MongoDB - Environment

32-bit versions of MongoDB only support databases smaller than **2GB** and suitable only for testing and evaluation purposes.

MongoDB requires a data folder to store its files. The default location for the MongoDB data directory is c:\data\db. So you need to create this folder using the Command Prompt. Execute the following command sequence.

C:\>md data
C:\md data\db

In the command prompt, navigate to the bin directory present in the MongoDB installation folder. Suppose my installation folder is D:\set up\mongodb

```
C:\Users\XYZ>d:
D:\>cd "set up"
D:\set up>cd mongodb
D:\set up\mongodb>cd bin
D:\set up\mongodb\bin>mongod.exe --dbpath "d:\set up\mongodb\data"
```

This will show waiting for connections message on the console output, which indicates that the mongod.exe process is running successfully.

Now to run the MongoDB, you need to open another command prompt and issue the following command.

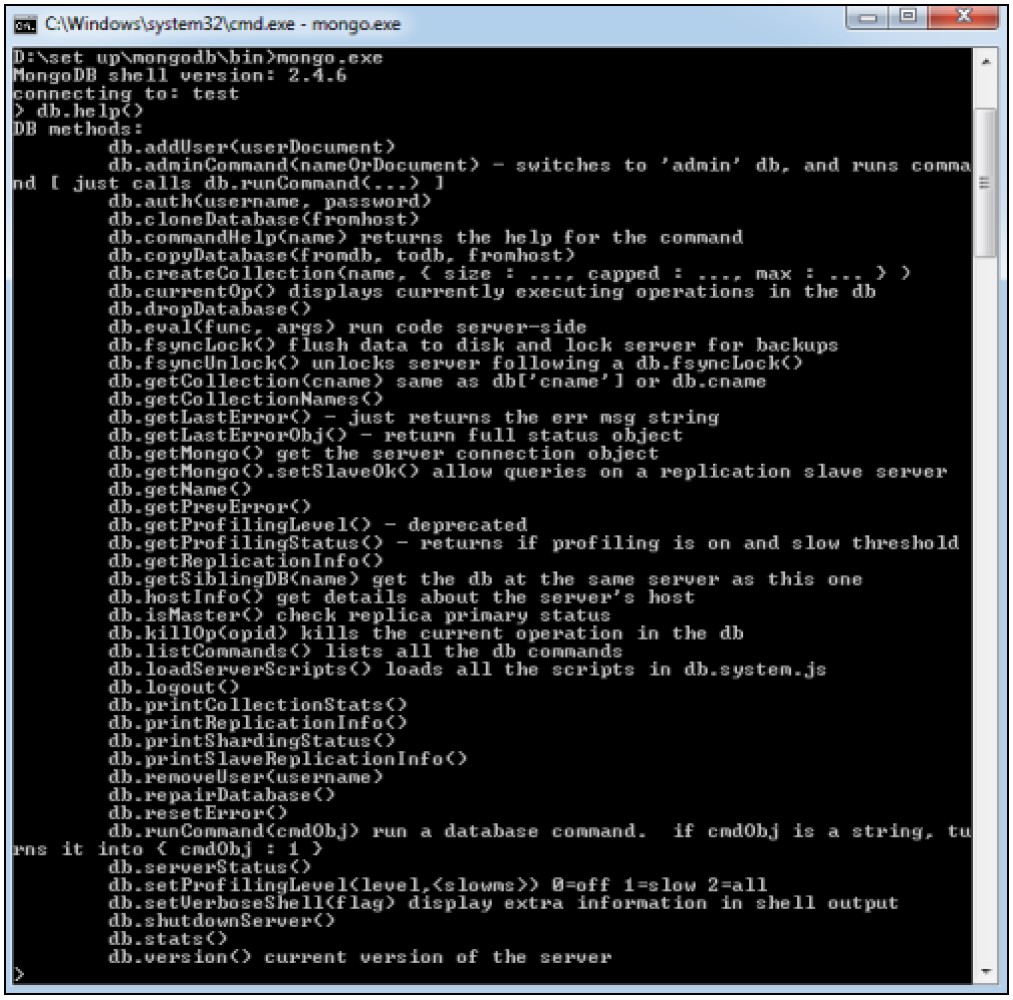
```
D:\set up\mongodb\bin>mongo.exe
MongoDB shell version: 2.4.6
connecting to: test
>db.test.save( { a: 1 } )
>db.test.find()
{ "_id" : ObjectId(5879b0f65a56a454), "a" : 1 }
>
```

This will show that MongoDB is installed and run successfully. Next time when you run MongoDB, you need to issue only commands.

D:\set up\mongodb\bin>mongod.exe --dbpath "d:\set up\mongodb\data"
D:\set up\mongodb\bin>mongo.exe

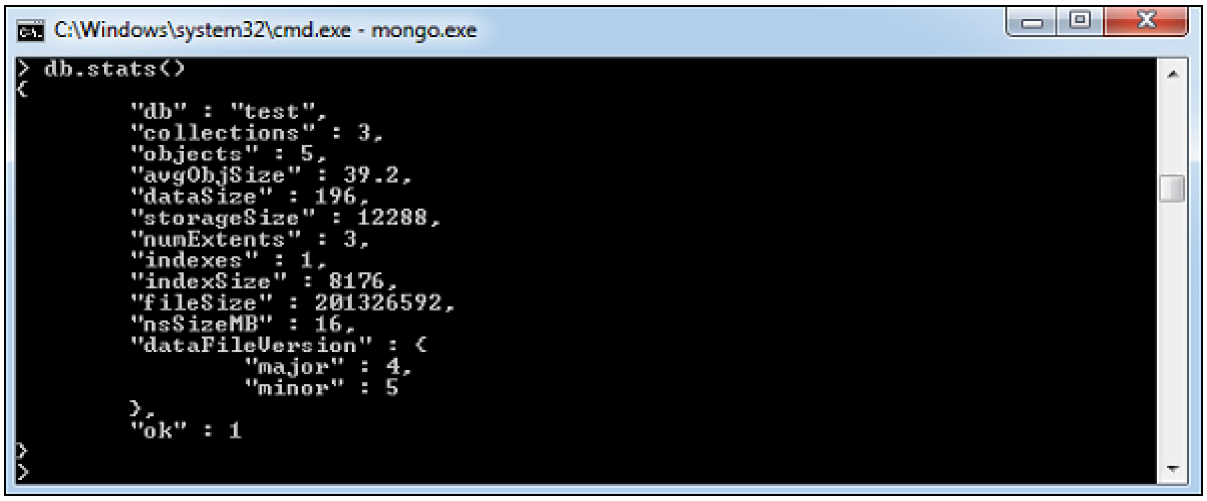
MongoDB Help

To get a list of commands, type db.help() in MongoDB client. This will give you a list of commands as shown in the following screenshot.



MongoDB Statistics

To get stats about MongoDB server, type the command db.stats() in MongoDB client. This will show the database name, number of collection and documents in the database. Output of the command is shown in the following screenshot.



MongoDB - Data Modelling

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, and common fields in a collection's documents may hold different types of data.

Some considerations while designing Schema in MongoDB

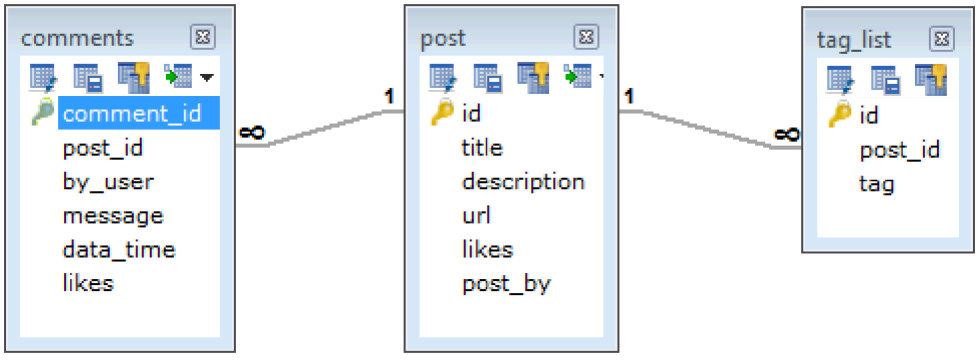
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the 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.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and 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: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    }
  ]
}
```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

MongoDB - Create Database

In this chapter, we will see how to create a database in MongoDB.

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
```

Example

If you want to use a database with name <mydb>, then use DATABASE statement would be as follows −

```
>use mydb
switched to db mydb
```

To check your currently selected database, use the command db

```
>db
mydb
```

If you want to check your databases list, use the command show dbs.

```
>show dbs
local     0.78125GB
test      0.23012GB
```

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

Your created database (mydb) is not present in list. To display database, you need to insert at least one document into it.

```
>db.movie.insert({"name":"tutorials point"})
>show dbs
local     0.78125GB
mydb      0.23012GB
test      0.23012GB
```

In MongoDB default database is test. If you didn't create any database, then collections will be stored in test database.

MongoDB - Drop Database

In this chapter, we will see how to drop a database using MongoDB command.

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

First, check the list of available databases by using the command, show dbs.

```
>show dbs
local     0.78125GB
mydb      0.23012GB
test      0.23012GB
>
```

If you want to delete new database <mydb>, then dropDatabase() command would be as follows −

```
>use mydb
switched to db mydb
>db.dropDatabase()
>{ "dropped" : "mydb", "ok" : 1 }
>
```

Now check list of databases.

```
>show dbs
local     0.78125GB
test      0.23012GB
>
```

MongoDB - Create Collection
In this chapter, we will see how to create a collection using MongoDB.

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.**
autoIndexId	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
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.

While inserting the document, MongoDB first checks size field of capped collection, then it checks max field.

Examples
Basic syntax of createCollection() method without options is as follows −

```
>use test
switched to db test
>db.createCollection("mycollection")
{ "ok" : 1 }
>
```

You can check the created collection by using the command show collections.

```
>show collections
mycollection
system.indexes
```

The following example shows the syntax of createCollection() method with few important options −

```
>db.createCollection("mycol", { capped : true, autoIndexId : true, size :
   6142800, max : 10000 } )
{ "ok" : 1 }
>
```

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

```
>db.tutorialspoint.insert({"name" : "tutorialspoint"})
>show collections
mycol
mycollection
system.indexes
tutorialspoint
>
```

MongoDB - Drop Collection

In this chapter, we will see how to drop a collection using MongoDB.

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()
```

Example

First, check the available collections into your database mydb.

```
>use mydb
switched to db mydb
>show collections
mycol
mycollection
system.indexes
tutorialspoint
>
```

Now drop the collection with the name mycollection.

```
>db.mycollection.drop()
true
>
```

Again check the list of collections into database.

```
>show collections
mycol
system.indexes
tutorialspoint
>
```

drop() method will return true, if the selected collection is dropped successfully, otherwise it will return false.

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

In this chapter, we will learn how to insert document in MongoDB collection.

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.mycol.insert({
   _id: ObjectId(7df78ad8902c),
```

```
  title: 'MongoDB Overview',
  description: 'MongoDB is no sql database',
  by: 'tutorials point',
  url: 'http://www.tutorialspoint.com',
  tags: ['mongodb', 'database', 'NoSQL'],
  likes: 100
})
```

Here mycol is our collection name, as created in the previous chapter. If the collection doesn't exist in the database, then MongoDB will create this collection and then insert a document into it.

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

_id is 12 bytes hexadecimal number unique for every document in a collection. 12 bytes are divided as follows −

_id: ObjectId(4 bytes timestamp, 3 bytes machine id, 2 bytes process id, 3 bytes incrementer)

To insert multiple documents in a single query, you can pass an array of documents in insert() command.

Example

```
>db.post.insert([
  {
    title: 'MongoDB Overview',
    description: 'MongoDB is no sql database',
    by: 'tutorials point',
    url: 'http://www.tutorialspoint.com',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 100
  },

  {
    title: 'NoSQL Database',
    description: "NoSQL database doesn't have tables",
    by: 'tutorials point',
    url: 'http://www.tutorialspoint.com',
    tags: ['mongodb', 'database', 'NoSQL'],
    likes: 20,
    comments: [
      {
        user:'user1',
        message: 'My first comment',
        dateCreated: new Date(2013,11,10,2,35),
        like: 0
      }
    ]
  }
```

])

To insert the document you can use db.post.save(document) also. If you don't specify _id in the document then save() method will work same as insert() method. If you specify _id then it will replace whole data of document containing _id as specified in save() method.

MongoDB - Query Document
In this chapter, we will learn how to query document from MongoDB collection.

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()

find() method will display all the documents in a non-structured way.

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

Syntax
>db.mycol.find().pretty()

Example
```
>db.mycol.find().pretty()
{
  "_id": ObjectId(7df78ad8902c),
  "title": "MongoDB Overview",
  "description": "MongoDB is no sql database",
  "by": "tutorials point",
  "url": "http://www.tutorialspoint.com",
  "tags": ["mongodb", "database", "NoSQL"],
  "likes": "100"
}
>
```

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

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

		te:50}}).pretty()	
Greater Than	{<key>:{$gt:<value>}}	db.my-col.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.my-col.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.my-col.find({"likes":{$ne:50}}).pretty()	where likes != 50

AND in MongoDB

Syntax

In the find() method, if you pass multiple keys by separating them by ',' then MongoDB treats it as AND condition. Following is the basic syntax of AND −

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

Example

Following example will show all the tutorials written by 'tutorials point' and whose title is 'MongoDB Overview'.

```
>db.mycol.find({$and:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty() {
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
```

For the above given example, equivalent where clause will be ' where by = 'tutorials point' AND title = 'MongoDB Overview' '. You can pass any number of key, value pairs in find clause.

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
Following example will show all the tutorials written by 'tutorials point' or whose title is 'MongoDB Overview'.

```
>db.mycol.find({$or:[{"by":"tutorials point"},{"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

Using AND and OR Together
Example
The following example will show the documents that have likes greater than 10 and whose title is either 'MongoDB Overview' or by is 'tutorials point'. Equivalent SQL where clause is 'where likes>10 AND (by = 'tutorials point' OR title = 'MongoDB Overview')'

```
>db.mycol.find({"likes": {$gt:10}, $or: [{"by": "tutorials point"},
   {"title": "MongoDB Overview"}]}).pretty()
{
   "_id": ObjectId(7df78ad8902c),
   "title": "MongoDB Overview",
   "description": "MongoDB is no sql database",
   "by": "tutorials point",
   "url": "http://www.tutorialspoint.com",
   "tags": ["mongodb", "database", "NoSQL"],
   "likes": "100"
}
>
```

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)
```

Example
Consider the mycol collection has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will set the new title 'New MongoDB Tutorial' of the documents whose title is 'MongoDB Overview'.

```
>db.mycol.update({'title':'MongoDB Overview'},{$set:{'title':'New MongoDB Tutorial'}})
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"New MongoDB Tutorial"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
>
```

By default, MongoDB will update only a single document. To update multiple documents, you need to set a parameter 'multi' to true.

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

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
Following example will replace the document with the _id '5983548781331adf45ec5'.

```
>db.mycol.save(
  {
    "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point New Topic",
     "by":"Tutorials Point"
  }
)

>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"Tutorials Point New Topic",
  "by":"Tutorials Point"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
>
```

MongoDB - Delete Document
In this chapter, we will learn how to delete a document using MongoDB.

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)
```

Example
Consider the mycol collection has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will remove all the documents whose title is 'MongoDB Overview'.

```
>db.mycol.remove({'title':'MongoDB Overview'})
>db.mycol.find()
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point 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()
>db.mycol.find()
>
```

MongoDB - Projection
In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document. If a document has 5 fields and you need to show only 3, then select only 3 fields from them.

The find() Method
MongoDB's find() method, explained in MongoDB Query Document accepts second optional parameter that is list of fields that you want to retrieve. In MongoDB, when you execute find() method,

then it displays all fields of a document. To limit this, you need to set a list of fields with value 1 or 0. 1 is used to show the field while 0 is used to hide the fields.

Syntax
The basic syntax of find() method with projection is as follows −

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

Example
Consider the collection mycol has the following data −

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will display the title of the document while querying the document.

```
>db.mycol.find({},{"title":1,_id:0})
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
{"title":"Tutorials Point Overview"}
>
```

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

MongoDB - Limit Records
In this chapter, we will learn how to limit records using MongoDB.

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
Consider the collection myycol has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will display only two documents while querying the document.

```
>db.mycol.find({},{"title":1,_id:0}).limit(2)
{"title":"MongoDB Overview"}
{"title":"NoSQL Overview"}
>
```

If you don't specify the number argument in limit() method then it will display all documents from the collection.

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)
{"title":"NoSQL Overview"}
>
```

Please note, the default value in skip() method is 0.

MongoDB - Sort Records
In this chapter, we will learn how to sort records in MongoDB.

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.

Syntax
The basic syntax of sort() method is as follows −
>db.COLLECTION_NAME.find().sort({KEY:1})

Example
Consider the collection myycol has the following data.

```
{ "_id" : ObjectId(5983548781331adf45ec5), "title":"MongoDB Overview"}
{ "_id" : ObjectId(5983548781331adf45ec6), "title":"NoSQL Overview"}
{ "_id" : ObjectId(5983548781331adf45ec7), "title":"Tutorials Point Overview"}
```

Following example will display the documents sorted by title in the descending order.

```
>db.mycol.find({},{"title":1,_id:0}).sort({"title":-1})
{"title":"Tutorials Point Overview"}
{"title":"NoSQL Overview"}
{"title":"MongoDB Overview"}
>
```

Please note, if you don't specify the sorting preference, then sort() method will display the documents in ascending order.

MongoDB - Indexing

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require MongoDB to process a large volume of data.

Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

The ensureIndex() Method

To create an index you need to use ensureIndex() method of MongoDB.

Syntax

The basic syntax of ensureIndex() method is as follows().

```
>db.COLLECTION_NAME.ensureIndex({KEY:1})
```

Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

Example

```
>db.mycol.ensureIndex({"title":1})
>
```

In ensureIndex() method you can pass multiple fields, to create index on multiple fields.

```
>db.mycol.ensureIndex({"title":1,"description":-1})
>
```

ensureIndex() method also accepts list of options (which are optional). Following is the list −

Parameter	Type	Description
background	Boolean	Builds the index in the background so that building an index does not block other database activities. Specify true to build in the background. The default value is **false**.
unique	Boolean	Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify true to create a unique index. The default value is **false**.
name	string	The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order.
dropDups	Boolean	Creates a unique index on a field that may have duplicates. MongoDB indexes only the first occurrence of a key and removes all documents from the collection that contain subsequent occurrences of that key. Specify true to create unique index. The default value is **false**.
sparse	Boolean	If true, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is **false**.
expireAfterSeconds	integer	Specifies a value, in seconds, as a TTL to control how long MongoDB retains documents in this collection.

v	index version	The index version number. The default index version depends on the version of MongoDB running when creating the index.
weights	document	The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score.
default_language	string	For a text index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. The default value is **english**.
language_override	string	For a text index, specify the name of the field in the document that contains, the language to override the default language. The default value is language.

MongoDB - Relationships

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.

Following is the sample document structure of user document −

```
{
   "_id":ObjectId("52ffc33cd85242f436000001"),
   "name": "Tom Hanks",
   "contact": "987654321",
   "dob": "01-01-1991"
}
```

Following is the sample document structure of address document −

```
{
   "_id":ObjectId("52ffc4a5d85242602e000000"),
   "building": "22 A, Indiana Apt",
   "pincode": 123456,
   "city": "Los Angeles",
   "state": "California"
}
```

Modeling Embedded Relationships

In the embedded approach, we will embed the address document inside the user document.

```
{
   "_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"
      }
   ]
}
```

This approach maintains all the related data in a single document, which makes it easy to retrieve and

maintain. The whole document can be retrieved in a single query such as −

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

Note that in the above query, db and users are the database and collection respectively.

The drawback is that if the embedded document keeps on growing too much in size, it can impact the read/write performance.

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"]}})
```

MongoDB - Database References
As seen in the last chapter of MongoDB relationships, to implement a normalized database structure in MongoDB, we use the concept of **Referenced Relationships** also referred to as **Manual References** in which we manually store the referenced document's id inside other document. However, in cases where a document contains references from different collections, we can use **MongoDB DBRefs**.

DBRefs vs Manual References
As an example scenario, where we would use DBRefs instead of manual references, consider a database where we are storing different types of addresses (**home, office, mailing**, etc.) in different collections (**address_home, address_office, address_mailing, etc**). Now, when a user collection's document references an address, it also needs to specify which collection to look into based on the address type. In such scenarios where a document references documents from many collections, we should use DBRefs.

Using DBRefs
There are three fields in DBRefs −

- **$ref** − This field specifies the collection of the referenced document
- **$id** − This field specifies the _id field of the referenced document
- **$db** − This is an optional field and contains the name of the database in which the referenced document lies

Consider a sample user document having DBRef field address as shown in the code snippet −

```
{
  "_id":ObjectId("53402597d852426020000002"),
  "address": {
  "$ref": "address_home",
  "$id": ObjectId("534009e4d852427820000002"),
  "$db": "tutorialspoint"},
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin"
}
```

The address DBRef field here specifies that the referenced address document lies in address_home collection under tutorialspoint database and has an id of 534009e4d852427820000002.

The following code dynamically looks in the collection specified by $ref parameter (address_home in our case) for a document with id as specified by $id parameter in DBRef.

```
>var user = db.users.findOne({"name":"Tom Benzamin"})
>var dbRef = user.address
>db[dbRef.$ref].findOne({"_id":(dbRef.$id)})
```

The above code returns the following address document present in address_home collection −

```
{
  "_id" : ObjectId("534009e4d852427820000002"),
  "building" : "22 A, Indiana Apt",
  "pincode" : 123456,
  "city" : "Los Angeles",
  "state" : "California"
}
```

MongoDB - Covered Queries

In this chapter, we will learn about covered queries.

What is a Covered Query?

As per the official MongoDB documentation, a covered query is a query in which −

- All the fields in the query are part of an index.
- All the fields returned in the query are in the same index.

Since all the fields present in the query are part of an index, MongoDB matches the query conditions and returns the result using the same index without actually looking inside the documents. Since indexes are present in RAM, fetching data from indexes is much faster as compared to fetching data by scanning documents.

Using Covered Queries
To test covered queries, consider the following document in the users collection −

```
{
  "_id": ObjectId("53402597d852426020000002"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "gender": "M",
  "name": "Tom Benzamin",
  "user_name": "tombenzamin"
}
```

We will first create a compound index for the users collection on the fields gender and user_name using the following query −

```
>db.users.ensureIndex({gender:1,user_name:1})
```

Now, this index will cover the following query −

```
>db.users.find({gender:"M"},{user_name:1,_id:0})
```

That is to say that for the above query, MongoDB would not go looking into database documents. Instead it would fetch the required data from indexed data which is very fast.

Since our index does not include _id field, we have explicitly excluded it from result set of our query, as MongoDB by default returns _id field in every query. So the following query would not have been covered inside the index created above −

```
>db.users.find({gender:"M"},{user_name:1})
```

Lastly, remember that an index cannot cover a query if −

- Any of the indexed fields is an array
- Any of the indexed fields is a subdocument

MongoDB - Analyzing Queries
Analyzing queries is a very important aspect of measuring how effective the database and indexing design is. We will learn about the frequently used $explain and $hint queries.

Using $explain
The $explain operator provides information on the query, indexes used in a query and other statistics. It is very useful when analyzing how well your indexes are optimized.

In the last chapter, we had already created an index for the users collection on fields gender and user_name using the following query −

```
>db.users.ensureIndex({gender:1,user_name:1})
```

We will now use $explain on the following query −

```
>db.users.find({gender:"M"},{user_name:1, _id:0}).explain()
```

The above explain() query returns the following analyzed result −

```
{
  "cursor" : "BtreeCursor gender_1_user_name_1",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 0,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 0,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : true,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    "gender" : [
      [
        "M",
        "M"
      ]
    ],
    "user_name" : [
      [
        {
          "$minElement" : 1
        },
        {
          "$maxElement" : 1
        }
      ]
    ]
  }
}
```

We will now look at the fields in this result set −

- The true value of indexOnly indicates that this query has used indexing.
- The cursor field specifies the type of cursor used. BTreeCursor type indicates that an index was used and also gives the name of the index used. BasicCursor indicates that a full scan was made without using any indexes.
- n indicates the number of documents matching returned.
- nscannedObjects indicates the total number of documents scanned.
- nscanned indicates the total number of documents or index entries scanned.

Using $hint

The $hint operator forces the query optimizer to use the specified index to run a query. This is particularly useful when you want to test performance of a query with different indexes. For example, the following query specifies the index on fields gender and user_name to be used for this query −

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,user_name:1})
```

To analyze the above query using $explain −

```
>db.users.find({gender:"M"},{user_name:1,_id:0}).hint({gender:1,user_name:1}).explain()
```

MongoDB - Atomic Operations

MongoDB does not support multi-document atomic transactions. However, it does provide atomic operations on a single document. So if a document has hundred fields the update statement will either update all the fields or none, hence maintaining atomicity at the document-level.

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.

Consider the following products document −

```
{
  "_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"
    }
  ]
}
```

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"}}
  }
})
```

Our approach of embedded document and using findAndModify query makes sure that the product purchase information is updated only if it the product is available. And the whole of this transaction being in the same query, is atomic.

In contrast to this, consider the scenario where we may have kept the product availability and the information on who has bought the product, separately. In this case, we will first check if the product is available using the first query. Then in the second query we will update the purchase information. However, it is possible that between the executions of these two queries, some other user has purchased the product and it is no more available. Without knowing this, our second query will update the purchase information based on the result of our first query. This will make the database inconsistent because we have sold a product which is not available.

MongoDB - Advanced Indexing

Consider the following document of the users collection −

```
{
  "address": {
    "city": "Los Angeles",
    "state": "California",
    "pincode": "123"
  },
  "tags": [
    "music",
    "cricket",
    "blogs"
  ],
  "name": "Tom Benzamin"
}
```

The above document contains an address sub-document and a tags array.

Indexing Array Fields

Suppose we want to search user documents based on the user's tags. For this, we will create an index on tags array in the collection.

Creating an index on array in turn creates separate index entries for each of its fields. So in our case when we create an index on tags array, separate indexes will be created for its values music, cricket and blogs.

To create an index on tags array, use the following code −

```
>db.users.ensureIndex({"tags":1})
```

After creating the index, we can search on the tags field of the collection like this −
>db.users.find({tags:"cricket"})

To verify that proper indexing is used, use the following explain command −
>db.users.find({tags:"cricket"}).explain()

The above command resulted in **"cursor" : "BtreeCursor tags_1"** which confirms that proper indexing is used.

Indexing Sub-Document Fields
Suppose that we want to search documents based on city, state and pincode fields. Since all these fields are part of address sub-document field, we will create an index on all the fields of the sub-document.

For creating an index on all the three fields of the sub-document, use the following code −
>db.users.ensureIndex({"address.city":1,"address.state":1,"address.pincode":1})

Once the index is created, we can search for any of the sub-document fields utilizing this index as follows −
>db.users.find({"address.city":"Los Angeles"})

Remember that the query expression has to follow the order of the index specified. So the index created above would support the following queries −
>db.users.find({"address.city":"Los Angeles","address.state":"California"})

It will also support the following query −
>db.users.find({"address.city":"LosAngeles","address.state":"California", "address.pincode":"123"})

MongoDB - Indexing Limitations
In this chapter, we will learn about Indexing Limitations and its other components.

Extra Overhead
Every index occupies some space as well as causes an overhead on each insert, update and delete. So if you rarely use your collection for read operations, it makes sense not to use indexes.

RAM Usage
Since indexes are stored in RAM, you should make sure that the total size of the index does not exceed the RAM limit. If the total size increases the RAM size, it will start deleting some indexes, causing performance loss.

Query Limitations
Indexing can't be used in queries which use −

- Regular expressions or negation operators like $nin, $not, etc.
- Arithmetic operators like $mod, etc.
- $where clause

Hence, it is always advisable to check the index usage for your queries.

Index Key Limits
Starting from version 2.6, MongoDB will not create an index if the value of existing index field exceeds the index key limit.

Inserting Documents Exceeding Index Key Limit
MongoDB will not insert any document into an indexed collection if the indexed field value of this document exceeds the index key limit. Same is the case with mongorestore and mongoimport utilities.

Maximum Ranges

- A collection cannot have more than 64 indexes.
- The length of the index name cannot be longer than 125 characters.
- A compound index can have maximum 31 fields indexed.