**MongoDB**

It is a NOSQL database written in C++. It is an open source, cross platform and document oriented database, developed and supported by company known as 10gen.

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 real time web applications and very large amount of data.

**Advantages:**

1. Scalability
2. High Performance
3. Availablity
4. Develop Faster
5. Deploy Easier
6. Scale Bigger

Example of document oriented database :

{FirstName : "John",Address: "24 Venue, South Street"}

**Database,Collection,Document:**

MongoDB stores data records as documents which are gathered together in collections. A database stores one or more collection of documents.

**JSON and BSON:**

JSON is basically known as JavaScript Object Notation.It is widely used in client side web development. By virtue of its being human and machine readable and comparatively simple to implement support for in different languages, it moved beyond web pages and is now in software development.

But it has some issues which is not ideal for storing data in database which are as follows:

1. It is a text based format and text parsing is very slow.
2. It only supports limited no of data types.

BSON is basically known as Binary JSON.It was formulated to add some additional features like adding some optional non-JSON native datatypes like dates and binary data, without which MongoDB would be missing some valuable support.It supports different types of integer data types (int vs long) or various level of decimal precision(float vs double).

MongoDB stores data both internally and over the network as BSON only but it doesn't mean we can't think MongoDB as a JSON database.

**Create Database:**

The mongoDB **use databaseName** command is used to create a database. If there is no db, then it'll create a new db else return the existing one.

**Drop Database:**

The mongodb **db.dropDatabase()** is used to drop a db.

**Create Collection:**

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

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

You can check the created collection with **show collections** command.

**Drop Collections:**

**db.collectionName.drop()** is used to drop collection.

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

**CRUD Operations:**

**Create**

1. insertOne(data,options)
2. insertMany(data,options)

**Read**

1. find(filter,options)
2. findOne(filter,options)

**Update**

1. updateOne(filter,data,options)
2. updateMany(filter,data,options)
3. replaceOne(filter,data,options)

**Delete**

1. deleteOne(filter,options)
2. deleteMany(filter,options)

**CRUD Operations Example:**

```
 db.flightData.insertOne({
...    "departureAirport": "MUC",
...    "arrivalAirport": "SFO",
...    "aircraft": "Airbus A380",
...    "distance": 12000,
...    "intercontinental": true
...  })
{
      "acknowledged" : true,
      "insertedId" : ObjectId("60a124468bfaaac5a3a2133f")
}

db.flightData.insertOne({
...    "departureAirport": "LHR",
...    "arrivalAirport": "TXL",
...    "aircraft": "Airbus A320",
...    "distance": 950,
...    "intercontinental": false
...  })
{
      "acknowledged" : true,
      "insertedId" : ObjectId("60a1245b8bfaaac5a3a21340")
}
```

```
> db.flightData.find().pretty()
{
	"_id" : ObjectId("60a124468bfaaac5a3a2133f"),
	"departureAirport" : "MUC",
	"arrivalAirport" : "SFO",
	"aircraft" : "Airbus A380",
	"distance" : 12000,
	"intercontinental" : true
}
{
	"_id" : ObjectId("60a1245b8bfaaac5a3a21340"),
	"departureAirport" : "LHR",
	"arrivalAirport" : "TXL",
	"aircraft" : "Airbus A320",
	"distance" : 950,
	"intercontinental" : false
}
> db.flightData.deleteOne({departureAirport:"TXL"})
{ "acknowledged" : true, "deletedCount" : 0 }
> db.flightData.find().pretty()
{
	"_id" : ObjectId("60a124468bfaaac5a3a2133f"),
	"departureAirport" : "MUC",
	"arrivalAirport" : "SFO",
	"aircraft" : "Airbus A380",
	"distance" : 12000,
	"intercontinental" : true
}
{
	"_id" : ObjectId("60a1245b8bfaaac5a3a21340"),
	"departureAirport" : "LHR",
	"arrivalAirport" : "TXL",
	"aircraft" : "Airbus A320",
	"distance" : 950,
	"intercontinental" : false
}
> db.flightData.deleteOne({departureAirport:"LHR"})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.flightData.find().pretty()
{
	"_id" : ObjectId("60a124468bfaaac5a3a2133f"),
	"departureAirport" : "MUC",
	"arrivalAirport" : "SFO",
	"aircraft" : "Airbus A380",
	"distance" : 12000,
	"intercontinental" : true
}
> db.flightData.updateOne({distance:12000},{$set : {marker : "delete"}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.flightData.find().pretty()
{
	"_id" : ObjectId("60a124468bfaaac5a3a2133f"),
	"departureAirport" : "MUC",
	"arrivalAirport" : "SFO",
	"aircraft" : "Airbus A380",
	"distance" : 12000,
	"intercontinental" : true,
```

```
        "marker" : "delete"
}
> db.flightData.updateMany({},{$set : {marker : "toDelete"}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.flightData.find().pretty()
{
        "_id" : ObjectId("60a124468bfaaac5a3a2133f"),
        "departureAirport" : "MUC",
        "arrivalAirport" : "SFO",
        "aircraft" : "Airbus A380",
        "distance" : 12000,
        "intercontinental" : true,
        "marker" : "toDelete"
}
> db.flightData.deleteMany({marker:"toDelete"})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.flightData.find().pretty()
>

 db.flightData.insertMany([
...   {
...     "departureAirport": "MUC",
...     "arrivalAirport": "SFO",
...     "aircraft": "Airbus A380",
...     "distance": 12000,
...     "intercontinental": true
...   },
...   {
...     "departureAirport": "LHR",
...     "arrivalAirport": "TXL",
...     "aircraft": "Airbus A320",
...     "distance": 950,
...     "intercontinental": false
...   }
... ])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("60a1274a8bfaaac5a3a21341"),
                ObjectId("60a1274a8bfaaac5a3a21342")
        ]
}
> db.flightData.find().pretty()
{
        "_id" : ObjectId("60a1274a8bfaaac5a3a21341"),
        "departureAirport" : "MUC",
        "arrivalAirport" : "SFO",
        "aircraft" : "Airbus A380",
        "distance" : 12000,
        "intercontinental" : true
}
{
        "_id" : ObjectId("60a1274a8bfaaac5a3a21342"),
        "departureAirport" : "LHR",
        "arrivalAirport" : "TXL",
        "aircraft" : "Airbus A320",
        "distance" : 950,
```

```
        "intercontinental" : false
}

db.flightData.find({intercontinental:true}).pretty()
{
        "_id" : ObjectId("60a1274a8bfaaac5a3a21341"),
        "departureAirport" : "MUC",
        "arrivalAirport" : "SFO",
        "aircraft" : "Airbus A380",
        "distance" : 12000,
        "intercontinental" : true
}

db.flightData.find({distance :{$gt:10000}}).pretty()
{
        "_id" : ObjectId("60a1274a8bfaaac5a3a21341"),
        "departureAirport" : "MUC",
        "arrivalAirport" : "SFO",
        "aircraft" : "Airbus A380",
        "distance" : 12000,
        "intercontinental" : true
}

//UpdateOne(), updateMany() and replaceOne()

db.flightData.updateOne({_id: ObjectId("60a1274a8bfaaac5a3a21341")}, {$set:{delayed: true}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.flightData.find().pretty()
{
        "_id" : ObjectId("60a1274a8bfaaac5a3a21341"),
        "departureAirport" : "MUC",
        "arrivalAirport" : "SFO",
        "aircraft" : "Airbus A380",
        "distance" : 12000,
        "intercontinental" : true,
        "delayed" : true
}
{
        "_id" : ObjectId("60a1274a8bfaaac5a3a21342"),
        "departureAirport" : "LHR",
        "arrivalAirport" : "TXL",
        "aircraft" : "Airbus A320",
        "distance" : 950,
        "intercontinental" : false
}

db.test.updateMany({foo: "bar"}, {$set: {test: "success!"}})
```

Since replaceOne() replaces the entire document - fields in the old document not contained in the new will be lost. With updateOne() new fields can be added without losing the fields in the old document.

For update, we don't need to set anything.

When the **db.collection.find ()** function is used to search for documents in the collection, the result returns a pointer to the collection of documents returned which is called a cursor.

If we want to print all the data:
Db.passengers.find().forEach((passengerData => {printJson(passengerData)}));

Projection :
In MongoDB, projection means selecting only the necessary data rather than selecting whole of the data of a document.
Db.passengers.find({},{name:1})  // _id is always included by default
1 is used to show the fields and 0 is used to hide the fields.

We use dot operator (.) to drill into a nested document.

**Relations in MongoDB:**

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.

User Document:

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

Address Document:

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

Modelling embedded relationships:

```
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"
                        }
```

```
                    ]
        }})
```

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})
```

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

Modelling referenced relationship:

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")
  ]}
```

```
>var result = db.users.findOne({"name":"Tom Benzamin"},{"address_ids":1})
>var addresses = db.address.find({"_id":{"$in":result["address_ids"]}})
```

**Limiting Records:**

To limit the records in mongodb, we need to use limit() command. It accepts one argument which is the total no of documents that you want to see.

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

Apart from limit method, there is one more method **skip**, which is used to skip the results.

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

Default value for skip is 0.

**Sorting Records:**

To sort methods in MongoDB, you need to use sort method. 1 is used for ascending order and -1 is used for descending order.

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

**Indexing :**

Indexes support efficient use fo queries. If we don't have indexes, then mongodb needs to scan all the documents. The scan is highly inefficient and requires processing of large amount of data.

**CreateIndex :**

```
>db.mycol.createIndex({"title":1}){
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1}>
```

1 is used to order in ascending or descending order.

**DropIndex :**

```
> db.mycol.dropIndex({"title":1}){
        "ok" : 0,
        "errmsg" : "can't find index with key: { title: 1.0 }",
        "code" : 27,
        "codeName" : "IndexNotFound"}
```

**Aggregation :**

Aggregation is same as SQL aggregation. It processes the data and return the computed result.