then select only 3 fields from them

basic syntax of find() method with projection is -

db.COLLECTION_NAME.find({},{KEY:1})

_id field is always displayed while executing find() method, if we don't want this field, then we need to set it as 0 –

db.mycol.find({},{_id:0})

If we want a column 'name' to be displayed and we do not want id to be displayed-

db.mycol.find({},{"name":1,_id:0})

findOne()

The findOne() method in MongoDB is similar to the find() method, but it is used to retrieve a single document that matches the specified criteria. Once a matching document is found, findOne() returns that document.

Syntax: db.Collection_name.findOne({selection_criteria}, {projection})

Example: db.books.findOne({age:18})

Update documents inMongoDB

To update documents in MongoDB, you can use the

1. updateOne() or
2. updateMany() method, or
3. update() method (has been deprecated as of MongoDB version 4.2.)

1. **updateOne()**

The updateOne() method in MongoDB is used to update a single document that matches the specified filter criteria.

Syntax:

```
db.COLLECTION_NAME.updateOne(
    {          },

    {$set: {key:value},…}
);
```

2. **updateMany()**

The updateMany() method allows you to update multiple documents that match the specified criteria in a single operation

Syntax:

```
db.COLLECTION_NAME.updateMany(
        {
         },
        {$set: {key:value},…}
);
```

Example: Write a query to update all the users' status as ‘reject’ where theusers age is less than 18.

Collection Name : users

Keys: name,uid,age,status

```
db.users.updateMany(
 { age: { $lt: 18 } },
 {
   $set: {status:”reject”}
 }
)
```

Delete documents in MongoDB

Delete - Delete operations remove documents from a collection.

1. db.collection.deleteOne()
2. db.collection.deleteMany()
3. db.collection.delete() (method (has been deprecated as of MongoDB version 4.2.)

1. db.collection.deleteOne()

The deleteOne() method in MongoDB is used to delete a single document that matches the specified filter criteria.

Syntax:

```
db.COLLECTION_NAME.deleteOne(
        {           },
);
```

2. db.collection.deleteMany()

The deleteMany() method in MongoDB is used to delete all documents that matches the specified filter criteria.

Syntax:

```
db.COLLECTION_NAME.deleteMany(
        {           },
);
```

FindAndModify and Update , with upsert

- Using Update -
- db.student.findAndModify({query:{name:"Jack"},
- update:{$set:{address:"Australia"}},
- upsert:true})

If a student by name Jack exists then it will update the document, else it will insert a new document, because upsert if set to true.

- Using Update -
- db.employee.update({name:“John"}, {$set: {department: "HR"}},{upsert:true})

Additional Information:

Query and Projection Operators

Comparison Operators

For comparison of different BSON type values, see the specified BSON comparison order.

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

Syntax: { field: { $eq: value } }

Example: { age: { $eq: 25 } }

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

Syntax: { field: { $ne: value } }

Example: { status: { $ne: "inactive" } }

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

Syntax: { field: { $gt: value } }

Example: { quantity: { $gt: 10 } }

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

Syntax: { field: { $gte: value } }

Example: { price: { $gte: 50 } }

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

Syntax: { field: { $lt: value } }

Example: { rating: { $lt: 3.5 } }

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

Syntax: { field: { $lte: value } }

Example: { score: { $lte: 100 } }

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

Syntax:{ field: { $in: [<value1>, <value2>, ... <valueN>] } }

Example: { score: { $in:[100,12,190] } }

$nin: Matches none of the values specified in an array.

Syntax: { field: { $nin: [<value1>, <value2>, ... <valueN>] } }

Example: { score: { $nin:[100,12,190] } }

LogicalOperators

In MongoDB, logical operators are used in queries to combine multiple conditions, allowing you to express complex criteria for document retrieval.

$and: Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.

Syntax:

```
db.COLLECTION_NAME.find(
        {
                $and: [
                        {<key1>:<value1>}, { <key2>:<value2>},.....
                 ]
      }
)
```

$not: Inverts the effect of a query expression and returns documents that do not match the query expression.

Syntax:

```
db.COLLECTION_NAME.find(
        {
                field: { $not: { condition1,condittion2,…. }  }
      }
)
```

$nor: Joins query clauses with a logical NOR returns all documents that fail to match both clauses.

Syntax:

```
db.COLLECTION_NAME.find(
```

```
    {
        $nor: [ { condition1 }, { condition2 }, ... ]
    }
)
```

$or: Joins query clauses with a logical OR returns all documents that match the conditions of either clause.

Syntax:

```
db.COLLECTION_NAME.find(
    {
        $or: [
            {key1: value1}, {key2:value2},……
        ]
    }
).pretty()
```

Example1:

```
db.product.find({ $or: [ { status: "A" }, { qty: { $lt: 30 } } ] })
```

Example2 : In the following example, the compound query document selects all documents in the collection where the status equals "A" **and** *either* qty is less than ($lt) 30 *or* item starts with the character p:

```
db.product.find ({ status: "A", $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ] })
```

ElementOperators

$exists: Matches documents that have the specified field.

$type: Selects documents if a field is of the specified type.

EvaluationOperators

$expr: Allows use of aggregation expressions within the query language.

$jsonSchema: Validate documents against the given JSON Schema.

$mod: Performs a modulo operation on the value of a field and selects documents with a specified result.

$regex: Selects documents where values match a specified regular expression.

$text: Performs text search.

$where: Matches documents that satisfy a JavaScript expression.

GeospatialOperators

$geoIntersects: Selects geometries that intersect with a GeoJSON geometry. The 2dsphere index supports $geoIntersects.

$geoWithin: Selects geometries within a bounding GeoJSON geometry. The 2dsphere and 2d indexes support $geoWithin.

$near: Returns geospatial objects in proximity to a point. Requires a geospatial index. The 2dsphere and 2d indexes support $near.

$nearSphere: Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The 2dsphere and 2d indexes support $nearSphere.

ArrayOperators

$all: Matches arrays that contain all elements specified in the query.

$elemMatch: Selects documents if element in the array field matches all the specified $elemMatch conditions.

$size: Selects documents if the array field is a specified size.

BitwiseOperators

$bitsAllClear: Matches numeric or binary values in which a set of bit positions all have a value of 0.

$bitsAllSet: Matches numeric or binary values in which a set of bit positions all have a value of 1.

$bitsAnyClear: Matches numeric or binary values in which any bit from a set of bit positions has a value of 0.

$bitsAnySet: Matches numeric or binary values in which any bit from a set of bit positions has a value of 1.

Projection Operators

$: Projects the first element in an array that matches the query condition.

$elemMatch: Projects the first element in an array that matches the specified $elemMatch condition.

$meta: Projects the document's score assigned during $text operation.

$slice: Limits the number of elements projected from an array. Supports skip and limit slices.

Miscellaneous Operators

$comment: Adds a comment to a query predicate.

$rand: Generates a random float between 0 and 1.

Regular Expressions in MongoDB

The **$regex** operator of MongoDB is implemented for searching any specific string from the collection.

Here is an example that shows how it is usually done.

- db.books.find({"title":{$regex:/h/i}}).pretty()
- Starts with:

db.books.find({"title":{$regex:/^h/i}}).pretty()

- Ends with:

db. books.find({title : {$regex : /w$/i}})

Aggregation Pipeline in MongoDB

In MongoDB, aggregation operations process the data records/documents and return computed(aggregated) results.

MongoDB provides three ways to perform aggregation

1. Single-purpose aggregation
2. Map-reduce function
3. Aggregation pipeline

Note: Most recommended approach is Aggregation pipeline

1. Single Purpose Aggregation:

Simple aggregations on documents can be carried out by using Single purpose aggregation

It provides the access to the common aggregation processes using the sort(),count(), skip(),distinct(), and estimatedDocumentCount(),…..methods.

2. Map-reduce function:

To extract results from large chunks of data we use Map Reduce

Map reduce has two main functions

- Map: It groups all the documents
- Reduce: It performs operations on grouped data.

3. Aggregation pipeline:

An aggregation pipeline consists of one or more stages that process documents:

- Each stage performs an operation on the input documents. For example, a stage can filter documents, group documents, and calculate values.
- The documents that are output from a stage are passed to the next stage.
- An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.

Aggregation pipeline Started in MongoDB 4.2, you can update documents with an aggregation pipeline

Aggregation Pipeline Syntax: The aggregation pipeline is an array of one or more stages passed in the db.aggregate() or db.collectionName.aggregate() method.

db.collectionName.aggregate(pipeline, options);

*collectionName – is the name of a collection,

*pipeline – is an array that contains the aggregation stages([]),

*options – optional parameters for the aggregation

Syntax: db.collection.aggregate([{stage1}, {stage2}, {stage3}...])

Stages: Each stage starts from stage operators.

Some of the important stages are:

Stage	Description
$match:	It is used for filtering the documents can reduce the amount of documents that are given as input to the next stage.
$project:	It is used to select some specific fields from a collection.
$count:	It is used in the aggregation pipeline to return the count of documents that match the specified criteria.
$group:	It is used to group documents based on some value.
$sort:	It is used to sort the document that is rearranging them
$skip:	It is used to skip n number of documents and passes the remaining documents
$limit:	It is used to pass first n number of documents thus limiting them.
$unwind:	It is used to unwind documents that are using arrays i.e. it deconstructs an array field in the documents to return documents for each element.
$lookup:	This allows you to combine documents from two collections based on a specified condition.
$out:	It is used to write resulting documents to a new collection

$match stage in MongoDB

In MongoDB's aggregation framework, the $match stage is used to filter and select documents that match a specified condition.

It is similar to the find method in regular queries but is part of the aggregation pipeline, allowing you to filter documents as part of a larger aggregation operation

Syntax:

```
db.collection.aggregate([
 {
  $match: {
   // Your condition here
   field1: value1,
   field2: value2
  }
 },
 // Additional pipeline stages can follow
```

]);

Consider the emp collection to demonstrates an aggregation pipeline stages.

Collection Name : emp

Keys : "_id" ,"empno", "name" ,"age" ,"gender" ,"department" ,"year" ,

"salary

The following example demonstrates an aggregation pipeline with a single $match stage.

Query: Print all Female employees in the organization?

Solution:

db.emp.aggregate([{$match:{gender:"Female"}}])

$count stage in MongoDB

In MongoDB, the $count stage is used in the aggregation pipeline to return the count of documents that match the specified criteria.

It's particularly useful when you want to know the number of documents in a collection that satisfy certain conditions.

Syntax:

```
db.collection.aggregate([
    { },  // previous stage
    {$count: "string"},          //count stage
    { }  // next stage
  ]);
```

The following example demonstrates an aggregation pipeline with a $count stage.

Query: How many Female employees in the organization?

Solution:

```
db.emp.aggregate([{$match:{gender:"Female"}},
                  {$count:'female are’}
                        ])
```

$group stage in MongoDB

In MongoDB's aggregation framework, the $group stage is used to group documents by a specified expression and perform aggregate operations on these groups.

It is often used in conjunction with other aggregation stages to analyze and transform data in a collection.

Syntax:

```
{
  $group: {
    _id: <expression>,  // Grouping key
    <field1>: { <accumulator1>: <expression1> },
    <field2>: { <accumulator2>: <expression2> },
    // Additional fields and accumulators
  }
}
```

The following example demonstrates an aggregation pipeline with a $group stage.

Query: What departments are available in the organization?

Solution:

```
db.emp.aggregate([{$group:{_id:"$department"}}])
```

$limit stage in MongoDB

In MongoDB, the $limit stage is used in the aggregation pipeline to restrict the number of documents that are passed to the next stage. It takes a single numeric argument that specifies the maximum number of documents to allow through to the next stage.

Here is a simple example of using $limit in an aggregation pipeline:

```
db.collection.aggregate([
  { $match: { status: "active" } },  // Match documents with status "active"
  { $limit: 10 }  // Limit the output to a maximum of 10 documents
]);
```

In this example:

The first stage, $match, filters documents based on a condition (in this case, documents with the "active" status).

The second stage, $limit, restricts the output to a maximum of 10 documents.

It's important to place the $limit stage in the pipeline after stages that reduce the result set (e.g., $match, $filter, etc.) to ensure that you are limiting the number of documents after applying any necessary filters.

Keep in mind that using $limit in aggregation pipelines might have performance implications, especially when dealing with large datasets, as it may affect the efficiency of certain optimization strategies.

$sort stage in MongoDB

In MongoDB, the $sort stage is used in the aggregation pipeline to reorder documents based on specified criteria. It allows you to sort the input documents and pass them to the next stage in the pipeline.

Here is a basic example of using $sort in an aggregation pipeline:

```
db.collection.aggregate([
  { $match: { status: "active" } },  // Match documents with status "active"
  { $sort: { createdAt: -1 } }  // Sort by the "createdAt" field in descending order
]);
```

In this example:

The first stage, $match, filters documents based on a condition (in this case, documents with the "active" status).

The second stage, $sort, sorts the documents based on the createdAt field in descending order (-1 indicates descending order, and 1 would indicate ascending order).

You can sort by one or more fields, and you can specify ascending or descending order for each field.

Here's an example of sorting by multiple fields:

```
db.collection.aggregate([
  { $sort: { category: 1, createdAt: -1 } }
]);
```

In this example, the documents are first sorted by the category field in ascending order and then, within each category, sorted by the createdAt field in descending order.

$look stage in MongoDB

The $lookup stage in MongoDB's aggregation pipeline is used to perform a left outer join between documents from the current collection (referred to as the "left" collection) and documents from another collection (referred to as the "right" collection). This allows you to combine documents from two collections based on a specified condition.

Here's a basic example of how to use $lookup:

```
db.orders.aggregate([
  {
    $lookup: {
      from: "products",        // The "right" collection
```

```
      localField: "product_id",  // Field from the "left" collection
      foreignField: "_id",      // Field from the "right" collection
      as: "productInfo"         // Output field containing the joined data
    }
  }
]);
```

In this example:

The $lookup stage specifies the "right" collection as "products."

It then defines the localField as "product_id" in the "left" collection (the collection being aggregated, which is assumed to be "orders").

The foreignField is specified as "_id" in the "right" collection ("products").

The result is stored in a new field called "productInfo" in the output documents.

This operation effectively adds the "productInfo" field to each document in the "orders" collection, containing the matching document from the "products" collection based on the specified fields.

$out aggregation stage in MongoDB

In MongoDB, the $out aggregation stage is used to write the results of an aggregation pipeline to a specified collection. It allows you to store the output of the aggregation pipeline in a new or existing collection.

Here's a basic example of how to use $out in a MongoDB aggregation pipeline:

```
db.sourceCollection.aggregate([
 // Your aggregation pipeline stages here
 {
  $group: {
   _id: "$fieldToGroupBy",
   total: { $sum: "$numericField" }
  }
 },
 {
  $out: "outputCollection" // Specify the collection where the result will be stored
 }
```

])

In this example:

sourceCollection is the collection you are aggregating.

The $group stage groups documents by a specific field (fieldToGroupBy) and calculates the total sum for a numeric field (numericField).

The $out stage specifies the output collection as "outputCollection".

After running this aggregation pipeline, the results will be stored in the "outputCollection" collection. If "outputCollection" does not exist, MongoDB will create it. If "outputCollection" already exists, MongoDB will replace its contents with the new aggregated data.

MongoDB Integration with ExpressJS

Integrating MongoDB with Express.js is a common task in web development, especially when building Node.js applications. MongoDB is a NoSQL database, and Express.js is a popular web framework for Node.js.

Below are the steps to integrate MongoDB with Express.js:

Step1 : Create a Project folder

Step2 : Generate package.json file (npm init)

Step3 : Install express and mongoose (npm i express mongoose)

Step4 : Import required modules into project (index.js)

Step5 : connect to the MongoDB

Step6 : Create a Schema

Step7 : Create a model

Step8 : Create routes based on queries

Step9 : Run the Server

Step10 : Give request from browser

Step1 : Create a Project folder

Here we are going to create an express application because all of our work will be going to execute inside express.

Step2 : Generate package.json file

Make sure that you have successfully installed npm. The npm init will ask you for some configuration about your project and that is super easy to provide.

Use the following command for that:

npm init -y

This will ask you for few configurations about your project you can fill them accordingly, also you can change it later from the package.json file.

Step3 : Install express and mongoose (npm i express mongoose)

Install necessary dependencies for our application.

Write these commands in your terminal to start a node app and then install express.

npm install express mongoose

express: The web framework for Node.js.

mongoose: A MongoDB object modeling tool designed to work in an asynchronous environment.

Step4 : Import required modules into project (index.js)

Create a basic Express app in your main JavaScript file (e.g., app.js or index.js).

Write the following code in index.js file:

```
const express = require('express');
const mongoose = require('mongoose');
const app = express();
const port = 8085;
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(port, () => {
  console.log(`Server is listening at http://localhost:${port}`);
});
```

Step5 : connect to the MongoDB

Use mongoose to connect to your MongoDB database. Replace the connection URL (your-mongodb-url) with your actual MongoDB connection string.

Add the following code to index.js file:

```
mongoose.connect(`mongodb://127.0.0.1:27017/your_DB_Name`)
.then(()=>{
   console.log(`MongoDB Connected on port : 27017`)
})
.catch((err)=>{ console.log(err)});
```

Step6 : Create a Schema

Schema define the structure of documents within a collection.

Add the following code to index.js file:

```
let myschema=new mongoose.Schema({
```

```
    fname:String,
    lname:String,
    rno:Number
})
```

Step7 : Create a model

Define a MongoDB model using Mongoose. Models define the structure of documents within a collection.

Add the following code to index.js file:

let mymodel=new mongoose.model(Your_Collection_Name,Your_Schema_Name);

Step8 : Create routes based on queries

Routes are the endpoints of the server, which are configured on our backend server and whenever someone tries to access those endpoints they respond accordingly to their definition at the backend.

Now that you have the model, you can perform CRUD (Create, Read, Update, Delete) operations on your MongoDB database through your model.

Add the following code to index.js file for inserting documents to collection:

```
app.get('/spidy',(req,res)=>{
    let mydata=new mymodel({fname:'spidy',lname:'koti',rno:111});
    //to store data in collection use save()
    mydata.save().then(()=>{
        res.send(`Spidy data is inserted`)
    })
})
```

Step9 : Run the Server

Save this code, start the server using “node index.js” at your terminal.

The successful start of the server denotes that our express app is ready to listen to the connection on the specified path(localhost:8085 in our example).

Step10 : Give request from browser

Open the web browser and give a request, localhost on the given port. When client request with the appropriate method on the specified path ex: GET request on ‘/’ path, our function is returning the response as plain text If we open the network section in chrome developers tools (press Ctrl+Shift+I to open) we will see the response returned by the localhost along with all information.

This example provides a basic setup, and you can extend it based on your application's specific requirements.