501-01 : Advance Web Designing

Unit-1: Concepts of NoSQL: MongoDB

MongoDB is a No SQL database. It is an open-source, cross-platform, document-oriented database written in C++.

Our MongoDB tutorial includes all topics of MongoDB database such as insert documents, update documents, delete documents, query documents, projection, sort() and limit() methods, create a collection, drop collection, etc. There are also given MongoDB interview questions to help you better understand the MongoDB database.

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.

In simple words, you can say that - Mongo DB is a document-oriented database. It is an open source product, developed and supported by a company named 10gen.

MongoDB is available under General Public license for free, and it is also available under Commercial license from the manufacturer.

The manufacturing company **10gen** has defined MongoDB as:

"MongoDB is a scalable, open source, high performance, document-oriented database." - 10gen"

MongoDB was designed to work with commodity servers. Now it is used by the company of all sizes, across all industry.

History of MongoDB

The initial development of MongoDB began in 2007 when the company was building a platform as a service similar to window azure.

Window azure is a cloud computing platform and infrastructure, created by Microsoft, to build, deploy and manage applications and service through a global network.

MongoDB was developed by a NewYork based organization named 10gen which is now known as MongoDB Inc. It was initially developed as a PAAS (Platform as a Service). Later in 2009, it is introduced in the market as an open source database server that was maintained and supported by MongoDB Inc.

The first ready production of MongoDB has been considered from version 1.4 which was released in March 2010.

MongoDB2.4.9 was the latest and stable version which was released on January 10, 2014.

1.1 concepts of NoSQL. Advantages and features.

Purpose of building MongoDB

It may be a very genuine question that - "what was the need of MongoDB although there were many databases in action?"

There is a simple answer:

All the modern applications require big data, fast features development, flexible deployment, and the older database systems not competent enough, so the MongoDB was needed.

The primary purpose of building MongoDB is:

- Scalability
- Performance
- High Availability
- Scaling from single server deployments to large, complex multi-site architectures.
- Key points of MongoDB
- Develop Faster
- Deploy Easier
- Scale Bigger

First of all, we should know what is document oriented database?

Example of document oriented database

MongoDB is a document oriented database. It is a key feature of MongoDB. It offers a document oriented storage. It is very simple you can program it easily.

MongoDB stores data as documents, so it is known as document-oriented database.

FirstName = "John",

Address = "Detroit",

Spouse = [{Name: "Angela"}].

FirstName ="John",

Address = "Wick"

There are two different documents (separated by ".").

Storing data in this manner is called as document-oriented database.

Mongo DB falls into a class of databases that calls Document Oriented Databases. There is also a broad category of database known as No SQL Databases.

Features of MongoDB

These are some important features of MongoDB:

1. Support ad hoc queries

In MongoDB, you can search by field, range query and it also supports regular expression searches.

2. Indexing

You can index any field in a document.

3. Replication

MongoDB supports Master Slave replication.

A master can perform Reads and Writes and a Slave copies data from the master and can only be used for reads or back up (not writes)

4. Duplication of data

MongoDB can run over multiple servers. The data is duplicated to keep the system up and also keep its running condition in case of hardware failure.

5. Load balancing

It has an automatic load balancing configuration because of data placed in shards.

6. Supports map reduce and aggregation tools.

7. Uses JavaScript instead of Procedures.

8. It is a schema-less database written in C++.

9. Provides high performance.

10. Stores files of any size easily without complicating your stack.

11. Easy to administer in the case of failures.

12. It also supports:

JSON data model with dynamic schemas

Auto-sharding for horizontal scalability

Built in replication for high availability

Now a day many companies using MongoDB to create new types of applications, improve performance and availability.

NoSQL Databases

We know that MongoDB is a NoSQL Database, so it is very necessary to know about NoSQL Database to understand MongoDB throughly.

What is NoSQL Database

Databases can be divided in 3 types:

1. RDBMS (Relational Database Management System)
2. OLAP (Online Analytical Processing)
3. NoSQL (recently developed database)

NoSQL Database

NoSQL Database is used to refer a non-SQL or non relational database.

It provides a mechanism for storage and retrieval of data other than tabular relations model used in relational databases. NoSQL database doesn't use tables for storing data. It is generally used to store big data and real-time web applications.

History behind the creation of NoSQL Databases

In the early 1970, Flat File Systems are used. Data were stored in flat files and the biggest problems with flat files are each company implement their own flat files and there are no standards. It is very difficult to store data in the files, retrieve data from files because there is no standard way to store data.

Then the relational database was created by E.F. Codd and these databases answered the question of having no standard way to store data. But later relational database also get a problem that it could not handle big data, due to this problem there was a need of database which can handle every types of problems then NoSQL database was developed.

Advantages of NoSQL

- It supports query language.
- It provides fast performance.
- It provides horizontal scalability.

MongoDB advantages over RDBMS

In recent days, MongoDB is a new and popularly used database. It is a document based, non relational database provider.

Although it is 100 times faster than the traditional database but it is early to say that it will broadly replace the traditional RDBMS. But it may be very useful in term to gain performance and scalability.

A Relational database has a typical schema design that shows number of tables and the relationship between these tables, while in MongoDB there is no concept of relationship.

MongoDB Advantages

- MongoDB is schema less. It is a document database in which one collection holds different documents.
- There may be difference between number of fields, content and size of the document from one to other.
- Structure of a single object is clear in MongoDB.
- There are no complex joins in MongoDB.
- MongoDB provides the facility of deep query because it supports a powerful dynamic query on documents.
- It is very easy to scale.
- It uses internal memory for storing working sets and this is the reason of its fast access.

Distinctive features of MongoDB

- Easy to use
- Light Weight
- Extremely faster than RDBMS

Where MongoDB should be used

- Big and complex data
- Mobile and social infrastructure
- Content management and delivery
- User data management
- Data hub

Performance analysis of MongoDB and RDBMS

- In relational database (RDBMS) tables are using as storing elements, while in MongoDB collection is used.
- In the RDBMS, we have multiple schema and in each schema we create tables to store data while, MongoDB is a document oriented database in which data is written in BSON format which is a JSON like format.
- MongoDB is almost 100 times faster than traditional database systems.

1.1.1 MongoDB Datatypes (String, Integer, Boolean, Double, Arrays, Objects)

Data Types	Description
String	String is the most commonly used datatype. It is used to store data. A string must be UTF 8 valid in mongodb.
Integer	Integer is used to store the numeric value. It can be 32 bit or 64 bit depending on the server you are using.
Boolean	This datatype is used to store boolean values. It just shows YES/NO values.
Double	Double datatype stores floating point values.
Min/Max Keys	This datatype compare a value against the lowest and highest bson elements.
Arrays	This datatype is used to store a list or multiple values into a single key.
Object	Object datatype is used for embedded documents.
Null	It is used to store null values.
Symbol	It is generally used for languages that use a specific type.
Date	This datatype stores the current date or time in unix time format. It makes you possible to specify your own date time by creating object of date and pass the value of date, month, year into it.

1.1.2 Database creation and dropping database

How to download MongoDB

You can download an appropriate version of MongoDB which your system supports, from the link "http://www.mongodb.org/downloads" to install the MongoDB on Windows. You should choose correct version of MongoDB acording to your computer's Window.

How to set up the MongoDB environment

A data directory is required in MongoDB to store all the information. Its by default data directory path is \data\db. you can create this folder by command prompt.

md\data\db

For example:

If you want to start MongoDB, run mongod.exe

You can do it from command prompt.

C:\Program Files\MongoDB\bin\mongod.exe

This will start the mongoDB database process. If you get a message "waiting for connection" in the console output, it indicates that the mongodb.exe process is running successfully.

For example:

When you connect to the MongoDB through the mongo.exe shell, you should follow these steps:

Open another command prompt.

At the time of connecting, specify the data directory if necessary.

Note: If you use the default data directory while MongoDB installation, there is no need to specify the data directory.

For example:

C:\mongodb\bin\mongo.exe

If you use the different data directory while MongoDB installation, specify the directory when connecting.

For example:

C:\mongodb\bin\mongod.exe-- dbpath d:\test\mongodb\data

If you have spaces in your path, enclose the entire path in double space.

For example:

C:\mongodb\bin\mongod.exe-- dbpath "d: \ test\mongodb\data"

How to configure directory and files

First to create a configuration file and a directory path for MongoDB log output after that create a specific directory for MongoDB log files.

Data Modeling in MongoDB

In MongoDB, data has a flexible schema. It is totally different from SQL database where you had to determine and declare a table's schema before inserting data. MongoDB collections do not enforce document structure.

The main challenge in data modeling is balancing the need of the application, the performance characteristics of the database engine, and the data retrieval patterns.

Consider the following things while designing the schema in MongoDB

- Always design schema according to user requirements.
- Do join on write operations not on read operations.
- Objects which you want to use together, should be combined into one document. Otherwise they should be separated (make sure that there should not be need of joins).
- Optimize your schema for more frequent use cases.
- Do complex aggregation in the schema.
- You should duplicate the data but in a limit, because disc space is cheaper than compute time.

For example:

let us take an example of a client who needs a database design for his website. His website has the following requirements:

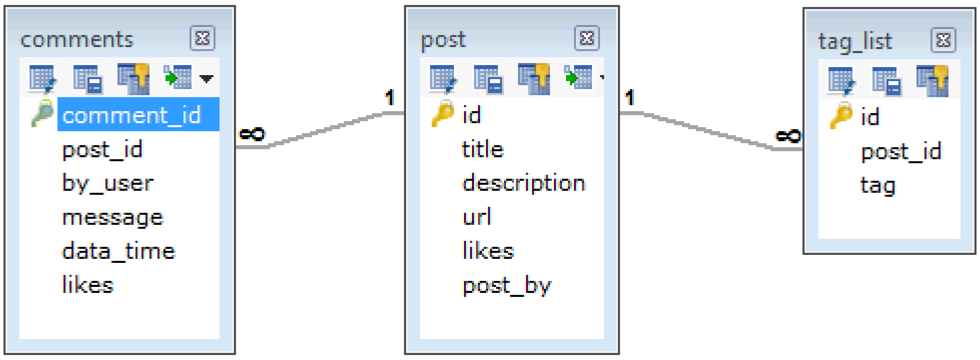
Every post is distinct (contains unique title, description and url).

Every post can have one or more tags.

Every post has the name of its publisher and total number of likes.

Each post can have zero or more comments and the comments must contain user name, message, data-time and likes.

For the above requirement, a minimum of three tables are required in RDBMS.



But in MongoDB, schema design will have one collection post and has the following structure:

```
{
_id: POST_ID
title: TITLE_OF_POST,
description: POST_DESCRIPTION,
by: POST_BY,
url: URL_OF_POST,
tags: [TAG1, TAG2, TAG3],
likes: TOTAL_LIKES,
comments: [
{
user: 'COMMENT_BY',
message: TEXT,
datecreated: DATE_TIME,
like: LIKES
},
{
user: 'COMMENT_BY',
message: TEST,
dateCreated: DATE_TIME,
like: LIKES
}}}
```

MongoDB Create Database

Use Database method:

There is no create database command in MongoDB. Actually, MongoDB do not provide any command to create database.

It may be look like a weird concept, if you are from traditional SQL background where you need to create a database, table and insert values in the table manually.

Here, in MongoDB you don't need to create a database manually because MongoDB will create it automatically when you save the value into the defined collection at first time.

You also don't need to mention what you want to create, it will be automatically created at the time you save the value into the defined collection.

Here one thing is very remarkable that you can create collection manually by "db.createCollection()" but not the database.

How and when to create database

If there is no existing database, the following command is used to create a new database.

Syntax:

```
1. use DATABASE_NAME
```

If the database already exists, it will return the existing database.

Let' take an example to demonstrate how a database is created in MongoDB. In the following example, we are going to create a database "javatpointdb".

See this example

```
1. >use javatpointdb
   Swithched to db javatpointdb
```

To **check the currently selected database**, use the command db:

```
1. >db
   javatpointdb
```

To **check the database list**, use the command show dbs:

```
1. >show dbs
   local 0.078GB
```

Here, your created database "javatpointdb" is not present in the list, **insert at least one document** into it to display database:

```
1. >db.movie.insert({"name":"javatpoint"})
   WriteResult({ "nInserted": 1})
1. >show dbs
   javatpointdb 0.078GB
   local 0.078GB
```

MongoDB Drop Database

The dropDatabase command is used to drop a database. It also deletes the associated data files. It operates on the current database.

Syntax:

1. db.dropDatabase()

This syntax will delete the selected database. In the case you have not selected any database, it will delete default "test" database.

To **check the database list**, use the command show dbs:

1. >show dbs

```
javatpointdb 0.078GB

local 0.078GB
```

If you want to **delete the database "javatpointdb"**, use the dropDatabase() command as follows:

1. >use javatpointdb

```
switched to the db javatpointdb
```

1. >db.dropDatabase()

```
{ "dropped": "javatpointdb", "ok": 1}
```

Now check the list of databases:

1. >show dbs

```
local 0.078GB
```

1.2 create and Drop collections

MongoDB Create Collection

In MongoDB, db.createCollection(name, options) is used to create collection. But usually you don?t need to create collection. MongoDB creates collection automatically when you insert some documents. It will be explained later. First see how to create collection:

Syntax:

1. db.createCollection(**name**, options)

Here,

Name: is a string type, specifies the name of the collection to be created.

Options: is a document type, specifies the memory size and indexing of the collection. It is an optional parameter.

Following is the list of options that can be used.

Field	Type	Description
Capped	Boolean	(Optional) If it is set to true, enables a capped collection. Capped collection is a fixed size collecction that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.
AutoIndexID	Boolean	(Optional) If it is set to true, automatically create index on ID field. Its default value is false.
Size	Number	(Optional) It specifies a maximum size in bytes for a capped collection. Ifcapped is true, then you need to specify this field also.
Max	Number	(Optional) It specifies the maximum number of documents allowed in the capped collection.

Let's take an **example to create collection**. In this example, we are going to create a collection name SSSIT.

1. >use test

switched to db test

1. >db.createCollection("SSSIT")

{ "ok" : 1 }

To **check the created collection**, use the command "show collections".

1. >show collections

SSSIT

How does MongoDB create collection automatically

MongoDB creates collections automatically when you insert some documents. For example: Insert a document named seomount into a collection named SSSIT. The operation will create the collection if the collection does not currently exist.

1. >db.SSSIT.**insert**({"name" : "seomount"})
2. >show collections
3. SSSIT

If you want to see the inserted document, use the find() command.

Syntax:

db.collection_name.find()

MongoDB Drop collection

In MongoDB, db.collection.drop() method is used to drop a collection from a database. It completely removes a collection from the database and does not leave any indexes associated with the dropped collections.

The db.collection.drop() method does not take any argument and produce an error when it is called with an argument. This method removes all the indexes associated with the dropped collection.

Syntax:

1. db.COLLECTION_NAME.**drop**()

MongoDB Drop collection example

Let's take an example to drop collection in MongoDB.

First **check the already existing collections** in your database.

1. >use mydb

Switched to db mydb

1. > show collections

SSSIT

system.indexes

Note: Here we have a collection named SSSIT in our database.

Now **drop the collection** with the name SSSIT:

1. >db.SSSIT.**drop**()

True

Now **check the collections** in the database:

1. >show collections

System.indexes

Now, there are no existing collections in your database.

Note: The drop command returns true if it successfully drops a collection. It returns false when there is no existing collection to drop.

1.3 CRUD operations (Insert, update, delete, find, Query and Projection operators)

MongoDB insert documents

In MongoDB, the **db.collection.insert()** method is used to add or insert new documents into a collection in your database.

Upsert

There are also two methods "db.collection.update()" method and "db.collection.save()" method used for the same purpose. These methods add new documents through an operation called upsert.

Upsert is an operation that performs either an update of existing document or an insert of new document if the document to modify does not exist.

Syntax

```
>db.COLLECTION_NAME.insert(document)
```

Let?s take an example to demonstrate how to insert a document into a collection. In this example we insert a document into a collection named javatpoint. This operation will automatically create a collection if the collection does not currently exist.

Example

```
db.javatpoint.insert(
   {
     course: "java",
     details: {
        duration: "6 months",
        Trainer: "Sonoo jaiswal"
     },
     Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
     category: "Programming language"
          }
       )
```

After the successful insertion of the document, the operation will return a WriteResult object with its status.

Output:

```
WriteResult({ "nInserted" : 1 })
```

Here the **nInserted** field specifies the number of documents inserted. If an error is occurred then the **WriteResult** will specify the error information.

Check the inserted documents

If the insertion is successful, you can view the inserted document by the following query.

```
>db.javatpoint.find()
```

You will get the inserted document in return.

Output:

```
{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "java", "details" :

{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :

[ {"size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],

 "category" : "Programming language" }
```

Note: Here, the ObjectId value is generated by MongoDB itself. It may differ from the one shown.

MongoDB insert multiple documents

If you want to insert multiple documents in a collection, you have to pass an array of documents to the db.collection.insert() method.

Create an array of documents

Define a variable named Allcourses that hold an array of documents to insert.

```
var Allcourses =
    [
      {
        Course: "Java",
        details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
        Batch: [ { size: "Medium", qty: 25 } ],
        category: "Programming Language"
      },
      {
                Course: ".Net",
                details: { Duration: "6 months", Trainer: "Prashant Verma" },
                Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 }, ],
                category: "Programming Language"
              },
              {
                Course: "Web Designing",
                details: { Duration: "3 months", Trainer: "Rashmi Desai" },
                Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
                category: "Programming Language"
              }
            ];
```

Inserts the documents

Pass this Allcourses array to the db.collection.insert() method to perform a bulk insert.

```
> db.javatpoint.insert( Allcourses );
```

After the successful insertion of the documents, this will return a BulkWriteResult object with the status.

```
BulkWriteResult({
  "writeErrors" : [ ],
  "writeConcernErrors" : [ ],
  "nInserted" : 3,
  "nUpserted" : 0,
  "nMatched" : 0,
  "nModified" : 0,
  "nRemoved" : 0,
  "upserted" : [ ]
})
```

Note: Here the nInserted field specifies the number of documents inserted. In the case of any error during the operation, the **BulkWriteResult** will specify that error.

You can check the inserted documents by using the following query:

```
>db.javatpoint.find()
```

Insert multiple documents with Bulk

In its latest version of MongoDB (MongoDB 2.6) provides a Bulk() API that can be used to perform multiple write operations in bulk.

You should follow these steps to insert a group of documents into a MongoDB collection.

Initialize a bulk operation builder

First initialize a bulk operation builder for the collection javatpoint.

```
var bulk = db.javatpoint.initializeUnorderedBulkOp();
```

This operation returns an unorder operations builder which maintains a list of operations to perform .

Add insert operations to the bulk object

```
bulk.insert(
   {
     course: "java",
     details: {
        duration: "6 months",
        Trainer: "Sonoo jaiswal"
     },
     Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
     category: "Programming language"
          }
        );
```

Execute the bulk operation

Call the execute() method on the bulk object to execute the operations in the list.

```
bulk.execute();
```

After the successful insertion of the documents, this method will return a **BulkWriteResult** object with its status.

BulkWriteResult({

```
  "writeErrors" : [ ],

  "writeConcernErrors" : [ ],

  "nInserted" : 1,

  "nUpserted" : 0,

  "nMatched" : 0,

  "nModified" : 0,

  "nRemoved" : 0,

  "upserted" : [ ]

})
```

Here the nInserted field specifies the number of documents inserted. In the case of any error during the operation, the **BulkWriteResult** will specify that error.

MongoDB update documents

In MongoDB, update() method is used to update or modify the existing documents of a collection.

Syntax:

1. db.COLLECTION_NAME.**update**(SELECTIOIN_CRITERIA, UPDATED_DATA)

Example

Consider an example which has a collection name javatpoint. Insert the following documents in collection:

```
db.javatpoint.insert(
   {
     course: "java",
     details: {
        duration: "6 months",
        Trainer: "Sonoo jaiswal"
     },
     Batch: [ { size: "Small", qty: 15 }, { size: "Medium", qty: 25 } ],
9.      category: "Programming language"
10.           }
11.       )
```

After successful insertion, check the documents by following query:

```
1. >db.javatpoint.find()
```

Output:

```
{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "java", "details" :
{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :
[ {"size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],
 "category" : "Programming language" }
```

Update the existing course "java" into "android":

1. >db.javatpoint.**update**({'course':'java'},{$**set**:{'course':'android'}})

Check the updated document in the collection:

```
1. >db.javatpoint.find()
```

Output:

```
{ "_id" : ObjectId("56482d3e27e53d2dbc93cef8"), "course" : "android", "details" :
{ "duration" : "6 months", "Trainer" : "Sonoo jaiswal" }, "Batch" :
[ {"size" : "Small", "qty" : 15 }, { "size" : "Medium", "qty" : 25 } ],
 "category" : "Programming language" }
```

MongoDB Delete documents

In MongoDB, the db.colloction.remove() method is used to delete documents from a collection. The remove() method works on two parameters.

1. Deletion criteria: With the use of its syntax you can remove the documents from the collection.

2. JustOne: It removes only one document when set to true or 1.

Syntax:

```
1. db.collection_name.remove (DELETION_CRITERIA)
```

Remove all documents

If you want to remove all documents from a collection, pass an empty query document {} to the remove() method. The remove() method does not remove the indexes.

Let's take an example to demonstrate the remove() method. In this example, we remove all documents from the "javatpoint" collection.

1. db.javatpoint.remove({})

Remove all documents that match a condition

If you want to remove a document that match a specific condition, call the remove() method with the <query> parameter.

The following example will remove all documents from the javatpoint collection where the type field is equal to programming language.

1. db.javatpoint.remove({ type : "programming language" })

Remove a single document that match a condition

If you want to remove a single document that match a specific condition, call the remove() method with justOne parameter set to true or 1.

The following example will remove a single document from the javatpoint collection where the type field is equal to programming language.

1. db.javatpoint.remove({ type : "programming language" }, 1)

MongoDB Query documents

In MongoDB, the **db.collection.find()** method is used to retrieve documents from a collection. This method returns a cursor to the retrieved documents.

The db.collection.find() method reads operations in mongoDB shell and retrieves documents containing all their fields.

Note: You can also restrict the fields to return in the retrieved documents by using some specific queries. For example: you can use the db.collection.findOne() method to return a single document. It works same as the db.collection.find() method with a limit of 1.

Syntax:

1. db.COLLECTION_NAME.find({})

Select all documents in a collection:

To retrieve all documents from a collection, put the query document ({}) empty. It will be like this:

1. db.COLLECTION_NAME.find()

For example: If you have a collection name "canteen" in your database which has some fields like foods, snacks, beverages, price etc. then you should use the following query to select all documents in the collection "canteen".

1. db.canteen.find()

1.4 Operators (Projection, update, limit (), sort ()) and Aggregation commands

MongoDB Query and Projection Operator

The MongoDB query operator includes comparison, logical, element, evaluation, Geospatial, array, bitwise, and comment operators.

MongoDB Comparison Operators

$eq

The $eq specifies the equality condition. It matches documents where the value of a field equals the specified value.

Syntax:

1. { <field> : { $eq: <value> } }

Example:

1. db.books.find ({ price: { $eq: 300 } })

The above example queries the books collection to select all documents where the value of the price filed equals 300.

$gt

The $gt chooses a document where the value of the field is greater than the specified value.

Syntax:

1. { field: { $gt: value } }

Example:

1. db.books.find ({ price: { $gt: 200 } })

$gte

The $gte choose the documents where the field value is greater than or equal to a specified value.

Syntax:

1. { field: { $gte: value } }

Example:

1. db.books.find ({ price: { $gte: 250 } })

$in

The $in operator choose the documents where the value of a field equals any value in the specified array.

Syntax:

1. { filed: { $in: [<value1>, <value2>, ……] } }

Example:

1. db.books.find({ price: { $in: [100, 200] } })

$lt

The $lt operator chooses the documents where the value of the field is less than the specified value.

Syntax:

1. { field: { $lt: value } }

Example:

1. db.books.find ({ price: { $lt: 20 } })

$lte

The $lte operator chooses the documents where the field value is less than or equal to a specified value.

Syntax:

1. { field: { $lte: value } }

Example:

1. db.books.find ({ price: { $lte: 250 } })

$ne

The $ne operator chooses the documents where the field value is not equal to the specified value.

Syntax:

1. { <field>: { $ne: <value> } }

Example:

1. db.books.find ({ price: { $ne: 500 } })

$nin

The $nin operator chooses the documents where the field value is not in the specified array or does not exist.

Syntax:

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

Example:

1. db.books.find ({ price: { $nin: [50, 150, 200] } })

MongoDB Logical Operator

$and

The $and operator works as a logical AND operation on an array. The array should be of one or more expressions and chooses the documents that satisfy all the expressions in the array.

Syntax:

1. { $and: [{ <exp1> }, { <exp2> }, ...]}

Example:

1. db.books.find ({ $and: [{ price: { $ne: 500 } }, { price: { $exists: **true** } }] })

$not

The $not operator works as a logical NOT on the specified expression and chooses the documents that are not related to the expression.

Syntax:

1. { field: { $not: { <operator-expression> } } }

Example:

1. db.books.find ({ price: { $not: { $gt: 200 } } })

$nor

The $nor operator works as logical NOR on an array of one or more query expression and chooses the documents that fail all the query expression in the array.

Syntax:

1. { $nor: [{ <expression1> } , { <expresion2> } , ...] }

Example:

1. db.books.find ({ $nor: [{ price: 200 }, { sale: **true** }] })

$or

It works as a logical OR operation on an array of two or more expressions and chooses documents that meet the expectation at least one of the expressions.

Syntax:

1. { $or: [{ <exp_1> }, { <exp_2> }, ... , { <exp_n> }] }

Example:

1. db.books.find ({ $or: [{ quantity: { $lt: 200 } }, { price: 500 }] })

MongoDB Element Operator

$exists

The exists operator matches the documents that contain the field when Boolean is true. It also matches the document where the field value is null.

Syntax:

1. { field: { $exists: <boolean> } }

Example:

1. db.books.find ({ qty: { $exists: **true**, $nin: [5, 15] } })

$type

The type operator chooses documents where the value of the field is an instance of the specified BSON type.

Syntax:

1. { field: { $type: <BSON type> } }

Example:

1. db.books.find ({ "bookid" : { $type : 2 } });

MongoDB Evaluation Operator

$expr

The expr operator allows the use of aggregation expressions within the query language.

Syntax:

1. { $expr: { <expression> } }

Example:

1. db.store.find({ $expr: {$gt: ["$product" , "price"] } })

$jsonSchema

It matches the documents that satisfy the specified JSON Schema.

Syntax:

1. { $jsonSchema: <JSON **schema** object> }

$mod

The mod operator selects the document where the value of a field is divided by a divisor has the specified remainder.

Syntax:

1. { field: { $mod: [divisor, remainder] } }

Example:

1. db.books.find ({ qty: { $mod: [200, 0] } })

$regex

It provides regular expression abilities for pattern matching strings in queries. The MongoDB uses regular expressions that are compatible with Perl.

Syntax:

1. { <field>: /pattern/<options> }

Example:

1. db.books.find({ price: { $regex: /789$/ } })

$text

The $text operator searches a text on the content of the field, indexed with a text index.

Syntax:

```
1.            {
2.  $text:
3.    {
4.      $search: <string>,
5.      $language: <string>,
6.      $caseSensitive: <boolean>,
7.      $diacriticSensitive: <boolean>
```

```
    }
}
```

Example:

```
db.books.find( { $text: { $search: "Othelo" } } )
```

$where

The "where" operator is used for passing either a string containing a JavaScript expression or a full JavaScript function to the query system.

Example:

```
            db.books.find( { $where: function() {
  return (hex_md5(this.name)== "9b53e667f30cd329dca1ec9e6a8")
} } );
```

MongoDB Geospatial Operator

$geoIntersects

It selects only those documents whose geospatial data intersects with the given GeoJSON object.

Syntax:

```
{
  <location field>: {
     $geoIntersects: {
        $geometry: {
           type: "<object type>" ,
           coordinates: [ <coordinates> ]
        }
     }
  }
        }
```

Example:

```
          db.places.find(
{
  loc: {
    $geoIntersects: {
       $geometry: {
          type: "Triangle" ,
          coordinates: [
            [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ] ]
          ]
               }
             }
           }
         }
```

$geoWithin

The geoWithin operator chooses the document with geospatial data that exists entirely within a specified shape.

Syntax:

```
          {
<location field>: {
   $geoWithin: {
      $geometry: {
         type: <"Triangle" or "Rectangle"> ,
         coordinates: [ <coordinates> ]
      }
   }
}
```

$near

The near operator defines a point for which a geospatial query returns the documents from close to far.

Syntax:

```
{
<location field>: {
  $near: {
    $geometry: {
       type: "Point" ,
       coordinates: [ <longitude> , <latitude> ]
    },
    $maxDistance: <distance in meters>,
    $minDistance: <distance in meters>
  }
}
```

Example:

```
db.places.find(
{
  location:
    { $near :
       {
         $geometry: { type: "Point",  coordinates: [ -73.9667, 40.78 ] },
         $minDistance: 1000,
         $maxDistance: 5000
       }
    }
}
```

$nearSphere

The nearsphere operator specifies a point for which the geospatial query returns the document from nearest to farthest.

Syntax:

```
{
  $nearSphere: [ <x>, <y> ],
```

3. $minDistance: <distance in radians>,

4. $maxDistance: <distance in radians>

5. }

Example:

1. db.legacyPlaces.find(

2. { location : { $nearSphere : [-73.9667, 40.78], $maxDistance: 0.10 } }

3.)

$all

It chooses the document where the value of a field is an array that contains all the specified elements.

Syntax:

1. { <field>: { $all: [<value1> , <value2> ...] } }

Example:

1. db.books.find({ tags: { $all: ["Java", "MongoDB", "RDBMS"] } })

$elemMatch

The operator relates documents that contain an array field with at least one element that matches with all the given query criteria.

Syntax:

1. { <field>: { $elemMatch: { <query1>, <query2>, ... } } }

Example:

1. db.books.find(

2. { results: { $elemMatch: { $gte: 500, $lt: 400 } } }

3.)

$size

It selects any array with the number of the element specified by the argument.

Syntax:

1. db.collection.find({ field: { $**size**: 2 } });

MongoDB Bitwise Operator

$bitsAllClear

It matches the documents where all the bit positions given by the query are clear infield.

Syntax:

1. { <field>: { $bitsAllClear: <**numeric** bitmask> } }

Example:

1. db.inventory.find({ a: { $bitsAllClear: [1, 5] } })

$bitsAllSet

The bitallset operator matches the documents where all the bit positions given by the query are set in the field.

Syntax:

1. { <field>: { $bitsAllSet: <**numeric** bitmask> } }

Example:

1. db.inventory.find({ a: { $bitsAllClear: [1, 5] } })

$bitsAnyClear

The bitAnyClear operator matches the document where any bit of positions given by the query is clear in the field.

Syntax:

1. { <field>: { $bitsAnyClear: <**numeric** bitmask> } }

Example:

1. db.inventory.find({ a: { $bitsAnyClear: [5, 10] } })

$bitsAnySet

It matches the document where any of the bit positions given by the query are set in the field.

Syntax:

1. { <field>: { $bitsAnySet: <**numeric** bitmask> } }

Example:

1. db.inventory.find({ a: { $bitsAnySet: [1, 5] } })

MongoDB comments operator

$comment

The $comment operator associates a comment to any expression taking a query predicate.

Syntax:

```
1. db.inventory.find( { <query>, $comment: <comment> } )
```

Example:

```
1.       db.inventory.find(
2. {
3.   x: { $mod: [ 1, 0 ] },
4.   $comment: "Find Odd values."
5. }
```

MongoDB Projection Operator

$

The $ operator limits the contents of an array from the query results to contain only the first element matching the query document.

Syntax:

```
1. db.books.find( { <array>: <value> ... },
2.     { "<array>.$": 1 } )
3. db.books.find( { <array.field>: <value> ...},
4.     { "<array>.$": 1 } )
```

$elemMatch

The content of the array field made limited using this operator from the query result to contain only the first element matching the element $elemMatch condition.

Syntax:

```
1. db.library.find( { bookcode: "63109" },
2. { students: { $elemMatch: { roll: 102 } } } )
```

$meta

The meta operator returns the result for each matching document where the metadata associated with the query.

Syntax:

1. { $meta: <metaDataKeyword> }

Example:

1. db.books.find(
2. <query>,
3. { score: { $meta: "textScore" } }

$slice

It controls the number of values in an array that a query returns.

Syntax:

1. db.books.find({ field: value }, { array: {$slice: count } });

Example:

1. db.books.find({}, { comments: { $slice: [200, 100] } })

MongoDB Update Operator

The following modifiers are available to update operations. For example - in db.collection.update() and db.collection.findAndModify().

Defines the operator expression in the document of the form:

```
{
   <operator1>: { <field1>: <value1>, ... },
   <operator2>: { <field2>: <value2>, ... },
}
```

Field Operator

$currentDate

It updates the elements of a field to the current date, either as a Date or a timestamp. The default data type of this operator is the date.

Syntax:

```
{ $currentDate: { <field1>: <typeSpecification1>, ... } }
```

Example:

```
            db.books.insertOne(
   { _id: 1, status: "a", lastModified: purchaseDate("2013-10-02T01:11:18.965Z") }
)
```

$inc

It increases a filed by the specified value.

Syntax:

```
{ $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }
```

Example:

```
             {
  _id: 000438,
  sku: "MongoDB",
  quantity: 1,
  metrics: {
```

6. orders: 2,

7. ratings: 3.5

8. }

9. }

$min

It changes the value of the field to a specified value if the specified value is less than the current value of the filed.

Syntax:

1. { $**min**: { <field1>: <value1>, ... } }

Example:

1. { _id: 0021, highprice: 800, lowprice: 200 }
2. db.books.**update**({ _id: 0021 }, { $**min**: { highprice: 500 } })

$max

It changes the value of the field to a specified value if the specified value is greater than the current value of the filed.

Syntax:

1. { $**max**: { <field1>: <value1>, ... } }

Example:

1. { _id: 0021, highprice: 800, lowprice: 200 }
2. db.books.**update**({ _id: 0021 }, { $**max**: { highprice: 950 } })

$mul

It multiplies the value of a field by a number.

Syntax:

1. { $mul: { <field1>: <number1>, ... } }

Example:

1. db.books.**update**(
2. { _id: 1 },
3. { $mul: { price: NumberDecimal("180.25"), qty: 2 } }
4.)

$rename

The rename operator changes the name of a field.

Syntax:

1. {$rename: { <field1>: <newName1>, <field2>: <newName2>, ... } }

Example:

1. db.books.updateMany({}, { $rename: { "nmae": "name" } })

$set

The set operator changes the value of a field with the specified value.

Syntax:

1. { $**set**: { <field1>: <value1>, ... } }

Example:

1. {
2. _id: 100,
3. sku: "abc123",
4. quantity: 50,
5. instock: **true**,
6. reorder: **false**,
7. details: { model: "14Q2", make: "xyz" },
8. tags: ["technical", "non technical"],
9. ratings: [{ **by**: "ijk", rating: 4 }]

$setOnInsert

If the upsert is set to true, then it results in an insert of a document, then setOnInsert operator assigns the specified values to the field in the document.

Syntax:

1. db.collection.**update**(
2. <query>,
3. { $setOnInsert: { <field1>: <value1>, ... } },
4. { upsert: **true** }
5.)

$unset

It removes a specified field.

Syntax:

1. { $unset: { <field1>: "", ... } }

Example:

1. db.products.**update**(
2. { sku: "unknown" },
3. { $unset: { quantity: "", instock: "" } }

Array Operators

$

We can update an element in an array without explicitly specifying the position of the element.

Syntax:

1. { "<array>.$" : value }

Example:

1. db.collection.**update**(
2. { <array>: value ... },
3. { <**update** operator>: { "<array>.$" : value } }

$[ ]

The positional operator indicates that the update operator should change all the elements in the given array field.

Syntax:

1. { <**update** operator>: { "<array>.$[]" : value } }

Example:

1. db.collection.updateMany(
2. { <query conditions> },
3. { <**update** operator>: { "<array>.$[]" : value } }

$[<identifier>]

It is called a filtered positional operator that identifies the array elements.

Syntax:

1. { <**update** operator>: { "<array>.$[<identifier>]" : value } },
2. { arrayFilters: [{ <identifier>: <condition> }] }

Example:

1. db.collection.updateMany({ <query conditions> },
2. { <**update** operator>: { "<array>.$[<identifier>]" : value } },
3. { arrayFilters: [{ <identifier>: <condition> }] })

$addToSet

It adds an element to an array unless the element is already present, in which case this operator does nothing to that array.

Syntax:

1. { $addToSet: { <field1>: <value1>, ... } }

Example:

1. db.books.**update**(
2. { _id: 1 },
3. { $addToSet: { tags: "MongoDB" } }

$pop

We can remove the first or last element of an array using the pop operator. We need to pass the value of pop as -1 to remove the first element of an array and 1 to remove the last element in an array.

Syntax:

1. { $pop: { <field>: <-1 | 1>, ... } }

Example:

1. db.books.**update**({ _id: 1 }, { $pop: { mongoDB: -1 } })

$pull

Using a pull operator, we can remove all the instances of a value in an array that matches the specified condition.

Syntax:

1. { $pull: { <field1>: <value|condition>, <field2>: <value|condition>, ... } }

Example:

```
1.                db.books.update( {}, { $pull: { Development: { $in:["Java", "RDBMS" ] }, T
   ech: "Cybersecurity" } },
2.     { multi: true }
3. )
```

$push

It appends a specified value to an array.

Syntax:

```
1. { $push: { <field1>: <value1>, ... } }
```

Example:

```
1. db.students.update( { _id: 9 }, { $push: { scores: 91 } } )
```

$pullAll

We can remove all instances of the specified value from an existing array using the pullAll operator. It removes elements that match the listed value.

Syntax:

```
1. { $pullAll: { <field1>: [ <value1>, <value2> ... ], ... } }
```

Example:

```
1. db.survey.update( { _id: 1 }, { $pullAll: { scores: [ 0, 5 ] } } )
```

Modifiers

$each

It is used with the $addToSet operator and the $push operator. It is used with the addToSet operator to add multiple values to an array if the value does not exist in the field.

Syntax:

```
1. { $addToSet: { <field>: { $each: [ <value1>, <value2> ... ] } } }
```

It is used with the push operator to append multiple values to an array.

Syntax:

```
1. { $push: { <field>: { $each: [ <value1>, <value2> ... ] } } }
```

Example:

```
db.students.update( { name: "Akki" }, { $push: { scores: { $each: [ 90, 92, 85 ] } } } )
```

$position

It specifies the location where the push operator inserts elements inside an array.

Syntax:

```
        {
$push: {
  <field>: {
     $each: [ <value1>, <value2>, ... ],
     $position: <num>
  }
}
```

Example:

```
             db.students.update(
   { _id: 1 },
   {
     $push: {
        scores: {
           $each: [ 50, 60, 70 ],
           $position: 0
        }
     }
   }
 )
```

$slice

This modifier is used to limit the number of array elements during the push operation.

Syntax:

```
         {
$push: {
   <field>: {
     $each: [ <value1>, <value2>, ... ],
     $slice: <num>
   }
}
```

Example:

```
db.students.update(
   { _id: 1 },
   {
     $push: {
       scores: {
         $each: [ 80, 78, 86 ],
         $slice: -5
       }
     }
   }
)
```

$sort

The sort modifier arranges the values of an array during the push operation.

Syntax:

```
{
$push: {
   <field>: {
     $each: [ <value1>, <value2>, ... ],
     $sort: <sort specification>
   }
}
```

Example:

```
                db.students.update(
{ _id: 1 },
{
  $push: {
    quizzes: {
      $each: [ { id: 3, score: 8 }, { id: 4, score: 7 }, { id: 5, score: 6 } ],
      $sort: { score: 1 }
    }
  }
        }
      )
```

Bitwise Operator

$bit

The bit operator updates a field using a bitwise operation. It supports bitwise AND, bitwise OR, and bitwise XOR operations.

Syntax:

```
{ $bit: { <field>: { <and|or|xor>: <int> } } }
```

Example:

```
db.books.update( { _id: 1 }, { $bit: { expdata: { and: price(100) } } }
```

MongoDB limit() Method

In MongoDB, limit() method is used to limit the fields of document that you want to show. Sometimes, you have a lot of fields in collection of your database and have to retrieve only 1 or 2. In such case, limit() method is used.

The MongoDB limit() method is used with find() method.

Syntax:

```
db.COLLECTION_NAME.find().limit(NUMBER)
```

Scenario:

Consider an example which has a collection name javatpoint.

This collection has following fields within it.

```
[
 {
   Course: "Java",
   details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
   Batch: [ { size: "Medium", qty: 25 } ],
   category: "Programming Language"
 },
 {
   Course: ".Net",
          details: { Duration: "6 months", Trainer: "Prashant Verma" },
          Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 }, ],
          category: "Programming Language"
        },
        {
          Course: "Web Designing",
          details: { Duration: "3 months", Trainer: "Rashmi Desai" },
          Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
          category: "Programming Language"
        }
```

20.];

Here, you have to display only one field by using limit() method.

Example

```
db.javatpoint.find().limit(1)
```

After the execution, you will get the following result

Output:

```
{ "_id" : ObjectId("564dbced8e2c097d15fbb601"), "Course" : "Java", "details" : {
 "Duration" : "6 months", "Trainer" : "Sonoo Jaiswal" }, "Batch" : [ { "size" :
"Medium", "qty" : 25 } ], "category" : "Programming Language" }
```

MongoDB skip() method

In MongoDB, skip() method is used to skip the document. It is used with find() and limit() methods.

Syntax

```
db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```

Scenario:

Consider here also the above discussed example. The collection javatpoint has three documents.

```
[
  {
    Course: "Java",
    details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
    Batch: [ { size: "Medium", qty: 25 } ],
    category: "Programming Language"
  },
  {
    Course: ".Net",
            details: { Duration: "6 months", Trainer: "Prashant Verma" },
            Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 }, ],
            category: "Programming Language"
          },
          {
            Course: "Web Designing",
            details: { Duration: "3 months", Trainer: "Rashmi Desai" },
            Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
            category: "Programming Language"
          }
        ];
```

Execute the following query to retrieve only one document and skip 2 documents.

Example

```
db.javatpoint.find().limit(1).skip(2)
```

After the execution, you will get the following result

Output:

```
{ "_id" : ObjectId("564dbced8e2c097d15fbb603"), "Course" : "Web Designing", "det
ails" : { "Duration" : "3 months", "Trainer" : "Rashmi Desai" }, "Batch" : [ { "
size" : "Small", "qty" : 5 }, { "size" : "Large", "qty" : 10 } ], "category" : "
Programming Language" }
```

As you can see, the skip() method has skipped first and second documents and shows only third document.

MongoDB sort() method

In MongoDB, sort() method is used to sort the documents in the collection. This method accepts a document containing list of fields along with their sorting order.

The sorting order is specified as 1 or -1.

- 1 is used for ascending order sorting.
- -1 is used for descending order sorting.

Syntax:

```
db.COLLECTION_NAME.find().sort({KEY:1})
```

Scenario

Consider an example which has a collection name javatpoint.

This collection has following fields within it.

```
[
 {
   Course: "Java",
   details: { Duration: "6 months", Trainer: "Sonoo Jaiswal" },
   Batch: [ { size: "Medium", qty: 25 } ],
   category: "Programming Language"
 },
 {
   Course: ".Net",
          details: { Duration: "6 months", Trainer: "Prashant Verma" },
          Batch: [ { size: "Small", qty: 5 }, { size: "Medium", qty: 10 }, ],
          category: "Programming Language"
        },
        {
          Course: "Web Designing",
          details: { Duration: "3 months", Trainer: "Rashmi Desai" },
          Batch: [ { size: "Small", qty: 5 }, { size: "Large", qty: 10 } ],
          category: "Programming Language"
19.          }
20.        ];
```

Execute the following query to display the documents in descending order.

```
1. db.javatpoint.find().sort({"Course":-1})
```

This will show the documents in descending order.

```
{ "_id" : ObjectId("564dbced8e2c097d15fbb603"), "Course" : "Web Designing", "det
ails" : { "Duration" : "3 months", "Trainer" : "Rashmi Desai" }, "Batch" : [ { "
size" : "Small", "qty" : 5 }, { "size" : "Large", "qty" : 10 } ], "category" : "
Programming Language" }
{ "_id" : ObjectId("564dbced8e2c097d15fbb601"), "Course" : "Java", "details" : {
 "Duration" : "6 months", "Trainer" : "Sonoo Jaiswal" }, "Batch" : [ { "size" :
"Medium", "qty" : 25 } ], "category" : "Programming Language" }
{ "_id" : ObjectId("564dbced8e2c097d15fbb602"), "Course" : ".Net", "details" : {
 "Duration" : "6 months", "Trainer" : "Prashant Verma" }, "Batch" : [ { "size" :
 "Small", "qty" : 5 }, { "size" : "Medium", "qty" : 10 } ], "category" : "Progra
mming Language" }
```

Note: By default sort() method displays the documents in ascending order. If you don't specify the sorting preference, it will display documents in ascending order.

Aggregation Commands

MongoDB aggregate command

The aggregate command does the aggregation operation using the aggregation pipeline. The aggregation pipeline allows the user to perform data processing from a record or other source using a stage-based application sequence.

Syntax:

```
{
    aggregate: "<collection>" || 1, pipeline: [ <stage>, <...>],
    explain: <boolean>, allowDiskUse: <boolean>,
    cursor: <doc>,
    maxTimeMS: <int>,
    bypassDocumentValidation: <boolean>,
    readConcern: <doc>,
    collation: <doc>,
    hint: <string or doc>,
            comment: <string>,
            writeConcern: <doc>
        }
```

Command fields:

Fields	**Type**	**Description**
aggregate	string	It contains the name of the aggregation pipeline
pipeline	array	The array that transforms the list of documents as a part of the aggregation pipeline.
explain	boolean	The explain field is optional that is used to return the information on the processing of the pipeline.
allowDiskUse	boolean	It enables the command to write to the temporary files.

cursor	document	It addresses the documents that contains the control option for the creation of the cursor object.
maxTimeMS	non-negative integer	It defines a time limit for the processing operations on a cursor.
Bypass Document Validation	boolean	It is applicable only in case if you specify the out or merge aggregation stages.
readConcern	document	It specifies the read concern. The possible read concern levels are - local, available, majority, and linearizable.
collation	document	We can specify language-specific rules for string comparison using collation. Such as - rules for case and accent marks. collation: { locale: <string>, caseLevel: <boolean>, caseFirst: <string>, strength: <int>, numericOrdering: <boolean>, alternate: <string>, maxVariable: <string>, backwards: <boolean>}
hint	string or document	It declares the index either by the index name or by the index specification document.
comment	string	We can declare an arbitrary string to help trace the operation through the database profiler, currentOp, and logs.
writeConcern	document	It is set to use the default write concern with the $out or $merge pipeline stages.

Example:

We have the following documents in the article:

```
{
   _id: ObjectId("52769ea0f3dc6ead47c9a1b2"),
   author: "Ankit",
   title: "JavaTpoint",
   tags: [ "Java Tutorial", "DBMS Tutorial", "mongodb"]
}
```

Now, we'll perform an aggregate operation on the articles collection to calculate the count of every distinct element within the tags array that appears within the collection.

```
db.runCommand( { aggregate: "articles",
   pipeline: [
      { $project: { tags: 1 } },
      { $unwind: "$tags" },
      { $group: { _id: "$tags", count: { $sum : 1 } } }
   ],
   cursor: { }
} )
```

MongoDB Count command

The MongoDB count command counts the number of documents in a collection or a view. It returns a document that contains the count and command status.

Syntax:

```
{
  count: <collection or view>,
  query: <document>,
  limit: <integer>,
  skip: <integer>,
  hint: <hint>,
```

7. readConcern: <document>,

8. collation: <document>

9. }

Command fields

Field	**Type**	**Description**
count	string	It is the name of the collection or view to count.
query	document	It is optional and used to select the document to count in the collection or view.
limit	integer	It is optional and used to limit the maximum number of matching documents to return.
skip	integer	It is optional and is used for matching documents to skip before returning results.
hint	string	It is used to define either the index name as a string or the index specification document.
readConcern document	It specifies the read concern. readConcern: { level: <value> }	
collation	document	It allows us to define language-specific rules for string comparison. Syntax: collation: { locale: <string>, caseLevel: <boolean>,

		caseFirst: <string>, strength: <int>, numericOrdering: <boolean>, alternate: <string>, maxVariable: <string>, backwards: <boolean> }

Examples:

To count the number of all the documents in the collection:

1. db.runCommand({ count: 'orders' })

MongoDB Distinct Command

This command finds the distinct values for the given field across a single collection. It returns a document that contains an array of different values. The return document contains an embedded record with query statistics and the query plan.

Syntax:

1. **distinct**: "<collection>",
2. **key**: "<field>",
3. query: <query>,
4. readConcern: <**read** concern document>,
5. collation: <collation document>
6. }

Command fields

Field	**Type**	**Description**
distinct	string	It is the name of the collection to query for different values
key	string	This is the field for which the command

		returns the distinct value.
query	document	It specifies the documents from where the distinct value will be retrieved.
readConcern document	It specifies the read concern. readConcern: { level: <value> }	

collation	document	It allows us to define language-specific rules for string comparison.
		Syntax:
		collation: {
		locale: <string>,
		caseLevel: <boolean>,
		caseFirst: <string>,
		strength: <int>,
		numericOrdering: <boolean>,
		alternate: <string>,
		maxVariable: <string>,
		backwards: <boolean>
		}

Example:

The following example returns different values for the field books from all documents in the library collection:

1. db.runCommand ({ **distinct**: "library", **key**: "books" })

MongoDB MapReduce command

The MapReduce command allows us to run the map-reduce aggregation operations over a collection.

Syntax:

```
db.runCommand(
            {
              mapReduce: <collection>,
              map: <function>,
              reduce: <function>,
              finalize: <function>,
              out: <output>,
              query: <document>,
              sort: <document>,
                      limit: <number>,
                      scope: <document>,
                      jsMode: <boolean>,
                      verbose: <boolean>,
                      bypassDocumentValidation: <boolean>,
                      collation: <document>,
                      writeConcern: <document>
                    }
                  )
```

Command Fields

Field	Type	Description
MapReduce	collection	It is the name of the collection on which we want to perform the map-reduce operation.
map	function	It is a JavaScript function that associates or maps the key-value pair.
reduce	function	It is a JavaScript function that reduces to a single object all the values associated with a particular key.

out	string	It specifies where to store the output.
query	document	It specifies the selection criteria to determine the documents input to the map function.
sort	document	It sorts the input document.
limit	number	It specifies a maximum number of documents for the input into the map function.
finalize	function	It follows the reduce method to modify the output.
scope	document	It is an option and declares the global variables that are accessible on the map.
jsMode	boolean	The jsMode specifies whether to convert intermediate data into BSON format.
verbose	boolean	It specifies whether to include the timing information in the result or not.
bypass Document Validation	boolean	It enables map-reduce to bypass document validation during the operation.
collation	document	It allows us to specify language-specific rules for string comparison. collation: { locale: <string>, caseLevel: <boolean>, caseFirst: <string>, strength: <int>, numericOrdering: <boolean>, alternate: <string>, maxVariable: <string>,

		backwards: <boolean> }
writeConcern	document	It is a document that expresses the write concern to use when outputting to a collection.

Example:

```
var mapFunction = function() { ... };
var reduceFunction = function(key, values) { ... };

db.runCommand(
        {
          mapReduce: <input-collection>,
          map: mapFunction,
          reduce: reduceFunction,
          out: { merge: <output-collection> },
               query: <query>
             }
           )
```