<https://www.tutorialspoint.com/mongodb/index.htm>

<https://github.com/sanjivnaik/mongodb>

Local installation using ZIP file: <https://mkyong.com/mongodb/how-to-install-mongodb-on-windows/>

**In one terminal**

C:\PERSONAL\study\mangoDB\mongodb\bin>mongod.exe --config C:\PERSONAL\study\mangoDB\mongodb\mongo.config

**Another terminal**

C:\PERSONAL\study\mangoDB\mongodb\bin>mongo

MongoDB

# *MongoDB - Overview*

**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.

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.

Relationship of RDBMS terminology with MongoDB:

**RDBMS** **MongoDB**

Database Database

Table Collection

Tuple/Row Document

column Field

Table Join Embedded Documents

Primary Key Primary Key (Default key \_id provided by MongoDB itself)

"***\_id*** " is a 12 bytes hexadecimal number which assures the uniqueness of every document. You can provide \_id while inserting the document. If you don’t provide then MongoDB provides a unique id for every document. These 12 bytes first 4 bytes for the current timestamp, next 3 bytes for machine id, next 2 bytes for process id of MongoDB server and remaining 3 bytes are simple incremental VALUE.

# *MongoDB - Advantages*

## **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

# *MongoDB - Data Modelling*

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.

For example, assume we are getting the details of **employees** in three different documents namely, Personal\_details, Contact and, Address, you can embed all the three documents in a single one as shown below –

{

\_id: ,

Emp\_ID: "10025AE336"

Personal\_details:{

First\_Name: "Radhika",

Last\_Name: "Sharma",

Date\_Of\_Birth: "1995-09-26"

},

Contact: {

e-mail: "radhika\_sharma.123@gmail.com",

phone: "9848022338"

},

Address: {

city: "Hyderabad",

Area: "Madapur",

State: "Telangana"

}

}

### **Normalized Data Model:**

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

{

\_id: <ObjectId101>,

Emp\_ID: "10025AE336"

}

Personal\_details:

{

\_id: <ObjectId102>,

empDocID: " ObjectId101",

First\_Name: "Radhika",

Last\_Name: "Sharma",

Date\_Of\_Birth: "1995-09-26"

}

Contact:

{

\_id: <ObjectId103>,

empDocID: " ObjectId101",

e-mail: "radhika\_sharma.123@gmail.com",

phone: "9848022338"

}

Address:

{

\_id: <ObjectId104>,

empDocID: " ObjectId101",

city: "Hyderabad",

Area: "Madapur",

State: "Telangana"

}

## **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.

# *MongoDB - Create Database*

“***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.

> use mydb

switched to db mydb

> db

mydb

> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

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"})

WriteResult({ "nInserted" : 1 })

> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

mydb 0.000GB

# *MongoDB - Drop Database*

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

> use mydb

switched to db mydb

> db.dropDatabase()

{ "dropped" : "mydb", "ok" : 1 }

# *MongoDB - Create Collection*

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

**name**: (type = String) – Name of the collection to be created

**options**: (type =Document) - (Optional) Specify options about memory size and indexing

**capped** : (type = 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** : (type = Boolean) - (Optional) If true, automatically create index on \_id field.s Default value is false.

**size** : (type = 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** : (type = number) - (Optional) Specifies the maximum number of documents allowed in the capped collection.

> use mydb

switched to db mydb

> db.createCollection("mycollection")

{ "ok" : 1 }

> show collections

mycollection

> 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"

}

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

> db.tutorialspoint.insert({"name" : "tutorialspoint"})

WriteResult({ "nInserted" : 1 })

> show collections

mycollection

tutorialspoint

# *MongoDB - Drop Collection*

db.*COLLECTION\_NAME*.drop()

|  |
| --- |
| > show collections  mycollection  tutorialspoint  > db.tutorialspoint.drop()  true  > show collections  Mycollection |

# *MongoDB - Datatypes*

* **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*

db.*COLLECTION\_NAME*.insert(document)

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.

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

***\_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)

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: "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  }  ]  }  ])  BulkWriteResult({  "writeErrors" : [ ],  "writeConcernErrors" : [ ],  "nInserted" : 2,  "nUpserted" : 0,  "nMatched" : 0,  "nModified" : 0,  "nRemoved" : 0,  "upserted" : [ ]  })  > |

If you need to insert only one document into a collection you can use this method.

db.*COLLECTION\_NAME*.insertOne(document)

You can insert multiple documents using the insertMany() method. To this method you need to pass an array of documents.

db.*COLLECTION\_NAME*.insertMany(array\_of\_document)

# *MongoDB - Query Document*

db.*COLLECTION\_NAME*.find()

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

db.*COLLECTION\_NAME*.find().pretty()

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

db.*COLLECTION\_NAME*.findOne()

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

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

## **OR in MongoDB:**

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

## **Using AND and OR Together:**

Eg: '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()

## **NOR in MongoDB:**

|  |
| --- |
| db.*COLLECTION\_NAME*.find(  {  $nor: [  {key1: value1}, {key2:value2}  ]  }  ) |

## **NOT in MongoDB:**

Syntax: { field: { $not: { <operator-expression> } } }

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

# *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:**

>db.COLLECTION\_NAME.update(SELECTION\_CRITERIA, UPDATED\_DATA)

## **MongoDB Save() Method:**

>db.COLLECTION\_NAME.save({\_id:ObjectId(),NEW\_DATA})

## **MongoDB findOneAndUpdate() method:**

>db.COLLECTION\_NAME.findOneAndUpdate(SELECTIOIN\_CRITERIA, UPDATED\_DATA)

## **MongoDB updateOne() method**

>db.COLLECTION\_NAME.updateOne(<filter>, <update>)

## **MongoDB updateMany() method:**

>db.COLLECTION\_NAME.updateMany(<filter>, <update>)

# *MongoDB - Delete Document*

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.

>db.COLLECTION\_NAME.remove(DELLETION\_CRITTERIA)

Ex: db.empDetails.remove({"First\_Name" : "Rachel"})

## **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.**Ex:

> db.mycol.remove({})

# *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](https://www.tutorialspoint.com/mongodb/mongodb_query_document.htm) 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.

>db.COLLECTION\_NAME.find({},{KEY:1})

Ex: > db.empDetails.find({}, {"First\_Name":1, "\_id":0}).pretty()

# *MongoDB - Limit Records*

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.

>db.COLLECTION\_NAME.find().limit(NUMBER)

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

## **Skip() Method:**

Method **skip()** which also accepts number type argument and is used to skip the number of documents.

>db.COLLECTION\_NAME.find().limit(NUMBER).skip(NUMBER)

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

# *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.COLLECTION\_NAME.find().sort({KEY:1})

# *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 createIndex() Method:**

>db.COLLECTION\_NAME.createIndex({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.

In **createIndex()** method you can pass multiple fields, to create index on multiple fields. Ex:

db.mycol.createIndex({"title":1,"description":-1})

This 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. |
| 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. |
| 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. |

## **The dropIndex() method:**

>db.COLLECTION\_NAME.dropIndex({KEY:1})

## **The dropIndexes() method:**

>db.COLLECTION\_NAME.dropIndexes()

## **The getIndexes() method:**

db.COLLECTION\_NAME.getIndexes()

# *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:**

>db.COLLECTION\_NAME.aggregate(AGGREGATE\_OPERATION)

 if you want to display a list stating how many tutorials are written by each user, then you will use the following **aggregate()** method

> db.mycol.aggregate([{$group : {\_id : "$by\_user", num\_tutorial : {$sum : 1}}}])

{ "\_id" : "tutorials point", "num\_tutorial" : 2 }

{ "\_id" : "Neo4j", "num\_tutorial" : 1 }

>

Sql equivalent query for the above use case will be **select by\_user, count(\*) from mycol group by by\_user**.

Following is a list of available aggregation expressions.

|  |  |  |
| --- | --- | --- |
| **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"}}}]) |
| $min | Gets the minimum of the corresponding values from all documents in the collection. | db.mycol.aggregate([{$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"}}}]) |

## **Pipeline Concept:**

In UNIX command, shell pipeline means the possibility to execute an operation on some input and use the output as the input for the next command and so on. MongoDB also supports same concept in aggregation framework. There is a set of possible stages and each of those is taken as a set of documents as an input and produces a resulting set of documents (or the final resulting JSON document at the end of the pipeline). This can then in turn be used for the next stage and so on.

Following are the possible stages in aggregation framework −

* **$project** − Used to select some specific fields from a collection.
* **$match** − This is a filtering operation and thus this can reduce the amount of documents that are given as input to the next stage.
* **$group** − This does the actual aggregation as discussed above.
* **$sort** − Sorts the documents.
* **$skip** − With this, it is possible to skip forward in the list of documents for a given amount of documents.
* **$limit** − This limits the amount of documents to look at, by the given number starting from the current positions.
* **$unwind** − This is used to unwind document that are using arrays. When using an array, the data is kind of pre-joined and this operation will be undone with this to have individual documents again. Thus with this stage we will increase the amount of documents for the next stage.