MONGO DB

MongoDB is a cross-platform, document-oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document.

Database

Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

Collection

Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.

Document

A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

Advantages of MongoDB over RDBMS

- **Schema less** – MongoDB is a document database in which one collection holds different documents. Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Tuning.
- **Ease of scale-out** – MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.
- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Why Use MongoDB?

- **Document Oriented Storage** – Data is stored in the form of JSON style documents.
- Index on any attribute

- Replication and high availability
- Auto-Sharding
- Rich queries
- Fast in-place updates
- Professional support by MongoDB

Where to Use MongoDB?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

Data in MongoDB has a flexible schema.documents in the same collection. They do not need to have the same set of fields or structure Common fields in a collection's documents may hold different types of data.

Data Model Design

MongoDB provides two types of data models: — Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.

Embedded Data Model

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

MongoDB - Create Database

The use Command

MongoDB **use DATABASE_NAME** is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

Syntax

Basic syntax of **use DATABASE** statement is as follows −

use DATABASE_NAME

MongoDB - Drop Database

The dropDatabase() Method

MongoDB **db.dropDatabase()** command is used to drop a existing database.

Syntax

Basic syntax of **dropDatabase()** command is as follows −

db.dropDatabase()

This will delete the selected database. If you have not selected any database, then it will delete default 'test' database.

Example

MongoDB - Create Collection

The createCollection() Method

MongoDB **db.createCollection(name, options)** is used to create collection.

Syntax

Basic syntax of **createCollection()** command is as follows –

db.createCollection(name, options)

In the command, **name** is name of collection to be created. **Options** is a document and is used to specify configuration of collection.

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and indexing

Options parameter is optional, so you need to specify only the name of the collection. Following is the list of options you can use –

Field	Type	Description

capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. **If you specify true, you need to specify size parameter also.**
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. **If capped is true, then you need to specify this field also.**
max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

>use test

switched to db test

>db.createCollection("mycollection")

{ "ok" : 1 }

>

>show collections

mycollection

system.indexes

> db.createCollection("mycol", { capped : true, autoIndexID : true, size : 6142800, max : 10000 }
){

"ok" : 0,

"errmsg" : "BSON field 'create.autoIndexID' is an unknown field.",

"code" : 40415,

"codeName" : "Location40415"

}

>

MongoDB - Drop Collection

The drop() Method

MongoDB's **db.collection.drop()** is used to drop a collection from the database.

Syntax

Basic syntax of **drop()** command is as follows –

db.COLLECTION_NAME.drop()

MongoDB - Datatypes

MongoDB supports many datatypes. Some of them are –

- **String** – This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
- **Integer** – This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
- **Boolean** – This type is used to store a boolean (true/ false) value.
- **Double** – This type is used to store floating point values.
- **Min/ Max keys** – This type is used to compare a value against the lowest and highest BSON elements.
- **Arrays** – This type is used to store arrays or list or multiple values into one key.
- **Timestamp** – ctimestamp. This can be handy for recording when a document has been modified or added.
- **Object** – This datatype is used for embedded documents.
- **Null** – This type is used to store a Null value.
- **Symbol** – This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date** – This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID** – This datatype is used to store the document's ID.
- **Binary data** – This datatype is used to store binary data.
- **Code** – This datatype is used to store JavaScript code into the document.
- **Regular expression** – This datatype is used to store regular expression.

MongoDB - Insert Document

The insert() Method

To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()** method.

Syntax

The basic syntax of **insert()** command is as follows –

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

Example

```
> db.users.insert({
... _id : ObjectId("507f191e810c19729de860ea"),
... title: "MongoDB Overview",
... description: "MongoDB is no sql database",
... by: "samir",
... url: "http://www.samir.com",
... tags: ['mongodb', 'database', 'NoSQL'],
... likes: 100
... })
WriteResult({ "nInserted" : 1 })
>
```

You can also pass an array of documents into the insert() method as shown below:.

```
> db.createCollection("post")
> db.post.insert([
        {
                title: "MongoDB Overview",
                description: "MongoDB is no SQL database",
                by: "samir",
                url: "http://www.samir.com",
                tags: ["mongodb", "database", "NoSQL"],
                likes: 100
        },
```

```
        {
        title: "NoSQL Database",
        description: "NoSQL database doesn't have tables",
        by: "samir",
        url: "http://www.samir.com",
        tags: ["mongodb", "database", "NoSQL"],
        likes: 20,
        comments: [
                {
                user:"user1",
                message: "My first comment",
                dateCreated: new Date(2013,11,10,2,35),
                like: 0
                }
        ]
}
])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
```

})

>

MongoDB - Query Document

The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()** method.

Syntax

The basic syntax of **find()** method is as follows –

```
>db.COLLECTION_NAME.find()
```

The pretty() Method

To display the results in a formatted way, you can use pretty() method.

Syntax

```
>db.COLLECTION_NAME.find().pretty()
```

The findOne() method

Apart from the find() method, there is **findOne()** method, that returns only one document.

Syntax

```
>db.COLLECTIONNAME.findOne()
```

RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:{$eg;<value>}}	db.mycol.find({"by":"samir"}).pretty()	where by = 'samir'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50

Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50
Values in an array	{<key>:{$in:[<value1>, <value2>,……<valueN>]}}	db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty()	Where name matches any of the value in :["Raj", "Ram", "Raghu"]
Values not in an array	{<key>:{$nin:<value>}}	db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty()	Where name values is not in the array :["Ramu", "Raghav"] or, doesn't exist at all

AND in MongoDB

Syntax

To query documents based on the AND condition, you need to use $and keyword. Following is the basic syntax of AND –

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

Example

> db.mycol.find({$and:[{"by":"samir"},{"title": "MongoDB Overview"}]}).pretty()

OR in MongoDB

Syntax

To query documents based on the OR condition, you need to use **$or** keyword. Following is the basic syntax of **OR** –

>db.mycol.find(

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

Example

```
>db.mycol.find({$or:[{"by":"samir"},{"title": "MongoDB Overview"}]}).pretty()
```

Using AND and OR Together

he following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is 'samir''. Equivalent SQL where clause is **'where post>10 AND (by = 'samir' OR title = 'MongoDB Overview')'**

```
db.mycol.find({"post": {$gt:10}, $or: [{"by": "samir"},
  {"title": "MongoDB Overview"}]}).pretty()
```

NOR in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword. Following is the basic syntax of **NOT** −

```
>db.COLLECTION_NAME.find(
        {
                $nor: [
                {key1: value1}, {key2:value2}
                ]
        }
)
db.empDetails.find(
        {
```

```
        $nor:[
        40
        {"First_Name": "Radhika"},
        {"Last_Name": "Christopher"}
        ]
    }
).pretty()
```

NOT in MongoDB

Syntax

To query documents based on the NOT condition, you need to use $not keyword following is the basic syntax of **NOT** –

```
db.COLLECTION_NAME.find(
    {
        $NOT: [
        {key1: value1}, {key2:value2}
        ]
    }
).pretty()
```

```
> db.empDetails.find( { "Age": { $not: { $gt: "25" } } } )
```

MongoDB - Delete Document

The remove() Method

MongoDB's **remove()** method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.

- **deletion criteria** – (Optional) deletion criteria according to documents will be removed.
- **justOne** – (Optional) if set to true or 1, then remove only one document.

Syntax

Basic syntax of remove() method is as follows –

>db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)

db.mycol.remove({'title':'MongoDB Overview'})

Remove Only One

If there are multiple records and you want to delete only the first record, then set **justOne** parameter in **remove()** method.

db.COLLECTION_NAME.remove(DELETION_CRITERIA,1)

Remove All Documents

If you don't specify deletion criteria, then MongoDB will delete whole documents from the collection. **This is equivalent of SQL's truncate command.**

db.mycol.remove({})

MongoDB - Update Document

MongoDB's **update()** and **save()** methods are used to update document into a collection. The update() method updates the values in the existing document while the save() method replaces the existing document with the document passed in save() method.

MongoDB Update() Method

The update() method updates the values in the existing document.

Syntax

The basic syntax of **update()** method is as follows –

>db.COLLECTION_NAME.update(SELECTION_CRITERIA, UPDATED_DATA)

>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})

MongoDB Save() Method

The **save()** method replaces the existing document with the new document passed in the save() method.

Syntax

The basic syntax of MongoDB **save()** method is shown below –

>db.COLLECTION_NAME.save({_id:ObjectId(),NEW_DATA})

Example

```
>db.mycol.save(
  {
        "_id" : ObjectId("507f191e810c19729de860ea"),
              "title":"mongo New Topic",
        "by":"samir"
  }
)
```

MongoDB findOneAndUpdate() method

```
>db.COLLECTION_NAME.findOneAndUpdate(SELECTIOIN_CRITERIA, UPDATED_DATA)
```

```
db.empDetails.findOneAndUpdate(
        {First_Name: 'Amit'},
        { $set: { Age: '25',e_mail: 'amit_newemail@gmail.com'}}
)
```

MongoDB updateOne() method

This methods updates a single document which matches the given filter.

```
>db.COLLECTION_NAME.updateOne(<filter>, <update>)
```

```
db.empDetails.updateOne(
        {First_Name: 'Amit'},
        { $set: { Age: '25',e_mail: 'Amit_newemail@gmail.com'}}
)
```

MongoDB - Limit Records

The Limit() Method

To limit the records in MongoDB, you need to use **limit()** method. The method accepts one number type argument, which is the number of documents that you want to be displayed.

Syntax

The basic syntax of **limit()** method is as follows −

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

Example

db.mycol.find({},{"title":1,_id:0}).limit(2)

MongoDB Skip() Method

Apart from limit() method, there is one more method **skip()** which also accepts number type argument and is used to skip the number of documents.

Syntax

The basic syntax of **skip()** method is as follows −

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

Example

Following example will display only the second document.

>db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)

MongoDB - Sort Records

The sort() Method

To sort documents in MongoDB, you need to use **sort()** method. The method accepts a document containing a list of fields along with their sorting order. To specify sorting order 1 and -1 are used. 1 is used for ascending order while -1 is used for descending order.

>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})

MongoDB - Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. In SQL count(*) and with group by is an equivalent of MongoDB aggregation. The aggregate() Method

For the aggregation in MongoDB, you should use **aggregate()** method.

Syntax

Basic syntax of **aggregate()** method is as follows −

>db.COLLECTION_NAME.aggregate(AGGREGATE_OPERATION)

db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : 1}}}])

Expression	Description	Example
$sum	Sums up the defined value from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$sum : "$likes"}}}])
$avg	Calculates the average of all given values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$avg : "$likes"}}}])
$minMongoDB - Relationships	Gets the minimum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$min : "$likes"}}}])
$max	Gets the maximum of the corresponding values from all documents in the collection.	db.mycol.aggregate([{$group : {_id : "$by_user", num_tutorial : {$max : "$likes"}}}])

$push	Inserts the value to an array in the resulting document.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$push: "$url"}}}])
$addToSet	Inserts the value to an array in the resulting document but does not create duplicates.	db.mycol.aggregate([{$group : {_id : "$by_user", url : {$addToSet : "$url"}}}])
$first	Gets the first document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", first_url : {$first : "$url"}}}])

$last	Gets the last document from the source documents according to the grouping. Typically this makes only sense together with some previously applied “$sort”-stage.	db.mycol.aggregate([{$group : {_id : "$by_user", last_url : {$last : "$url"}}}])

Relationships in MongoDB represent how various documents are logically related to each other. Relationships can be modeled via **Embedded** and **Referenced** approaches. Such relationships can be either 1:1, 1:N, N:1 or N:N.

Let us consider the case of storing addresses for users. So, one user can have multiple addresses making this a 1:N relationship.

```
db.users.insert({
    {
        "_id":ObjectId("52ffc33cd85242f436000001"),
        "contact": "987654321",
        "dob": "01-01-1991",
        "name": "Tom Benzamin",
        "address": [
        {
            "building": "22 A, Indiana Apt",
```

```
                    "pincode": 123456,
                    "city": "Los Angeles",
                    "state": "California"
                },
                {
                    "building": "170 A, Acropolis Apt",
                    "pincode": 456789,
                    "city": "Chicago",
                    "state": "Illinois"
                }
                ]
        }
})
```

```
db.users.findOne({"name":"Tom Benzamin"},{"address":1})
```

Modeling Referenced Relationships

This is the approach of designing normalized relationship. In this approach, both the user and address documents will be maintained separately but the user document will contain a field that will reference the address document's **id** field.

```
{
  "_id":ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address_ids": [
    ObjectId("52ffc4a5d85242602e000000"),
    ObjectId("52ffc4a5d85242602e000001")
  ]
```

}

As shown above, the user document contains the array field **address_ids** which contains ObjectIds of corresponding addresses. Using these ObjectIds, we can query the address documents and get address details from there. With this approach, we will need two queries: first to fetch the **address_ids** fields from **user** document and second to fetch these addresses from **address** collection.

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})

>var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
```

Aggregation Pipelines

Stages:

$match,$group,$sort,

$match Example

```
db.employees.insertMany([
        {
        _id:1,
        firstName: "John",
        lastName: "King",
        gender:'male',
        email: "john.king@abc.com",
        salary: 5000,
        department: {
                "name":"HR"
        }
        },
        {
        _id:2,
        firstName: "Sachin",
```

```
lastName: "T",
gender:'male',
email: "sachin.t@abc.com",
salary: 8000,
department: {
        "name":"Finance"
}
},
{
_id:3,
firstName: "James",
lastName: "Bond",
gender:'male',
email: "jamesb@abc.com",
salary: 7500,
department: {
        "name":"Marketing"
}
},
{
_id:4,
firstName: "Rosy",
lastName: "Brown",
gender:'female',
email: "rosyb@abc.com",
salary: 5000,
```

```
department: {
        "name":"HR"
}

},
{
_id:5,
firstName: "Kapil",
lastName: "D",
gender:'male',
email: "kapil.d@abc.com",
salary: 4500,
department: {
        "name":"Finance"
}

},
{
_id:6,
firstName: "Amitabh",
lastName: "B",
gender:'male',
email: "amitabh.b@abc.com",
salary: 7000,
department: {
        "name":"Marketing"
```

```
        }
        }
])
db.employees.aggregate([ {$match:{ gender: 'female'}} ])

db.employees.aggregate([
  { $group:{ _id:'$department.name'} }
])
db.employees.aggregate([
  { $group:{ _id:'$department.name', totalEmployees: { $sum:1 } } }
}])
```

Match and Group

```
db.employees.aggregate([
  { $match:{ gender:'male'}},
  { $group:{ _id:'$department.name', totalEmployees: { $sum:1 } } }
}])
```

Example: Get Sum of Fields

```
db.employees.aggregate([
  { $match:{ gender:'male'}},
  { $group:{ _id:{ deptName:'$department.name'}, totalSalaries: { $sum:'$salary'} }
}])
```

Example: Sort Documents

```
db.employees.aggregate([
```

```
        { $match:{ gender:'male'}},
   { $sort:{ firstName:1}}
])
```

Example: Sort Groupped Data

```
db.employees.aggregate([
        { $match:{ gender:'male'}},
   { $group:{ _id:{ deptName:'$department.name'}, totalEmployees: { $sum:1} } },
        { $sort:{ deptName:1}}
])
```

```
db.post.insert([
{
   "title" : "my first post",
   "author" : "Jim",
   "likes" : 5
},
{
   "title" : "my second post",
   "author" : "Jim",
```

```
    "likes" : 2
},
{
    "title" : "hello world",
    "author" : "Joe",
    "likes" : 3
}
])

db.comment.insert([
{
    "postTitle" : "my first post",
    "comment" : "great read",
    "likes" : 3
},
{
    "postTitle" : "my second post",
    "comment" : "good info",
    "likes" : 0
},
{
    "postTitle" : "my second post",
    "comment" : "i liked this post",
    "likes" : 12
```

```
},
{
   "postTitle" : "hello world",
   "comment" : "not my favorite",
   "likes" : 8
},
{
   "postTitle" : "my last post",
   "comment" : null,
   "likes" : 0
}
])
db.post.aggregate([
   { $lookup:
      {
         from: "comment",
         localField: "title",
         foreignField: "postTitle",
         as: "comment"
      }
   }
])
```

Model Data for Atomic Operations

The recommended approach to maintain atomicity would be to keep all the related information, which is frequently updated together in a single document using **embedded documents**. This would make sure that all the updates for a single document are atomic.

Assume we have created a collection with name productDetails and inserted a documents in it as shown below –

```
db.createCollection("products")

{ "ok" : 1 }

> db.productDetails.insert(
        {
                "_id":1,
                "product_name": "Samsung S3",
                "category": "mobiles",
                "product_total": 5,
                "product_available": 3,
                "product_bought_by": [
                {
                        "customer": "john",
                        "date": "7-Jan-2014"
                },
                {
                        "customer": "mark",
                        "date": "8-Jan-2014"
                }
                ]
        }
)

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

In this document, we have embedded the information of the customer who buys the product in the **product_bought_by** field. Now, whenever a new customer buys the product, we will first check if the product is still available using **product_available** field. If available, we will reduce the value of product_available field as well as insert the new customer's embedded document in the product_bought_by field. We will use **findAndModify** command for this functionality because it searches and updates the document in the same go.

```
db.products.findAndModify({

  query:{_id:2,product_available:{$gt:0}},

  update:{

        $inc:{product_available:-1},

    $push:{product_bought_by:{customer:"rob",date:"9-Jan-2014"}}

  }

})
```