1) What is MongoDB?

MongoDB is a open-source document database.

MongoDB is NoSQL database.

MongoDB is written in C++.

MongoDB is widely used these days in huge applications.

2) Advantages of using MongoDB >

MongoDB provides very **high performance** because it allows very quick access to data because it uses internal memory for storing the working set.

MongoDB is easy **scalable**.

MongoDb is **tunable**.

MongoDb is easily available.

MongoDb can be used as a **Data Hub**.

MongoDb allows easy **indexing**

MongoDb allows **Auto-sharding**.

MongoDb can be used to store huge data.

MongoDb can be used for **user Data Management.**

3) What are **difference** in terminology in **MongoDB and RDBMS** >

MongoDB	RDBMS
database	database
Collection	Table
Document (or BSON document)	Row

Field (each document can have **different fields**)	Column (each rows **cannot** have **different columns**)
Embedded documents	Join in table
_id is the default Primary Key (which is generated automatically whenever document is inserted)	Primary Key
aggregation (e.g. group by)	aggregation pipeline

2) **Terminology** which are same in **MongoDB and RDBMS** >

MongoDB	**RDBMS**
database	database
index	index

3) **Executables** name in MongoDb vs RDBMS >

	MongoDB	MySQL	Oracle
Database **Server**	**mongod**	mysqld	oracle

Database **Client**	**mongo**	mysql	sqlplus

4) What is **Database** in MongoDB?

One MongoDB server can have multiple databases.

Each database has its own collections.

5) What are **collection** in MongoDB?

Collection in MongoDB is **same as table in RDBMS.**

Each collection has its own documents.

6) What is **Document** in MongoDb?

Document in MongoDB is **same as row in RDBMS.**

Document is set of key-value pairs in collection in MongoDB.

Each document have its own fields.

See below example of documents.

7) What is **field** in MongoDb?

Field in MongoDB is **same as column in RDBMS.**

Each document have its own fields.

Important and really interesting thing to note about document in collection is that **each document can have different fields**.

In RDBMS each table has its own rows but rows cannot have different columns.

See below example of fields.

8) Example of collection, document and fields>

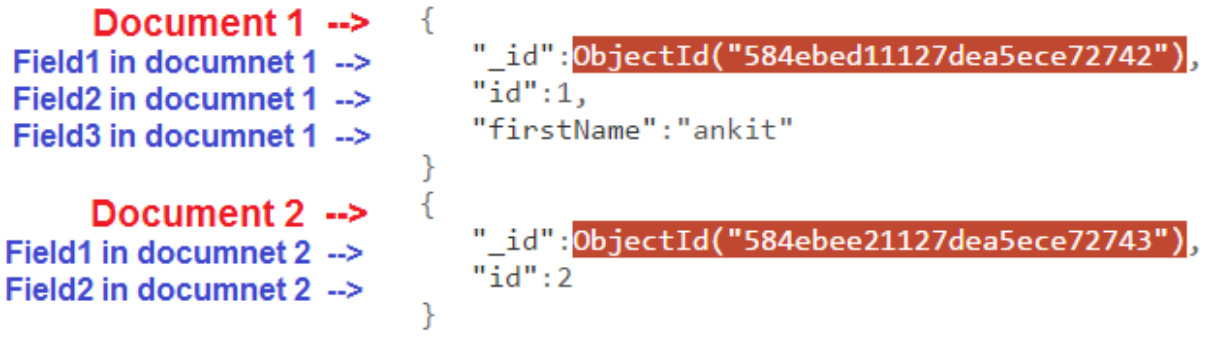
```
> db.employeeTable.insert({id : 1,  firstName:"ankit"})
> db.employeeTable.insert({id : 2})
```

Above line will create table (or collection) (if table already exists it will insert documents in it).

```
> db.employeeTable.find();
```

Now, query **collection**.

Output >



Important and really interesting thing to note about documents in collection is that **each document have different fields**.

Field _id which primary key inserted automatically.

- Document 1 has three fields.
- Document 2 has two fields.

SUMMARY>

So in this mongoDB tutorial we learned

Important **difference** in terminology in **MongoDB and RDBMS**

>

MongoDB	**RDBMS**
Collection	Table
Document	Row
Field	Column
_id is Primary Key	Primary Key

1) Create new database in mongoDB>

Once you are connected to mongoDB

```
MongoDB shell version: 3.0.6
connecting to: test     (Now, you are CONNECTED)
> use mydb
switched to db mydb
>
```

We created **new database(if not exists)** in mongoDB named **mydb**

2) See list of all databases in MongoDb>

```
> show dbs
admin   0.078GB
local   0.078GB
mydb    0.078GB
testdb  0.078GB
yourdb  0.078GB
>
```

You can see **mydb** which you have created.

3) You can switch any of above db by using command in MongoDb>

use testdb; It will switch to testdb.

```
> use testdb;
switched to db testdb
>
```

SUMMARY>

So in this mongoDB tutorial we learned how to

1) Create new database in mongoDB>

use mydb

2) See list of all databases in MongoDb>

```
show dbs
```

3) You can switch any of above db by using command in MongoDb>

use testdb; It will switch to testdb.

CRUD operations in MongoDB

CRUD STANDS FOR >

C (CREATE),

R (READ),

U (UPDATE),

D (DELETE)

1) C (CREATE) in MongoDb >

1.1) Create new database in mongoDB

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

We created **new database(if not exists)** in mongoDB named **mydb**

1.2) Create new collection(table) in mongoDB

Use createCollection() method to create collection in mongoDB>

```
> db.createCollection("employee")
{ "ok" : 1 }
>
```

Create new collection in different database in mongoDB>

```
> use testdb;
switched to db testdb
> db.createCollection("employee")
{ "ok" : 1 }
>
```

First we switched to testdb, and then created collection in it.

2) R (READ) in MongoDb >

2.1) Query (read/ display/ find/ search/ select) document (record/row) in collection (table) in mongoDB >

We will use **find** method of mongoDB.

Let's create new collection and insert document in it **before finding** >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

FIND Example > Query **all documents** of collection using find() method>

```
> db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

4 documents were found.

2.2) Display documents of collection in formatted manner.

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

FIND Example > Query **all document** of collection using find().pretty() method>

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

2.3) How to **Limit number of documents fetched** in MongoDB

1) FIND Example > Display only **first 2 documents** of collection in MongoDB >
We will use find() and **limit()** method>

```
> db.employee.find().limit(2)
```

FIND Example > Skip first document and display rest of **documents** of collection in MongoDB >

We will use find() and **skip()** method >

```
> db.employee.find().skip(1)
```

FIND Example > Display only 2nd and 3rd document of collection in MongoDB>

We will use find(), **limit()** and **skip()** method >

```
> db.employee.find().skip(1).limit(2)
```

FIND Example > Display only 2 documents of collection in MongoDB **where salary >= 1000** >

```
> db.employee.find( { salary : {$gte : 1000} }).limit(2)
```

The **$gte** operator is used to select documents where the value of the specified field is greater than or equal to a specified value.

3) U (UPDATE) in MongoDb >

3.1) Before **updating** let's create and query collection in MongoDB

Let's create collection and insert documents in it **before update** >

```
> db.employee.insert({id : 1,  firstName:"ankit"})
> db.employee.insert({id : 2,  firstName:"sam"})
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

Let's query/see what is there in collection **before update** >

```
> db.employee.find();
{ "_id" : ObjectId("584ebed11127dea5ece72742"), "id" : 1, "firstName" :
"ankit" }
{ "_id" : ObjectId("584ebee21127dea5ece72743"), "id" : 2, "firstName" : "sam" }
```

3.2) UPDATE **Example1** in MongoDB >

Let's **UPDATE** firstname where id=1 >

We will use update method of mongoDB

```
>
db.employee.update({_id:1},{$set:{firstName:"ankit_UPDATED"}})
```

Use the $set operator to replace the value of a field with the specified value.

Now let's display documents of collection **after update** >

```
> db.employee.find();
{ "_id" : ObjectId("584ebed11127dea5ece72742"), "id" : 1, "firstName"
: "ankit_UPDATED" }
{ "_id" : ObjectId("584ebee21127dea5ece72743"), "id" : 2, "firstName" :
"sam" }
```

3.3) UPDATE **Example2** in MongoDB >

Let's **UPDATE** id where firstName= "sam" >

```
> db.employee.update({firstName:"sam"},{$set:{id:22}})
```

Now, let's display documents of collection **after update** >

```
> db.employee.find();
{ "_id" : ObjectId("584ebed11127dea5ece72742"), "id" : 1,
"firstName" : "ankit_UPDATED" }
{ "_id" : ObjectId("584ebee21127dea5ece72743"), "id" : 22,
"firstName" : "sam" }
```

4) D (DELETE) in MongoDb >

4.1) Before **deleting** let's create and query collection in MongoDB

Let's create collection and insert documents in it **before delete** >

```
> db.employee.insert({id : 1,  firstName:"ankit"})
> db.employee.insert({id : 2,  firstName:"sam"})
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

Let's query/see what is there in collection **before delete** >

```
> db.employee.find();
{ "_id" : ObjectId("584ebed11127dea5ece72742"), "id" : 1, "firstName" :
"ankit" }
{ "_id" : ObjectId("584ebee21127dea5ece72743"), "id" : 2, "firstName" : "sam" }
```

4.2) DELETE **Example1** in MongoDB >

Let's **DELETE** document where id=1 >

We will use **remove** method of mongoDB.

```
> db.employee.remove({_id:1})
```

WriteResult object that tells us if the operation was successful or not.

Now, let's display documents of collection **after delete**>

```
> db.employee.find();
{ "_id" : ObjectId("584ebee21127dea5ece72743"), "id" : 2,
"firstName" : "sam" }
>
```

Only 1 document found in collection.

4.3) DELETE **Example2** in MongoDB >

Let's **DELETE** document where firstName= "sam" >

```
> db.employee.remove({firstName:"sam"})
```

Now, let's display documents of collection after **delete**>

```
> db.employee.find();
```

No document found in collection.

4.4) How to **Delete all documents from employee** collection in MongoDB>

```
> db.employee.remove({});
```

Let's query/see collection **after deleting all documents** >

```
> db.employee.find();
```

No, document was found.

SUMMARY>

So in this mongoDB tutorial we learned how to perform **CRUD operations in MongoDB** in MongoDb.

Use db.employee.drop() to drop collection in MongoDb.

1) Query (read/ display/ find/ search/ select) document (record/row) in collection (table) in mongoDB >

We will use **find** method of mongoDB.

Let's create new collection and insert document in it **before finding** >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

FIND Example > Query **all documents** of collection using find() method>

```
> db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

4 documents were found.

2) Display documents of collection in formatted manner.

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

FIND Example > Query **all document** of collection using find().pretty() method>

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

Output>

```
{
    "_id" : 1,
```

```
    "firstName" : "ankit"
}
{
    "_id" : 2,
    "firstName" : "ankit",
    "salary" : 1000
}
{
    "_id" : 3,
    "firstName" : "sam",
    "salary" : 2000
}
{
    "_id" : 4,
    "firstName" : "neh",
    "salary" : 3000
}
```

4 documents were found.

3) Selecting specific fields of documents in collection in MongoDB (Projections in MongoDB)

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

3.1) FIND Example > Display only **firstName** field of **document** of collection using find() method>

```
> db.employee.find( {}, {firstName : 1})
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit" }
{ "_id" : 3, "firstName" : "sam" }
{ "_id" : 4, "firstName" : "neh" }
```

By default **_id** field is always shown.

3.2) FIND Example > Display only **firstName and salary** field of **document** of collection using find() method>

```
> db.employee.find( {}, {firstName : 1, salary : 1})
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

By default **_id** field is always shown.

3.4) FIND Example > Display only **firstName and salary** field of **document** of collection, But **avoid _id field to be displayed** using find() method>

```
> db.employee.find( {}, { _id :0, firstName : 1, salary : 1})
```

Output>

```
{ "firstName" : "ankit" }
{ "firstName" : "ankit", "salary" : 1000 }
{ "firstName" : "sam", "salary" : 2000 }
{ "firstName" : "neh", "salary" : 3000 }
```

3.5) FIND Example > Display only **firstName and salary** field of **document** of collection **where salary > 1000**, But **avoid _id field to be displayed** using find() method>

```
> db.employee.find( { salary : {$gt : 1000} }, { _id :0,
firstName : 1, salary : 1})
```

Output>

```
{ "firstName" : "sam", "salary" : 2000 }
{ "firstName" : "neh", "salary" : 3000 }
```

4) FIND Example > find document where _id=1

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

FIND Example > **find** document where _id=1 >

```
> db.employee.find({_id:1})
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
```

FIND Example > **find** document where firstName= "sam" >

```
> db.employee.find({firstName:"sam"})
```

Output>

```
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
```

5) AND condition - using $and operator on document in collection in MongoDB

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

use **AND** condition - using **$and** operator.

Find employee **where**

_id = 1 and firstName = "ankit"

by using **find** method and **$and** operator.

```
> db.employee.find(
                        {$and : [
                                {_id:1},
                                {firstName:"ankit"}
                        ]
                }
                )
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
```

6) OR condition - using $or operator on document in collection in MongoDB

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

Use **OR** condition - using **$or** operator.

Find employee **where**

id = 1 or firstName = "ankit"

by using **find** method and **$or** operator.

```
> db.employee.find(
                        {$or : [
                                {_id:1},
                                {firstName:"ankit"}
                        ]
                }
                )
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
```

7) Method 1 to use **AND** and **OR** condition >

- using **$and** operator and
- using **$or** operator.

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

Find employee **where**

Salary =1000 and (id = 1 or firstName = "ankit")

by using **find** method, **$and** and **$or** operators.

```
> db.employee.find({
 $and : [{
   salary : 1000
  }, {
    $or : [{
      _id : 1
    }, {
      firstName : "ankit"
    }
   ]
```

```
  }
 ]
 })
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
```

Method 2 to use **AND** and **OR** condition >

- using **$and** operator and
- using **$or** operator.

Find employee **where**

Salary = 1000 and (id = 1 or firstName = "ankit")

by using **find** method, **$and** and **$or** operators.

Additionally, use **$eq** to find document where salary =1000

```
> db.employee.find({
 $and : [{
  salary : {$eq : 1000}
 }, {
  $or : [{
    _id : 1
   }, {
    firstName : "ankit"
   }
  ]
```

```
  }
 ]
 })
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
```

8) GREATER THAN (>) and GREATER THAN EQUALS (>=) conditions - using $where, $gt and $gte operator in MongoDB

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

GREATER THAN (>) and GREATER THAN EQUALS (>=) conditions.

8.1) GREATER THAN (>) - using $gt operator.

Find employee **where**

Salary > 1000

by using **find** method and **$gt** operator.

```
> db.employee.find( { salary : {$gt : 1000} } )
```

Output>

```
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

8.2) **GREATER THAN EQUALS (>=)** - using **$gte** operator.

Find employee **where**

Salary >= 1000

by using **find** method and **$gt** operator.

```
> db.employee.find( { salary : {$gte : 1000} } )
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

8.3) **GREATER THAN (>)** - using **$where** operator.

Find employee **where**

Salary > 1000

by using **find** method and **$where** operator.

```
> db.employee.find({  $where:"this.salary > 1000"})
```

Output>

```
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

8.4) **GREATER THAN EQUALS (>=)** - using **$where** operator.

Find employee **where**

Salary >= 1000

by using **find** method and **$where** operator.

```
> db.employee.find({  $where:"this.salary >= 1000"})
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

9) LESS THAN (<) and LESS THAN EQUALS (<=) - using **$where**, **$lt** and **$lte** operator in MongoDB

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

LESS THAN (<) and LESS THAN EQUALS (<=) conditions

9.1) **LESS THAN (<)** - using **$lt** operator.

Find employee **where**

Salary < 2000

by using **find** method and **$lt** operator.

```
> db.employee.find( { salary : {$lt : 2000} } )
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
```

9.2) **LESS THAN EQUALS (<=)** - using **$lte** operator.

Find employee **where**

Salary >= 2000

by using **find** method and **$lt** operator.

```
> db.employee.find( { salary : {$lte : 2000} } )
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
```

9.3) **LESS THAN (<)** - using **$where** operator.

Find employee **where**

Salary < 2000

by using **find** method and **$where** operator.

```
> db.employee.find({  $where:"this.salary < 2000"})
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
```

9.4) **LESS THAN EQUALS (<=)** - using **$where** operator.

Find employee **where**

Salary >= 2000

by using **find** method and **$where** operator.

```
> db.employee.find({  $where:"this.salary <= 2000"})
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
```

10) EQUALS (=) and NOT EQUALS (!=) conditions **-** using **$where**, **$eq** and **$ne** operator in MongoDB

EQUALS (=) and NOT EQUALS (!=) conditions

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

10.1) **EQUALS (=)** - using **$eq** operator.

Find employee **where**

Salary = 2000

by using **find** method and **$eq** operator.

```
> db.employee.find( { salary : {$eq : 2000} } )
```

Output>

```
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
```

10.2) **NOT EQUALS (!=)** - using **$ne** operator.

Find employee **where**

Salary != 2000

by using **find** method and **$ne** operator.

```
> db.employee.find( { salary : {$ne : 2000} } )
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

10.3) **EQUALS (=)** - using **$where** operator.

Find employee **where**

Salary = 2000

by using **find** method and **$where** operator.

```
> db.employee.find({  $where:"this.salary == 2000"})
```

Output>

```
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
```

10.4) **NOT EQUALS (!=)** - using **$where** operator.

Find employee **where**

Salary != 2000

by using **find** method and **$where** operator.

```
> db.employee.find({  $where:"this.salary != 2000"})
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

11) Find document from collection where field(column) EXISTS or not in collection in MongoDB >

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

11.1) FIND Example > **find** document from collection where field(column) **salary** exists >

Now, let's display document of collection where field(column) **salary** exists >

```
> db.employee.find({salary:{$exists : true}})
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

So, you will see that only newly inserted document which contains column salary was displayed.

11.2) Now, let's display documents of collection where field **salary DOESN'T** exists >

```
> db.employee.find({ salary : {$exists : false}})
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
```

So, you will see that newly inserted document which contains column salary was NOT displayed.

12) How to Limit number of documents fetched in MongoDB

Limit number of documents(record/rows) fetched from collection(table) in MongoDB

Limit method limits number of documents displayed.

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

12.1) FIND Example > Display only **first 2 documents** of collection in MongoDB >

We will use find() and **limit()** method>

```
> db.employee.find().limit(2)
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
```

Display only **first 2 documents** of collection.

12.2) FIND Example > Skip first document and display rest of **documents** of collection in MongoDB >

We will use find() and **skip()** method >

```
> db.employee.find().skip(1)
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

Skip 1 document of collection and display rest of **documents** of collection.

12.3) FIND Example > Display only 2nd and 3rd document of collection in MongoDB>

We will use find(), **limit()** and **skip()** method >

```
> db.employee.find().skip(1).limit(2)
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
```

Skip 1 document of collection and display 2 **documents** of collection.

12.4) FIND Example > Display only 2 documents of collection in MongoDB **where salary >= 1000** >

```
> db.employee.find( { salary : {$gte : 1000} }).limit(2)
```

Output>

```
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
```

13) Sorting (order by) documents in MongoDB

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

13.1) FIND Example > **Sort** documents of collection on basis of **salary in ascending** order in MongoDB>

We will use find() and **sort()** method>

```
> db.employee.find().sort({salary : 1})
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

13.2) Sql query equivalent to above MongoDB query is >

```
select * from employee
order by salary;
```

13.3) FIND Example > **Sort** documents of collection on basis of **salary in descending** order in MongoDB>

We will use find() and **sort()** method>

```
> db.employee.find().sort({salary : -1})
```

Output>

```
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 1, "firstName" : "ankit" }
```

13.4) Sql query equivalent to above MongoDB query is >

```
select * from employee
order by salary desc;
```

13.5) FIND Example > **Sort** documents of collection on basis of **firstName, salary in ascending** order in MongoDB>

We will use find() and **sort()** method>

```
> db.employee.find().sort({ firstName : 1, salary : 1})
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit", "salary" : 1000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
```

13.6) FIND Example > **Sort** documents of collection on basis of **salary in ascending** order in MongoDB **where salary > 1000** >

We will use find() and **sort()** method>

```
> db.employee.find({ salary : {$gt : 1000} }).sort({salary : 1})
```

Output>

```
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
```

13.7) FIND Example > **Sort** documents of collection on basis of **salary in descending** order in MongoDB **where salary > 1000** >

We will use find() and **sort()** method>

```
> db.employee.find({ salary : {$gt : 1000} }).sort({salary : -1})
```

Output>

```
{ "_id" : 4, "firstName" : "neh", "salary" : 3000 }
{ "_id" : 3, "firstName" : "sam", "salary" : 2000 }
```

14) **find** document where firstName length is greater (>) than 3

First, Let's create new collection and insert document in it >

```
> db.employee.insert({_id : 1,  firstName:"ankit"})
> db.employee.insert({_id : 2,  firstName:"ankit", salary : 1000 })
> db.employee.insert({_id : 3,  firstName:"sam", salary : 2000 })
> db.employee.insert({_id : 4,  firstName:"neh", salary : 3000 })
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

Let's **find** document where firstName length is greater (>) than 3

```
> db.employee.find({  $where:"this.firstName.length > 3"})
```

Output>

```
{ "_id" : 1, "firstName" : "ankit" }
{ "_id" : 2, "firstName" : "ankit", "salary" : "1000" }
```

15) Using like statement (as in sql) in MongoDB

15.1) First let's create records before using like statement (as in sql) in MongoDB

Let's create new collection and insert documents in it **before using like statement** >

```
> db.testCollection.insert({name: 'abc'})
> db.testCollection.insert({name: 'bcd'})
> db.testCollection.insert({name: 'def'})
```

First line above will create table (or collection) (if table already exists it will insert documents in it).

Read **all documents** of collection >

```
> db.testCollection.find()
{ "_id" : ObjectId("585008261127dea5ece72759"), "name" :
"abc" }
{ "_id" : ObjectId("585008261127dea5ece7275a"), "name" :
"bcd" }
{ "_id" : ObjectId("585008261127dea5ece7275b"), "name" :
"def" }
>
```

15.2) Below find document in mongoDB is similar to **like '%b%'** in sql >

```
> db.testCollection.find({name : /b/})
{ "_id" : ObjectId("585008261127dea5ece72759"), "name" :
"abc" }
{ "_id" : ObjectId("585008261127dea5ece7275a"), "name" :
"bcd" }
>
```

15.3) Below find document in mongoDB is similar to **like 'b%'** in sql >

```
> db.testCollection.find({name : /^b/})
{ "_id" : ObjectId("585008261127dea5ece7275a"), "name" :
"bcd" }
>
```

15.4) Below find document in mongoDB is similar to **like '%f'** in sql >

```
> db.testCollection.find({name : /f$/})
{ "_id" : ObjectId("585008261127dea5ece7275b"), "name" :
"def" }
```

```
>
```

1) **What is ObjectId** in MongoDB?

ObjectId is of **12-byte** hexadecimal string.

This is what ObjectId looks like > **ObjectId("hexadecimal")**
Example > ObjectId("584ebed11127dea5ece72742")

Let's see what exactly ObjectId consists of >

- First **4-byte** - represents the **seconds** since the **Unix epoch**,
- Next **3-byte** - represents **machine identifier,**
- Next **2-byte** - represents **process id, and**
- Next **3-byte** - represents **random value.**

2) **Inserting** in collection in MongoDB >

```
> db.employee.insert({ firstName:"ankit"})
```

Above line will create collection (if collection already exists it will insert records in it).

Let's query collection >

```
> db.employee.find();
```

Output>

```
{ "_id" : ObjectId("588341b303b0c717da77d641"), "firstName" : "ankit" }
```

We can see that field _**id** is formed **automatically**. Its **value** is ObjectId("588341b303b0c717da77d641")

3) **Creating new ObjectId()** in MongoDB>

Method ObjectId() - Returns the hexadecimal string representation of the object.

To generate a new ObjectId, Simply type **ObjectId()** with **no argument** and execute it

```
obj = ObjectId()
```

Output>
So, value of obj is -

```
ObjectId("588342aa03b0c717da77d642")
```

4) **Creating custom**/own ObjectId() in MongoDB>

To generate a ObjectId of **your choice**, Simply type **ObjectId("HexadecimalString")** with hexadecimal **string** of 12 bytes.
But, we need to **ensure** that **HexadecimalString** is **unique** otherwise error - 'errmsg" : "E11000 duplicate key error index' will be thrown.

```
myObj = ObjectId("123456aa01b0c234da56d789")
```

Output>
So, value of myObj is -

```
ObjectId("123456aa01b0c234da56d789")
```

5) Now, **Insert own objectId** in collection in MongoDB >

```
> db.employee.insert({ _id
:ObjectId("123341b303b0c717da77d123"),firstName:"ankit"})
```

6) How to **find ObjectId** was **created at what time** in MongoDB**?**

As we read above that first **4-byte** - represents the **seconds** since the **Unix epoch**,
Use **getTimestamp()** method.

```
time = ObjectId("588341b303b0c717da77d641").getTimestamp()
```

Output>
So, value of time is -

```
ISODate("2017-01-21T11:10:43Z")
```

7) How to **convert** ObjectId to **string** in MongoDB**?**

Use **str**

```
x = ObjectId("588341b303b0c717da77d641").str
```

Output>
So, value of x is -

```
588341b303b0c717da77d641
```

8) SUMMARY of *ObjectId in MongoDB*>

So in this mongoDB tutorial we learned about objectId in MongoDB.

1) **What is ObjectId** in MongoDB?
ObjectId is of **12-byte** hexadecimal string.
Example of > ObjectId("584ebed11127dea5ece72742")

- First **4-byte** - represents the **seconds** since the **Unix epoch**,
- Next **3-byte** - represents **machine identifier,**
- Next **2-byte** - represents **process id, and**
- Next **3-byte** - represents **random value.**

2) **Inserting** in collection in MongoDB >

```
> db.employee.insert({ firstName:"ankit"})
```

Let's query collection >

```
> db.employee.find();
{ "_id" : ObjectId("588341b303b0c717da77d641"), "firstName" : "ankit" }
```

3) **Creating new ObjectId()** in MongoDB>

```
obj = ObjectId()
```

4) **Creating custom**/own ObjectId() in MongoDB>

```
myObj = ObjectId("123456aa01b0c234da56d789")
```

5) Now, **Insert own objectId** in collection in MongoDB >

```
db.employee.insert({ _id
:ObjectId("123341b303b0c717da77d123"),firstName:"ankit"})
```

6) How to **find ObjectId** was **created at what time** in MongoDB**?**
Use **getTimestamp()** method.

```
time = ObjectId("588341b303b0c717da77d641").getTimestamp()
```

7) How to **convert** ObjectId to **string** in MongoDB**?**

```
x = ObjectId("588341b303b0c717da77d641").str
```

Aggregation functions in MongoDB >

1. Find **count of all employee by group** in collection in MongoDB
2. **count()** Method - **Find count of all employee** in collection in MongoDB
3. **$sum** operator - **sum of salary by group** in MongoDB
4. **$avg operator** - **average** of salary by group in MongoDB
5. **$min operator** - Find **minimum value from documents** in MongoDB
6. **$max operator** in MongoDB - Find **maximum** value from documents
7. **$first operator** in MongoDB
8. **$last** operator in MongoDB
9. **$stdDevPop** operator in MongoDB - Find **standard deviation** from documents
10. $push operator in MongoDB
11. $addToSet operator in MongoDB

1. Find count of all employee by group in collection in MongoDB

1) First let's create insert documents in collection in MongoDB

1.1) First, Let's create new collection and insert document >

```
db.employee.insert({_id : 1,  firstName:"ank", salary : 1000 })
db.employee.insert({_id : 2,  firstName:"ank", salary : 2000 })
db.employee.insert({_id : 3,  firstName:"sag", salary : 3000 })
db.employee.insert({_id : 4,  firstName:"sag", salary : 4000 })
db.employee.insert({_id : 5,  firstName:"neh", salary : 5000 })
```

Above will create collection (or table) (if collection already exists it will insert documents in it).

1.2) FIND > Query **all documents** of collection using find() method>

```
db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ank", "salary" : 1000 }
{ "_id" : 2, "firstName" : "ank", "salary" : 2000 }
{ "_id" : 3, "firstName" : "sag", "salary" : 3000 }
{ "_id" : 4, "firstName" : "sag", "salary" : 4000 }
{ "_id" : 5, "firstName" : "neh", "salary" : 5000 }
```

2) Find **count** of all **employee** with **same firstName** in collection in MongoDB >

2.1) We will **group employee by firstName** and **find count of of each group.**
We will aggregate() method and $group operator.

```
db.employee.aggregate([{$group : {_id : "$firstName", countOfEmp : {$sum : 1}}}])
```

Output >

```
{ "_id" : "neh", "countOfEmp" : 1 }
{ "_id" : "sag", "countOfEmp" : 2 }
{ "_id" : "ank", "countOfEmp" : 2 }
```

2.2) Sql query equivalent to above MongoDB query is >

```
select firstName, count(*) countOfEmp
from employee
group by firstName;
```

3) Summary -

So in this MongoDB tutorial we learned how to **Find count of all employee by group in collection in MongoDB**

2. **count() Method** - Find count of all employee in collection in MongoDB

count() method in Mongodb>

count() method to find **count of documents** in collection in MongoDB.

1) First let's create insert documents in collection in MongoDB

1.1) First, Let's create new collection and insert document >

```
db.employee.insert({_id : 1,  firstName:"ank", salary : 1000 })
db.employee.insert({_id : 2,  firstName:"ank", salary : 2000 })
db.employee.insert({_id : 3,  firstName:"sag", salary : 3000 })
db.employee.insert({_id : 4,  firstName:"sag", salary : 4000 })
db.employee.insert({_id : 5,  firstName:"neh", salary : 5000 })
```

Above will create collection (or table) (if collection already exists it will insert documents in it).

1.2) FIND > Query **all documents** of collection using find() method>

```
db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ank", "salary" : 1000 }
{ "_id" : 2, "firstName" : "ank", "salary" : 2000 }
{ "_id" : 3, "firstName" : "sag", "salary" : 3000 }
{ "_id" : 4, "firstName" : "sag", "salary" : 4000 }
{ "_id" : 5, "firstName" : "neh", "salary" : 5000 }
```

2) Find **count of all employee** in collection in MongoDB >

2.1) We will use count() method to find **count of all employee** in collection in MongoDB.

```
db.employee.count()
```

Output >

```
5
```

2.2) Sql query equivalent to above MongoDB query is >

```
select count(*) from employee;
```

3) Find **count** of **all** employee **where salary >= 2000** in collection in MongoDB >

3.1) We will use find() and count() methods.

```
db.employee.find( { salary : {$gte : 2000} } ).count()
```

Output >

```
4
```

3.2) We can also use $count() operator to find **count of all employee where salary >= 2000** in collection in MongoDB.
Note : $count can be used in MongoDB 3.4 or above.

```
db.employee.aggregate(
 [
  {
   $match: {
    salary : {
     $gte: 2000
    }
   }
  },
  {
   $count: "firstName"
  }
 ]
)
```

Output >

```
5
```

3.3) Sql query equivalent to above MongoDB query is >

```
select count(*) from employee where salary >= 2000
```

4) Summary -

So in this MongoDB tutorial we learned how to use count() Method - How to find count of all employee in collection in MongoDB

count of all employee in collection in MongoDB.

```
db.employee.count()
```

Find **count** of **all** employee **where salary >= 2000** in collection in MongoDB >

```
db.employee.find( { salary : {$gte : 2000} } ).count()
```

3. **$sum operator** - sum of salary by group in MongoDB

$sum operator in MongoDB >

$sum operator returns the sum of all numeric values of documents in the collection in MongoDB.

1) First let's create insert documents in collection in MongoDB

1.1) Let's create new collection and insert document in it **before finding** >

```
db.employee.insert({_id : 1,  firstName:"ank", salary : 1000 })
db.employee.insert({_id : 2,  firstName:"ank", salary : 2000 })
db.employee.insert({_id : 3,  firstName:"sag", salary : 3000 })
db.employee.insert({_id : 4,  firstName:"sag", salary : 4000 })
db.employee.insert({_id : 5,  firstName:"neh", salary : 5000 })
```

Above will create collection (or table) (if collection already exists it will insert documents in it).

1.2) FIND > Query **all documents** of collection using find() method>

```
db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ank", "salary" : 1000 }
{ "_id" : 2, "firstName" : "ank", "salary" : 2000 }
{ "_id" : 3, "firstName" : "sag", "salary" : 3000 }
{ "_id" : 4, "firstName" : "sag", "salary" : 4000 }
{ "_id" : 5, "firstName" : "neh", "salary" : 5000 }
```

2) Find **sum of salary** of all **employee** with **same firstName** in collection in MongoDB >

Step 2.1 - We will **group employee by firstName** and **find sum of salary of each group**.
Step 2.2- We will use aggregate() method, $sum operator and $group operator.

```
db.employee.aggregate([{$group : {_id : "$firstName", salary_sum : {$sum : "$salary"}}}])
```

Output >

```
{ "_id" : "ank", "salary_sum" : 3000 }
{ "_id" : "sag", "salary_sum" : 7000 }
{ "_id" : "neh", "salary_sum" : 5000 }
```

2.3) Sql query equivalent to above MongoDB query is >

```
select firstName, sum(salary) salary_sum from employee group by firstName;
```

3) Find **sum of salary** of all **employee** in collection in MongoDB >

Step 3.1) - We will **group employee by null (nothing)** and **find sum of salary of group** (there will be only 1 group - i.e. whole documents of collection)
Step 3.2)- We will use aggregate() method, $group operator, and **_id : null**

```
db.employee.aggregate([{$group : {_id : null, salary_sum : {$sum : "$salary"}}}])
```

Output >

```
{ "_id" : null, "salary_sum" : 15000 }
```

3.3) Sql query equivalent to above MongoDB query is >

```
select sum(salary) salary_sum from employee;
```

4) How to write **aggregate query** with **where clause** in MongoDB >

Find **sum of salary** of all **employee** with **same firstName** **where salary > 2000** in collection in MongoDB >

Step 4.1 - We will find document **where salary > 2000**
Step 4.2 - We will use **$match** operator
Step 4.3 - Then we will **group employee by firstName** and **find sum of salary of each group**.
Step 4.4 - We will use aggregate() method, $group operator

```
db.employee.aggregate([
 { $match: { salary : { $gt: 2000} } },
 {$group : {_id : "$firstName", salary_sum : {$sum : "$salary"}}}])
```

Output >

```
{ "_id" : "sag", "salary_sum" : 7000 }
{ "_id" : "neh", "salary_sum" : 5000 }
```

4.5) Sql query equivalent to above MongoDB query is >

```
select firstName, sum(salary) salary_sum from employee
where salary > 2000
group by firstName
```

5) Summary -

So in this MongoDB tutorial we learned how to use $sum operator. $sum operator returns the sum of all numeric values of documents in the collection in MongoDB.

Find **sum of salary** of all **employee** with **same firstName** in collection in MongoDB >

```
db.employee.aggregate([{$group : {_id : "$firstName", salary_sum : {$sum : "$salary"}}}])
```

Find **sum of salary** of all **employee** in collection in MongoDB >

```
db.employee.aggregate([{$group : {_id : null, salary_sum : {$sum : "$salary"}}}])
```

Find **sum of salary** of all **employee** with **same firstName** **where salary > 2000** in collection in MongoDB >

```
db.employee.aggregate([
 { $match: { salary : { $gt: 2000} } },
 {$group : {_id : "$firstName", salary_sum : {$sum : "$salary"}}}])
```

4. **$avg operator** - average of salary by group in MongoDB

$avg operator in MongoDB >

$avg operator returns the average of all numeric values of documents in the collection in MongoDB.

1) First let's create insert documents in collection in MongoDB

1.1) First, Let's create new collection and insert document >

```
db.employee.insert({_id : 1,  firstName:"ank", salary : 1000 })
db.employee.insert({_id : 2,  firstName:"ank", salary : 2000 })
db.employee.insert({_id : 3,  firstName:"sag", salary : 3000 })
db.employee.insert({_id : 4,  firstName:"sag", salary : 4000 })
db.employee.insert({_id : 5,  firstName:"neh", salary : 5000 })
```

Above will create collection (or table) (if collection already exists it will insert documents in it).

1.2) FIND > Query **all documents** of collection using find() method>

```
db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ank", "salary" : 1000 }
{ "_id" : 2, "firstName" : "ank", "salary" : 2000 }
{ "_id" : 3, "firstName" : "sag", "salary" : 3000 }
{ "_id" : 4, "firstName" : "sag", "salary" : 4000 }
{ "_id" : 5, "firstName" : "neh", "salary" : 5000 }
```

2) Find **average of salary** of all **employee** with **same firstName** in collection in MongoDB >

Step 2.1 - We will **group employee by firstName** and **find average of salary of each group**.
Step 2.2- We will use aggregate() method, $avg operator and $group operator.

```
db.employee.aggregate([{$group : {_id : "$firstName", salary_average : {$avg : "$salary"}}}])
```

Output >

```
{ "_id" : "ank", "salary_average" : 1500 }
{ "_id" : "sag", "salary_average" : 3500 }
{ "_id" : "neh", "salary_average" : 5000 }
```

2.3) Sql query equivalent to above MongoDB query is >

```
select firstName, avg(salary) salary_average from employee group by firstName;
```

3) Find **average of salary** of all **employee** in collection in MongoDB >

Step 3.1 - We will **group employee by null (nothing)** and **find average of salary of group** (there will be only 1 group - i.e. whole documents of collection)
Step 3.2- We will use aggregate() method, $group operator, $avg operator and **_id : null**

```
db.employee.aggregate([{$group : {_id : null, salary_average : {$avg : "$salary"}}}])
```

Output >

```
{ "_id" : null, "salary_average" : 3000 }
```

3.3) Sql query equivalent to above MongoDB query is >

```
select avg(salary) salary_average from employee;
```

4) How to write **aggregate query** with **where clause** in MongoDB >

Find **average of salary** of all **employee** with **same firstName** **where salary > 2000** in collection in MongoDB >

Step 4.1 - We will find document **where salary > 2000**
Step 4.2 - We will use **$match** operator
Step 4.3 - Then we will **group employee by firstName** and **find average of salary of each group**.
Step 4.4 - We will use aggregate() method, $group operator, $avg operator.

```
db.employee.aggregate([
 { $match: { salary : { $gt: 2000} } },
 {$group : {_id : "$firstName", salary_average : {$avg : "$salary"}}}])
```

Output >

```
{ "_id" : "sag", "salary_average" : 3500 }
{ "_id" : "neh", "salary_average" : 5000 }
```

4.5) Sql query equivalent to above MongoDB query is >

```
select firstName, avg(salary) salary_average from employee
where salary > 2000
group by firstName
```

5) Summary -

So in this MongoDB tutorial we learned how to use $avg operator in MongoDB. $avg operator returns the average of all numeric values of documents in the collection in MongoDB.

Find **average of salary** of all **employee** with **same firstName** in collection in MongoDB >

```
db.employee.aggregate([{$group : {_id : "$firstName", salary_average : {$avg : "$salary"}}}])
```

Find **average of salary** of all **employee** in collection in MongoDB >

```
db.employee.aggregate([{$group : {_id : null, salary_average : {$avg : "$salary"}}}])
```

How to write **aggregate query** with **where clause** in MongoDB >

```
db.employee.aggregate([
 { $match: { salary : { $gt: 2000} } },
 {$group : {_id : "$firstName", salary_average : {$avg : "$salary"}}}])
```

5. **$min operator** - Find minimum value from documents in MongoDB

$min operator in MongoDB >

$min operator returns the minimum value from documents in the collection in MongoDB.

1) First let's create insert documents in collection in MongoDB

1.1) First, Let's create new collection and insert document >

```
db.employee.insert({_id : 1,  firstName:"ank", salary : 1000 })
db.employee.insert({_id : 2,  firstName:"ank", salary : 2000 })
db.employee.insert({_id : 3,  firstName:"sag", salary : 3000 })
db.employee.insert({_id : 4,  firstName:"sag", salary : 4000 })
db.employee.insert({_id : 5,  firstName:"neh", salary : 5000 })
```

Above will create collection (or table) (if collection already exists it will insert documents in it).

1.2) FIND > Query **all documents** of collection using find() method>

```
db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ank", "salary" : 1000 }
{ "_id" : 2, "firstName" : "ank", "salary" : 2000 }
{ "_id" : 3, "firstName" : "sag", "salary" : 3000 }
{ "_id" : 4, "firstName" : "sag", "salary" : 4000 }
{ "_id" : 5, "firstName" : "neh", "salary" : 5000 }
```

2) Find **minimum salary** of **employee** with **same firstName** in collection in MongoDB >

Step 2.1 - We will **group employee by firstName** and **find minimum salary of each group**.
Step 2.2- We will use aggregate() method, $min operator and $group operator.

```
> db.employee.aggregate([{$group : {_id : "$firstName",
salary_minimum : {$min : "$salary"}}}])
```

Output >

```
{ "_id" : "ank", "salary_minimum" : 1000 }
{ "_id" : "sag", "salary_minimum" : 3000 }
{ "_id" : "neh", "salary_minimum" : 5000 }
```

2.3) Sql query equivalent to above MongoDB query is >

```
select firstName, min(salary) salary_minimum from employee group by
firstName;
```

3) Find **minimum salary** of all **employee** in collection in MongoDB >

Step 3.1 - We will **group employee by null (nothing)** and **find minimum salary of group** (there will be only 1 group - i.e. whole documents of collection)
Step 3.2- We will use aggregate() method, $min operator $group operator.

```
> db.employee.aggregate([{$group : {_id : null, salary_minimum :
{$min : "$salary"}}}])
```

Output >

```
{ "_id" : null, "salary_minimum" : 1000 }
```

3.3) Sql query equivalent to above MongoDB query is >

```
select min(salary) salary_minimum from employee;
```

4) How to write **aggregate query** with **where clause** in MongoDB >

Find **minimum salary** of all **employee** with **same firstName** **where salary > 2000** in collection in MongoDB >
Step 4.1 - We will find document **where salary > 2000**

Step 4.2 - We will use **$match** operator
Step 4.3 - Then we will **group employee by firstName** and **find minimum salary of each group**.
Step 4.4 - We will use aggregate() method, $group operator, $min operator and **_id : null**

```
> db.employee.aggregate([
 { $match: { salary : { $gt: 2000} } },
 {$group : {_id : "$firstName", salary_minimum: {$min : "$salary"}}}])
```

Output >

```
{ "_id" : "sag", "salary_minimum" : 3000 }
{ "_id" : "neh", "salary_minimum" : 5000 }
```

4.5) Sql query equivalent to above MongoDB query is >

```
select firstName, min(salary) salary_minimum from employee
where salary > 2000
group by firstName
```

5) Summary -

So in this MongoDB tutorial we learned about $min operator in MongoDB. $min operator returns the minimum value from documents in the collection in MongoDB.

Find **minimum salary** of **employee** with **same firstName** in collection in MongoDB >

```
> db.employee.aggregate([{$group : {_id : "$firstName",
salary_minimum : {$min : "$salary"}}}])
```

Find **minimum salary** of all **employee** in collection in MongoDB >

```
> db.employee.aggregate([{$group : {_id : null, salary_minimum :
{$min : "$salary"}}}])
```

Find **minimum salary** of all **employee** with **same firstName** **where salary > 2000** in collection in MongoDB >

```
> db.employee.aggregate([
 { $match: { salary : { $gt: 2000} } },
 {$group : {_id : "$firstName", salary_minimum: {$min : "$salary"}}}])
```

6. **$max operator** in MongoDB - Find **maximum** value from documents

$max operator in MongoDB >

$max operator returns the maximum value from documents in the collection in MongoDB.

1) First let's create insert documents in collection in MongoDB

1.1) First, Let's create new collection and insert document >

```
db.employee.insert({_id : 1,  firstName:"ank", salary : 1000 })
db.employee.insert({_id : 2,  firstName:"ank", salary : 2000 })
db.employee.insert({_id : 3,  firstName:"sag", salary : 3000 })
db.employee.insert({_id : 4,  firstName:"sag", salary : 4000 })
db.employee.insert({_id : 5,  firstName:"neh", salary : 5000 })
```

Above will create collection (or table) (if collection already exists it will insert documents in it).

1.2) FIND > Query **all documents** of collection using find() method>

```
db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ank", "salary" : 1000 }
{ "_id" : 2, "firstName" : "ank", "salary" : 2000 }
{ "_id" : 3, "firstName" : "sag", "salary" : 3000 }
{ "_id" : 4, "firstName" : "sag", "salary" : 4000 }
{ "_id" : 5, "firstName" : "neh", "salary" : 5000 }
```

2) Find **maximum salary** of **employee** with **same firstName** in collection in MongoDB >

Step 2.1 - We will **group employee by firstName** and **find maximum salary of each group**.
Step 2.2- We will use aggregate() method, $group operator and $max operator.

```
db.employee.aggregate([{$group : {_id : "$firstName", salary_maximum : {$max : "$salary"}}}])
```

Output >

```
{ "_id" : "ank", "salary_maximum" : 2000 }
{ "_id" : "sag", "salary_maximum" : 4000 }
{ "_id" : "neh", "salary_maximum" : 5000 }
```

2.3) Sql query equivalent to above MongoDB query is >

```
select firstName, max(salary) salary_maximum from employee group by firstName;
```

3) Find **maximum salary** of all **employee** in collection in MongoDB >

Step 3.1 - We will **group employee by null (nothing)** and **find maximum salary of group** (there will be only 1 group - i.e. whole documents of collection)
Step 3.2- We will use aggregate() method, $group operator, $max operator and **_id : null**

```
db.employee.aggregate([{$group : {_id : null, salary_maximum : {$max : "$salary"}}}])
```

Output >

```
{ "_id" : null, "salary_maximum" : 5000 }
```

3.3) Sql query equivalent to above MongoDB query is >

```
select max(salary) salary_maximum from employee;
```

4) How to write **aggregate query** with **where clause** in MongoDB >

Find **maximum salary** of all **employee** with **same firstName** **where salary > 2000** in collection in MongoDB >
Step 4.1 - We will find document **where salary > 2000**
Step 4.2 - We will use **$match** operator

Step 4.3 - Then we will **group employee by firstName** and **find maximum salary of each group**.
Step 4.4 - We will use aggregate() method, $group operator.

```
db.employee.aggregate([
 { $match: { salary : { $gt: 2000} } },
 {$group : {_id : "$firstName", salary_maximum : {$max : "$salary"}}}])
```

Output >

```
{ "_id" : "sag", "salary_maximum" : 4000 }
{ "_id" : "neh", "salary_maximum" : 5000 }
```

4.4) Sql query equivalent to above MongoDB query is >

```
select firstName, max(salary) salary_maximum from employee
where salary > 2000
group by firstName
```

5) Summary -

So in this MongoDB tutorial we learned about $max operator in MongoDB. $max operator returns the maximum value from documents in the collection in MongoDB.

Find **maximum salary** of **employee** with **same firstName** in collection in MongoDB >

```
db.employee.aggregate([{$group : {_id : "$firstName", salary_maximum
: {$max : "$salary"}}}])
```

Find **maximum salary** of all **employee** in collection in MongoDB >

```
db.employee.aggregate([{$group : {_id : null, salary_maximum : {$max
: "$salary"}}}])
```

Find **maximum salary** of all **employee** with **same firstName** **where salary > 2000** in collection in MongoDB >

```
db.employee.aggregate([
 { $match: { salary : { $gt: 2000} } },
```

```
{$group : {_id : "$firstName", salary_maximum : {$max : "$salary"}}])
```

7. **$first operator** in MongoDB

$first operator in MongoDB >

$first operator **returns the first document** from documents in the collection in MongoDB.

1) First let's create insert documents in collection in MongoDB

1.1) First, Let's create new collection and insert document >

```
db.employee.insert({_id : 1,  firstName:"ank", salary : 1000 })
db.employee.insert({_id : 2,  firstName:"ank", salary : 2000 })
db.employee.insert({_id : 3,  firstName:"sag", salary : 3000 })
db.employee.insert({_id : 4,  firstName:"sag", salary : 4000 })
db.employee.insert({_id : 5,  firstName:"neh", salary : 5000 })
```

Above will create collection (or table) (if collection already exists it will insert documents in it).

1.2) FIND > Query **all documents** of collection using find() method>

```
db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ank", "salary" : 1000 }
{ "_id" : 2, "firstName" : "ank", "salary" : 2000 }
{ "_id" : 3, "firstName" : "sag", "salary" : 3000 }
{ "_id" : 4, "firstName" : "sag", "salary" : 4000 }
{ "_id" : 5, "firstName" : "neh", "salary" : 5000 }
```

2) Find **first salary** of **employee** with **same firstName** in collection in MongoDB >

Step 2.1- We will use aggregate() method, $group operator and $first operator.
Step 2.2 - **group employee by firstName** and **find first salary of each group**.

```
db.employee.aggregate([{$group : {_id : "$firstName", salary_first : {$first : "$salary"}}}])
```

Output >

```
{ "_id" : "ank", "salary_first" : 1000 }
{ "_id" : "sag", "salary_first" : 3000 }
{ "_id" : "neh", "salary_first" : 5000 }
```

3) Find **first salary** of all **employee** in collection in MongoDB >

Step 3.1 - We will **group employee by null (nothing)** and **find first salary of group** (there will be only 1 group - i.e. whole documents of collection)
Step 3.2- We will use aggregate() method, $group operator, $first operator.

```
db.employee.aggregate([{$group : {_id : null, salary_first : {$first : "$salary"}}}])
```

Output >

```
{ "_id" : null, "salary_first" : 1000 }
```

4) Summary -

So in this MongoDB tutorial we learned $first operator in MongoDB. $first operator **returns the first document** from documents in the collection in MongoDB.

Find **first salary** of **employee** with **same firstName** in collection in MongoDB >

```
db.employee.aggregate([{$group : {_id : "$firstName", salary_first : {$first : "$salary"}}}])
```

Find **first salary** of all **employee** in collection in MongoDB >

```
db.employee.aggregate([{$group : {_id : null, salary_first : {$first : "$salary"}}}])
```

8. **$last** operator in MongoDB

$last operator in MongoDB >

$last operator **returns the last document** from documents in the collection in MongoDB.

1) First let's create insert documents in collection in MongoDB

1.1) First, Let's create new collection and insert document >

```
db.employee.insert({_id : 1,  firstName:"ank", salary : 1000 })
db.employee.insert({_id : 2,  firstName:"ank", salary : 2000 })
db.employee.insert({_id : 3,  firstName:"sag", salary : 3000 })
db.employee.insert({_id : 4,  firstName:"sag", salary : 4000 })
db.employee.insert({_id : 5,  firstName:"neh", salary : 5000 })
```

Above will create collection (or table) (if collection already exists it will insert documents in it).

1.2) FIND > Query **all documents** of collection using find() method>

```
db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ank", "salary" : 1000 }
{ "_id" : 2, "firstName" : "ank", "salary" : 2000 }
{ "_id" : 3, "firstName" : "sag", "salary" : 3000 }
{ "_id" : 4, "firstName" : "sag", "salary" : 4000 }
{ "_id" : 5, "firstName" : "neh", "salary" : 5000 }
```

2) Find **last salary** of **employee** with **same firstName** in collection in MongoDB >

Step 2.1 - We will **group employee by firstName** and **find last salary of each group**.
Step 2.2 - We will use aggregate() method, $last operator and $group operator.

```
> db.employee.aggregate([{$group : {_id : "$firstName", salary_last : {$last : "$salary"}}}])
```

Output >

```
{ "_id" : "ank", "salary_last" : 2000 }
{ "_id" : "sag", "salary_last" : 4000 }
{ "_id" : "neh", "salary_last" : 5000 }
```

3) Find **last salary** of all **employee** in collection in MongoDB >

Step 3.1 - We will **group employee by null (nothing)** and **find maximum salary of group** (there will be only 1 group - i.e. whole documents of collection)
Step 3.2- We will use aggregate() method, $group operator, $last operator

```
> db.employee.aggregate([{$group : {_id : null, salary_last : {$last : "$salary"}}}])
```

Output >

```
{ "_id" : null, "salary_last" : 5000 }
```

4) Summary -

So in this MongoDB tutorial we learned about $last operator in MongoDB. $last operator **returns the last document** from documents in the collection in MongoDB.

Find **last salary** of **employee** with **same firstName** in collection in MongoDB >

```
> db.employee.aggregate([{$group : {_id : "$firstName", salary_last : {$last : "$salary"}}}])
```

Find **last salary** of all **employee** in collection in MongoDB >

```
> db.employee.aggregate([{$group : {_id : null, salary_last : {$last : "$salary"}}}])
```

9. **$stdDevPop** operator in MongoDB - Find **standard deviation** from documents

$stdDevPop operator in MongoDB >

$stdDevPop operator returns the **standard deviation** from documents in the collection in MongoDB.

1) First let's create insert documents in collection in MongoDB

1.1) First, Let's create new collection and insert document >

```
db.employee.insert({_id : 1,  firstName:"ank", salary : 1000 })
db.employee.insert({_id : 2,  firstName:"ank", salary : 2000 })
db.employee.insert({_id : 3,  firstName:"sag", salary : 3000 })
db.employee.insert({_id : 4,  firstName:"sag", salary : 4000 })
db.employee.insert({_id : 5,  firstName:"neh", salary : 5000 })
```

Above will create collection (or table) (if collection already exists it will insert documents in it).

1.2) FIND > Query **all documents** of collection using find() method>

```
db.employee.find()
```

Output>

```
{ "_id" : 1, "firstName" : "ank", "salary" : 1000 }
{ "_id" : 2, "firstName" : "ank", "salary" : 2000 }
{ "_id" : 3, "firstName" : "sag", "salary" : 3000 }
{ "_id" : 4, "firstName" : "sag", "salary" : 4000 }
{ "_id" : 5, "firstName" : "neh", "salary" : 5000 }
```

2) Find **standard deviation** on salary of **employee** with **same firstName** in collection in MongoDB >

Step 2.1 - We will use aggregate() method, $group operator and $stdDevPop operator to

Step 2.2 - **group employee by firstName** and **find standard deviation of each group**.

```
db.employee.aggregate([{$group : {_id : "$firstName", standardDeviation : {$stdDevPop : "$salary"}}}])
```

Note : $stdDevPop can be used in MongoDB 3.2 or above.

3) Summary -

So in this MongoDB tutorial we learned $stdDevPop operator in MongoDB. $stdDevPop operator returns the **standard deviation** from documents in the collection in MongoDB.

Find standard deviation of **employee** with **same firstName** in collection in MongoDB >

```
db.employee.aggregate([{$group : {_id : "$firstName", standardDeviation : {$stdDevPop : "$salary"}}}])
```

10. **$push operator** in MongoDB

1) **$push** operator in MongoDB

$push operator **returns an array of *all* values in the group** from collection in MongoDB.

2) Find **all salary** of all **employee** with **same firstName** in **array** from collection in MongoDB >

Step 1 - We will **group employee by firstName** and **find all salary in each group**.
Step 2- We will use aggregate() method, $group operator and $push operator

```
> db.employee.aggregate([{$group : {_id : "$firstName", salary_all : {$push : "$salary"}}}])
```

Output >

```
{ "_id" : "ank", "salary_all" : [ 1000, 2000 ] }
{ "_id" : "sag", "salary_all" : [ 3000, 4000 ] }
{ "_id" : "neh", "salary_all" : [ 5000, 5000 ] }
```

3) Find **all salary** of all **employee** in **array** from collection in MongoDB >

Step 1 - We will **group employee by null (nothing)** and **find all salary in group** (there will be only 1 group - i.e. whole documents of collection)
Step 2- We will use aggregate() method, $group operator, $push operator and **_id : null**

```
> db.employee.aggregate([{$group : {_id : null, salary_all : {$push : "$salary"}}}])
```

Output >

```
{ "_id" : null, "salary_all" : [ 1000, 2000, 3000, 4000, 5000, 5000 ] }
```

4) How to write **aggregate query** with **where clause** in MongoDB >

Find **all salary** of all **employee** with **same firstName** **where salary > 2000** in **array** from collection in MongoDB >
Step 1 - We will find document **where salary > 2000**
Step 2 - We will use **$match** operator
Step 3 - Then we will **group employee by firstName** and **find all salary in each group**.
Step 4 - We will use aggregate() method, $group operator, and **_id : null**

```
> db.employee.aggregate([
 { $match: { salary : { $gt: 2000} } },
 {$group : {_id : "$firstName", salary_all : {$push : "$salary"}}}])
```

Output >

```
{ "_id" : "sag", "salary_all" : [ 3000, 4000 ] }
{ "_id" : "neh", "salary_all" : [ 5000, 5000 ] }
```

11. **$addToSet** operator in MongoDB

1) $addToSet operator in MongoDB.

$addToSet operator **returns an array of *all* UNIQUE values in the group** from collection in MongoDB.

2) Find **all salary** of all **employee** with **same firstName** in **array** from collection in MongoDB >

Step 1 - We will **group employee by firstName** and **find all UNIQUE salary in each group**.
Step 2- We will use aggregate() method, $group operator and $addToSet operator

```
> db.employee.aggregate([{$group : {_id : "$firstName", salary_all : {$addToSet : "$salary"}}}])
```

Output >

```
{ "_id" : "ank", "salary_all" : [ 1000, 2000 ] }
{ "_id" : "sag", "salary_all" : [ 3000, 4000 ] }
{ "_id" : "neh", "salary_all" : [ 5000 ] }
```

3) Find **all UNIQUE salary** of all **employee** in **array** from collection in MongoDB >

Step 1 - We will **group employee by null (nothing)** and **find all salary in group** (there will be only 1 group - i.e. whole documents of collection)
Step 2- We will use aggregate() method, $group operator, $addToSet operator and **_id : null**

```
> db.employee.aggregate([{$group : {_id : null, salary_all : {$addToSet : "$salary"}}}])
```

Output >

```
{ "_id" : null, "salary_all" : [ 1000, 2000, 3000, 4000, 5000 ] }
```

4) How to write **aggregate query** with **where clause** in MongoDB >

Find **all UNIQUE salary** of all **employee** with **same firstName** **where salary > 2000** in **array** from collection in MongoDB >
Step 1 - We will find document **where salary > 2000**
Step 2 - We will use **$match** operator
Step 3 - Then we will **group employee by firstName** and **find all salary in each group**.
Step 4 - We will use aggregate() method, $group operator, and **_id : null**

```
> db.employee.aggregate([
 { $match: { salary : { $gt: 2000} } },
 {$group : {_id : "$firstName", salary_all : {$addToSet : "$salary"}}}])
```

Output >

```
{ "_id" : "sag", "salary_all" : [ 3000, 4000 ] }
{ "_id" : "neh", "salary_all" : [ 5000 ] }
```

*1) (1-to-1) **One-to-One** Relationships with **Embedded document >***

Embed the ADDRESS document in the STUDENT **document**. (I will recommend you to go for this approach)
It helps in fetching all student and `address` **document** in **one query.**

1.1) Create collection in MongoDB >

STEP 1.1) create and insert in STUDENT collection >

```
db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit",
 "ADDRESS": {"CITY": "Delhi"}
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam",
 "ADDRESS": {"CITY": "London"}
})
```

When to use this approach?

When number of **transactions** are too high and need to be done atomically > This can be used when address is fetched too frequently. i.e (Read and write operations are too high).

MongoDB does **not** support transactions on **multiple document**.But, in **MongoDB you can perform atomic operations on a single document.**

So, **while designing your database and collections** you must try and ensure that all the **related data (**as much as possible**) which is needed to be updated atomically must be placed in single document as embedded documents** (in form of nest arrays OR nested documents)

1.2 > Now, let's read/query/find in above MongoDB collection >

Query 1.2.1 > Query to show all students

```
db.STUDENT.find().pretty()
```

Output>

```
{ "_id" : 1, "FIRST_NAME" : "Ankit", "ADDRESS" : { "CITY" : "Delhi" } }
{ "_id" : 2, "FIRST_NAME" : "Sam", "ADDRESS" : { "CITY" : "London" } }
```

Query 1.2.2 > Query to find address of student with FIRST_NAME="Ankit"

```
db.STUDENT.find({"FIRST_NAME":"Ankit"}).pretty();
```

Output>

```
{ "_id" : 1, "FIRST_NAME" : "Ankit", "ADDRESS" : { "CITY" : "Delhi" } }
```

*2) (1-to-1) **One-to-One** Relationships with **Document Reference** >*

Create separate STUDENT and ADDRESS collections.

Where documents in STUDENT contain a reference to the ADDRESS document.

2.1) Create collections in MongoDB >

STEP 2.1.1) create and insert in ADDRESS collection >

```
db.ADDRESS.insert({
 "_id": 11,
 "CITY": "Delhi"
})
db.ADDRESS.insert({
 "_id": 12,
```

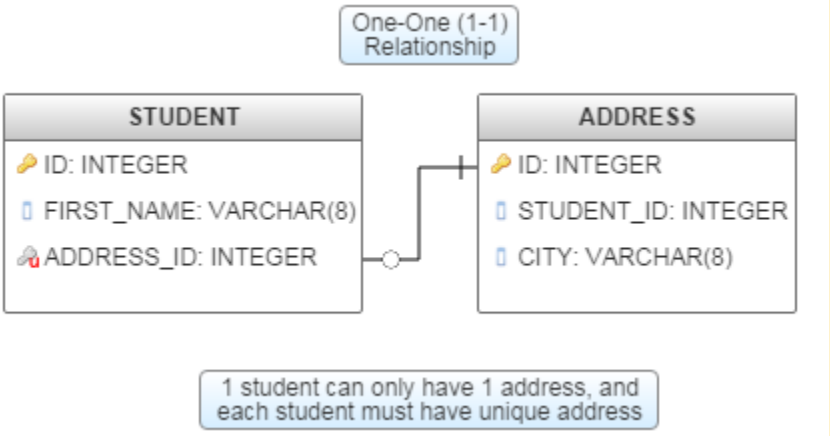
```
 "CITY": "London"
})
```

STEP 2.1.2) create and insert in STUDENT collection >

```
db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit",
 "ADDRESS_ID": {
    "$ref": "ADDRESS",
    "$id": 11,
    "$db": "mydb"
  }
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam",
 "ADDRESS_ID":  {
    "$ref": "ADDRESS",
    "$id": 12,
    "$db": "mydb"
  }
})
```

2.2) Now, let's see above **one-one** relationship of mongoDB collections in **RDBMS** (relational database) >

2.2.1) **One-One (1-1) Relationship -** Table structure in RDBMS >



2.2.2) Sql script to **create** above **tables** in RDBMS (in oracle) >

```
create table ADDRESS (ID number PRIMARY KEY,
                CITY varchar2(22) );

create table STUDENT (ID number PRIMARY KEY,
                FIRST_NAME varchar2(22),
                ADDRESS_ID number UNIQUE,
                FOREIGN KEY (ADDRESS_ID) REFERENCES ADDRESS (ID));
```

2.2.3) Let's see tables **after inserting data** in RDBMS >

```
select * from STUDENT;
```

	ID	FIRST_NAME	ADDRESS_ID
1	1	Ankit	11
2	2	Sam	12

```
select * from ADDRESS;
```

	ID	CITY
1	11	Delhi
2	12	London

Here, 1 student have 1 address, and each student have unique address.

2.3 > Now, let's read/query/find in above (2.1) MongoDB collections >

Query 2.3.1 > Query to show all students in MongoDB

```
db.STUDENT.find().pretty()
```

Output>

```
{ "_id" : 1, "FIRST_NAME" : "Ankit", "ADDRESS_ID" : 11 }
{ "_id" : 2, "FIRST_NAME" : "Sam", "ADDRESS_ID" : 12 }
```

Query 2.3.2 > Query to show all address in MongoDB

```
db.ADDRESS.find().pretty()
```

Output>

```
{ "_id" : 11, "CITY" : "Delhi" }
{ "_id" : 12, "CITY" : "London" }
```

IMPORTANT Query 2.3.3 > Query to Find address of student with **FIRST_NAME**="Ankit"

```
var student = db.STUDENT.findOne({"FIRST_NAME":"Ankit"})
var studentAddress = student.ADDRESS_ID
db[studentAddress.$ref].find({"_id":(studentAddress.$id)})
```

Output>

```
{ "_id" : 11, "CITY" : "Delhi" }
```

*3) Now let's cover above point (i.e. 2nd point) - (Inserting related documents in same collection) - **One-to-One** Relationships with **Document Reference** >*

3.1) create and insert in STUDENT collection (Also insert related data i.e. data (documents) of ADDRESS in STUDENT collection) >

```
db.STUDENT.insert({
 "_id": 11,
 "CITY": "Delhi"
})
db.STUDENT.insert({
 "_id": 12,
 "CITY": "London"
})


db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit",
 "ADDRESS_ID": 11
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam",
 "ADDRESS_ID":  12
})
```

4) Summary -

So in this mongoDB tutorial we learned **how can we create 1-1 (one to one) Relationship in MongoDB** *with* ***Embedded document*** *and* ***Document Reference*****.**

*1) (1-to-Many) **One-to-Many** Relationships with **Embedded documents** >*

Create STUDENT collections.

Embed the PHONE documents (completely) in the STUDENT document.
It helps in fetching all student and phone data in one query.

1.1) Create collection in MongoDB >

STEP 1.1.1) create and insert in STUDENT collection >

```
db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit",
 "PHONE": [
   {"PHONE_NUMBER": 1234},
   {"PHONE_NUMBER": 2345}
 ]
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam",
 "PHONE": [
   {"PHONE_NUMBER": 3456},
   {"PHONE_NUMBER": 4567}
 ]
})
```

When to use this approach?

When number of **transactions** are too high and need to be done atomically > This can be used when phone is fetched too frequently. i.e (Read and write operations are too high).

MongoDB does **not** support transactions on **multiple document**.But, in **MongoDB you can perform atomic operations on a single document.**

So, **while designing your database and collections** you must try and ensure that all the **related data (**as much as possible**) which is needed to be updated atomically must be placed in single document as embedded documents** (in form of nest arrays OR nested documents)

1.2 > Now, let's read/query/find in above MongoDB collection >

Query 1.2.1 > Query to show all students

```
db.STUDENT.find().pretty()
```

Output>

```
{
	"_id" : 1,
	"FIRST_NAME" : "Ankit",
	"PHONE" : [
		{
			"PHONE_NUMBER" : 1234
		},
		{
			"PHONE_NUMBER" : 2345
		}
	]
}
{
	"_id" : 2,
	"FIRST_NAME" : "Sam",
	"PHONE" : [
		{
			"PHONE_NUMBER" : 3456
		},
		{
			"PHONE_NUMBER" : 4567
		}
	]
}
```

Query 1.2.2 > Query to find all phones of student with FIRST_NAME="Ankit"

```
db.STUDENT.find({"FIRST_NAME":"Ankit"}).pretty()
```

Output>

```
{
	"_id" : 1,
	"FIRST_NAME" : "Ankit",
	"PHONE" : [
		{
			"PHONE_NUMBER" : 1234
		},
		{
			"PHONE_NUMBER" : 2345
		}
	]
}
```

Also please read related stuff : How to retrieve only specific element from an array in MongoDB

*2) (1-to-Many) **One-to-Many** Relationships with **Document References** >*

This is more **normalized** approach to model one to many relationship.
Most normalized data model, We exactly use this approach in RDBMS (See, below for RDBMS table diagram)

Create separate `STUDENT` and `PHONE` collections.

And, documents in `phone` contain a reference to the `student` document.

When to use this approach?

This can be used when data is huge.
And when read and write operations are not too high.

2.1) Create collections in MongoDB >

STEP 2.1.1) create and insert in STUDENT collection >

```
db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit"
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam"
})
```

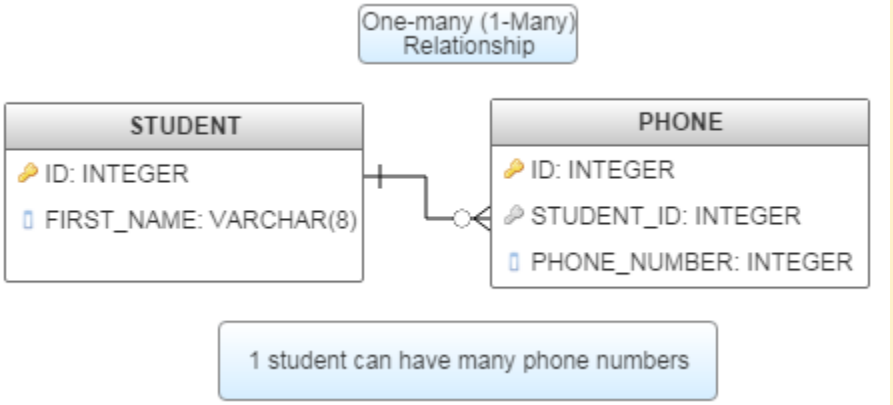
STEP 2.1.2) create and insert in PHONE collection >

```
db.PHONE.insert({
 "_id": 11,
 "PHONE_NUMBER": 1234,
 "STUDENT_ID":  {
    "$ref": "STUDENT",
    "$id": 1,
    "$db": "mydb"
  }
})
db.PHONE.insert({
 "_id": 12,
 "PHONE_NUMBER": 2345,
 "STUDENT_ID":  {
    "$ref": "STUDENT",
    "$id": 1,
    "$db": "mydb"
  }
})
db.PHONE.insert({
 "_id": 13,
 "PHONE_NUMBER": 3456,
 "STUDENT_ID":  {
    "$ref": "STUDENT",
    "$id": 2,
    "$db": "mydb"
  }
})
db.PHONE.insert({
 "_id": 14,
 "PHONE_NUMBER": 4567,
 "STUDENT_ID":  {
    "$ref": "STUDENT",
    "$id": 2,
    "$db": "mydb"
```

```
  }
})
```

2.2) Now, let's see above **one-many** relationship of mongoDB collections in **RDBMS** (relational database) >

2.2.1) **One-Many (1-Many) Relationship -** Table structure in RDBMS >



2.2.2) Sql script to **create** above **tables** in RDBMS (in oracle) >

```
create table STUDENT (ID number PRIMARY KEY,
                      FIRST_NAME varchar2(22));

create table PHONE (ID number PRIMARY KEY,
                    STUDENT_ID number,
                    PHONE_NUMBER number,
                    FOREIGN KEY (STUDENT_ID) REFERENCES STUDENT (ID));
```

2.2.3) Let's see tables **after inserting data** in RDBMS >

```
select * from STUDENT;
```

	ID	FIRST_NAME
1	1	Ankit
2	2	Sam

```
select * from PHONE;
```

	ID	STUDENT_ID	PHONE_NUMBER
1	11	1	1234
2	12	1	2345
3	13	2	3456
4	14	2	4567

Here, one student have many phone numbers.

2.3 > Now, let's read/query/find in above (2.1) MongoDB collections >

Query 2.3.1 > Query to show all students

```
db.STUDENT.find().pretty()
```

Output>

```
{ "_id" : 1, "FIRST_NAME" : "Ankit" }
{ "_id" : 2, "FIRST_NAME" : "Sam" }
```

Query 2.3.2 > Query to show all phones

```
db.PHONE.find().pretty()
```

Output>

```
{
	"_id" : 11,
	"PHONE_NUMBER" : 1234,
	"STUDENT_ID" : DBRef("STUDENT", 1, "mydb")
}
{
	"_id" : 12,
	"PHONE_NUMBER" : 2345,
	"STUDENT_ID" : DBRef("STUDENT", 1, "mydb")
}
{
	"_id" : 13,
	"PHONE_NUMBER" : 3456,
	"STUDENT_ID" : DBRef("STUDENT", 2, "mydb")
}
{
	"_id" : 14,
	"PHONE_NUMBER" : 4567,
	"STUDENT_ID" : DBRef("STUDENT", 2, "mydb")
}
```

IMPORTANT Query 2.3.3 > Query to **find phone_number=1234 belong to which student >**

```
var phone = db.PHONE.findOne({"PHONE_NUMBER":1234})
var student = phone.STUDENT_ID
db[student.$ref].find({"_id":(student.$id)})
```

Output>

```
{ "_id" : 1, "FIRST_NAME" : "Ankit" }
```

3) Another way to model One-Many relationship of data - (You may use any of the **approaches** depending on your requirement)

Here we use, one-to-Many Relationships with **Embedded References.**

Here `student` is referencing `phone`.
Embed the `phone` (only phone_id) in the `student` data >

3.1) Create collections in MongoDB >

STEP 3.1.1) create and insert in `PHONE` collection >

```
db.PHONE.insert({
 "_id": 11,
 "PHONE_NUMBER": 1234,
})
db.PHONE.insert({
 "_id": 12,
 "PHONE_NUMBER": 2345,
})
db.PHONE.insert({
 "_id": 13,
 "PHONE_NUMBER": 3456,
})
```

```
db.PHONE.insert({
 "_id": 14,
 "PHONE_NUMBER": 4567,
})
```

STEP 3.1.2) create and insert in STUDENT collection >

```
db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit",
 "PHONE_ID": [
   11,
   12
 ]
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam",
 "PHONE_ID": [
   13,
   14
 ]
})
```

*4) Now let's cover above point (i.e. 2nd point) - (Inserting related documents in same collection) - **One-to-Many** Relationships with **Document Reference** >*

4.1) create and insert in STUDENT collection (Also insert related data i.e. data (documents) of PHONE in STUDENT collection) >

```
db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit"
})
```

```
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam"
})


db.STUDENT.insert({
 "_id": 11,
 "PHONE_NUMBER": 1234,
 "STUDENT_ID":  1
})
db.STUDENT.insert({
 "_id": 12,
 "PHONE_NUMBER": 2345,
 "STUDENT_ID":  1
})
db.STUDENT.insert({
 "_id": 13,
 "PHONE_NUMBER": 3456,
 "STUDENT_ID":  2
})
db.STUDENT.insert({
 "_id": 14,
 "PHONE_NUMBER": 4567,
 "STUDENT_ID":  2
})
```

5) Summary -

So in this mongoDB tutorial we learned how to create **1 - many Relationship in MongoDB - Multiple table - one to many.**

*1) (Many-to-1) **Many-to-One** Relationships with **Embedded document** >*

Embed the CLASS's documents in the student documents.
It helps in fetching all student and CLASS data in one query.

But, this is **not** a **good approach**, here you can see too much **redundant** data (_id = 1 and 2 are having same class document, same is the case with _id = 3 and 4), you can go for it till the data is less, as the data grows you will need to **follow some normalized approach as shown below** (where documents in student contain a reference to the CLASS document).

1.1) Create collection in MongoDB >

STEP 1.1.1) create and insert in STUDENT collection >

```
db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit",
 "CLASS": {"CLASS_NAME": "FirstClass"}
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam",
 "CLASS": {"CLASS_NAME": "FirstClass"}
})
db.STUDENT.insert({
 "_id": 3,
 "FIRST_NAME": "Neha",
 "CLASS": {"CLASS_NAME": "SecondClass"}
})
db.STUDENT.insert({
 "_id": 4,
 "FIRST_NAME": "Amy",
 "CLASS": {"CLASS_NAME": "SecondClass"}
})
```

1.2 > Now, let's read/query/find in above MongoDB collection >

Query 1.2.1 > Query to show all students

```
db.STUDENT.find().pretty()
```

Output>

```
{
	"_id" : 1,
	"FIRST_NAME" : "Ankit",
	"CLASS" : {		"CLASS_NAME" : "FirstClass"	}
}
{
	"_id" : 2,
	"FIRST_NAME" : "Sam",
	"CLASS" : {		"CLASS_NAME" : "FirstClass"	}
}
{
	"_id" : 3,
	"FIRST_NAME" : "Neha",
	"CLASS" : {		"CLASS_NAME" : "SecondClass"	}
}
{
	"_id" : 4,
	"FIRST_NAME" : "Amy",
	"CLASS" : {		"CLASS_NAME" : "SecondClass"	}
}
```

Query 1.2.2 > Query to Find student with FIRST_NAME="Ankit"

```
db.STUDENT.find({"FIRST_NAME":"Ankit"}).pretty()
```

Output>

```
{
	"_id" : 1,
	"FIRST_NAME" : "Ankit",
	"CLASS" : {
```

```
            "CLASS_NAME" : "FirstClass"
    }
}
```

*2) (Many-to-1) **Many-to-One** Relationships with **Document Reference (BEST APPROACH)** >*

This most **normalized** approach to model many to one relationship.

This is a **good and best approach**, here you can see no redundant data (_id = 1 and 2 are having reference of same class document, same is the case with _id = 3 and 4), you can go for when data is huge, it's the most normalized approach to model data in many-to-one relationship.
Create CLASS collection and student collection.

Where documents in student contain a reference to the CLASS document.

2.1) Create collections in MongoDB >

STEP 2.1.1) create and insert in CLASS collection >

```
db.CLASS.insert({
 "_id": 11,
 "CLASS_NAME": "FirstClass"
})
db.CLASS.insert({
 "_id": 12,
 "CLASS_NAME": "SecondClass"
})
```

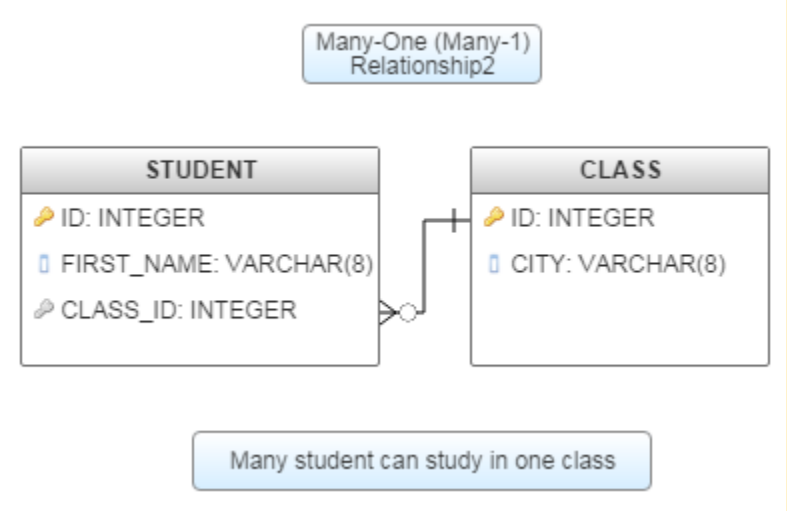
STEP 2.1.2) create and insert in STUDENT collection >

```
db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit",
```

```
 "CLASS_ID":  {
    "$ref": "CLASS",
    "$id":  11,
    "$db": "mydb"
  }
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam",
 "CLASS_ID":  {
    "$ref": "CLASS",
    "$id":  11,
    "$db": "mydb"
  }
})
db.STUDENT.insert({
 "_id": 3,
 "FIRST_NAME": "Neha",
 "CLASS_ID":  {
    "$ref": "CLASS",
    "$id":  12,
    "$db": "mydb"
  }
})
db.STUDENT.insert({
 "_id": 4,
 "FIRST_NAME": "Amy",
 "CLASS_ID":  {
    "$ref": "CLASS",
    "$id":  12,
    "$db": "mydb"
  }
})
```

2.2) Now, let's see above **many-one** relationship of mongoDB collections in **RDBMS** (relational database) >

2.2.1) **Many-one (Many-1) Relationship -** Table structure in RDBMS >



2.2.2) Sql script to **create** above **tables** in RDBMS (in oracle) >

```
create table CLASS (ID number PRIMARY KEY,
                    CLASS_NAME varchar2(22) );

create table STUDENT (ID number PRIMARY KEY,
                      FIRST_NAME varchar2(22),
                      CLASS_ID number,
                      FOREIGN KEY (CLASS_ID) REFERENCES CLASS (ID));
```

2.2.3) Let's see tables **after inserting data** in RDBMS >

```
select * from STUDENT;
```

	ID	FIRST_NAME	CLASS_ID
1	1	Ankit	11
2	2	Sam	11
3	3	Neha	12
4	4	Amy	12

```
select * from CLASS;
```

	ID	CLASS_NAME
1	11	FirstClass
2	12	SecondClass

Here, Many students study in one class.

2.3 > Now, let's read/query/find in above (2.1) MongoDB collections >

Query 2.3.1 > Query to show all students

```
db.STUDENT.find()
```

Output>

```
{ "_id" : 1, "FIRST_NAME" : "Ankit", "CLASS_ID" : DBRef("CLASS", 11, "mydb") }
{ "_id" : 2, "FIRST_NAME" : "Sam", "CLASS_ID" : DBRef("CLASS", 11, "mydb") }
{ "_id" : 3, "FIRST_NAME" : "Neha", "CLASS_ID" : DBRef("CLASS", 12, "mydb") }
{ "_id" : 4, "FIRST_NAME" : "Amy", "CLASS_ID" : DBRef("CLASS", 12, "mydb") }
```

Query 2.3.2 > Query to find and show all CLASS

```
db.CLASS.find().pretty()
```

Output>

```
{ "_id" : 11, "CLASS_NAME" : "FirstClass" }
{ "_id" : 12, "CLASS_NAME" : "SecondClass" }
```

Query 2.3.3 > Query to **find class of student with FIRST_NAME**=”Ankit”

```
var student = db.STUDENT.findOne({"FIRST_NAME":"Ankit"})
var studentClass = student.CLASS_ID
db[studentClass.$ref].find({"_id":(studentClass.$id)})
```

Output>

```
{ "_id" : 11, "CLASS_NAME" : "FirstClass" }
```

*3) Now let’s cover above point (i.e. 2nd point) - (Inserting related documents in same collection) - **Many-to-One** Relationships with **Document Reference** >*

3.1) create and insert in STUDENT collection (Also insert related data i.e. data (documents) of CLASS in STUDENT collection) >

```
db.STUDENT.insert({
 "_id": 11,
 "CLASS_NAME": "FirstClass"
})
db.STUDENT.insert({
 "_id": 12,
 "CLASS_NAME": "SecondClass"
})


db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit",
 "CLASS_ID": 11
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam",
 "CLASS_ID":  11
})
db.STUDENT.insert({
 "_id": 3,
 "FIRST_NAME": "Neha",
 "CLASS_ID":  12
})
db.STUDENT.insert({
 "_id": 4,
 "FIRST_NAME": "Amy",
 "CLASS_ID":  12
})
```

4) Summary -

So in this MongoDB tutorial we learned with example how to create, manage and establish Many - 1 relationship in MongoDB.

A) (Many-to-Many) ***Many-to-Many Relationships with Document Reference***

This is the most normalized and best form to represent many to many relationship data model.

Create student collection and COURSE collection.

Then create **STUDENT_COURSE** collection which contains reference to student and COURSE collection.

1) Create collections in MongoDB >

STEP 1.1) create and insert in STUDENT collection >

```
db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit"
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam"
})
```

STEP 1.2) create and insert in COURSE collection >

```
db.COURSE.insert({
 "_id": 11,
 "COURSE_NAME": "Hindi"
})
db.COURSE.insert({
 "_id": 12,
 "COURSE_NAME": "English"
})
```

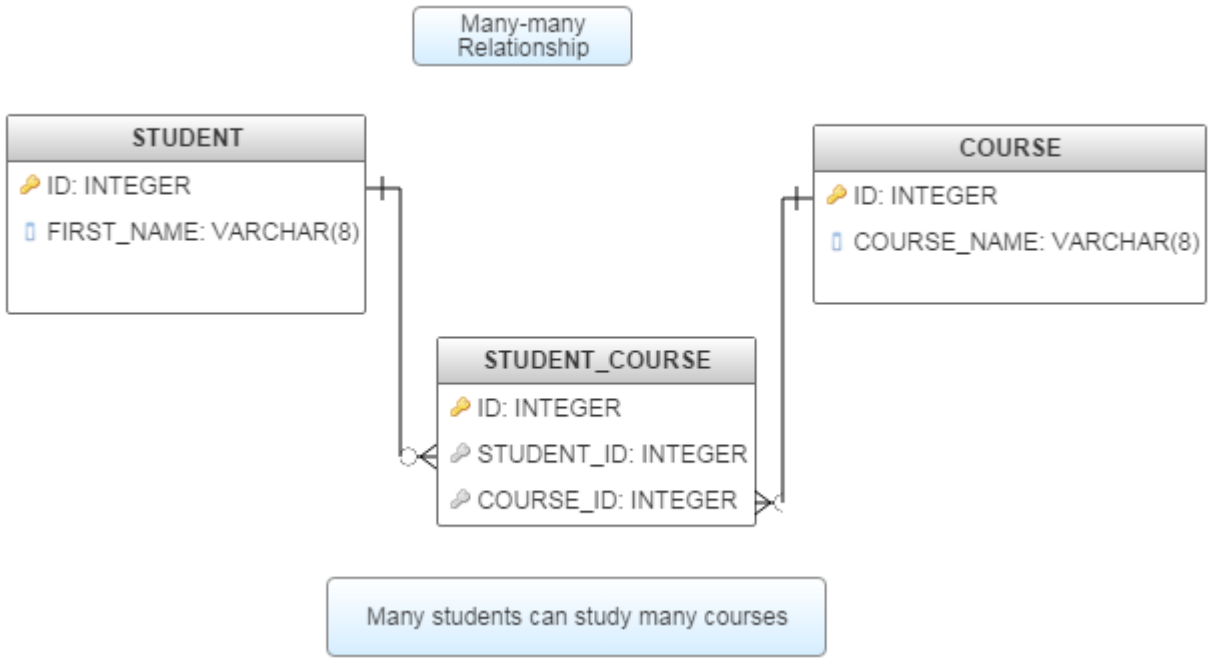
STEP 1.3) create and insert in **STUDENT_COURSE** collection >

```
db.STUDENT_COURSE.insert({
 "_id": 21,
 "STUDENT_ID": 1 ,
 "COURSE_ID": 11
})
```

```
db.STUDENT_COURSE.insert({
 "_id": 22,
 "STUDENT_ID": 1 ,
 "COURSE_ID": 12
})
db.STUDENT_COURSE.insert({
 "_id": 23,
 "STUDENT_ID": 2 ,
 "COURSE_ID": 11
})
db.STUDENT_COURSE.insert({
 "_id": 24,
 "STUDENT_ID": 2 ,
 "COURSE_ID": 12,
})
```

2) Now, let's see above **many-many** relationship of mongoDB collections in **RDBMS** (relational database) >

2.1) **Many-Many Relationship -** Table structure in RDBMS >



2.2) Sql script to **create** above **tables** in RDBMS (in oracle) >

```
create table STUDENT (ID number PRIMARY KEY,
                FIRST_NAME varchar2(22));

create table COURSE (ID number PRIMARY KEY,
                COURSE_NAME varchar2(22) );

create table STUDENT_COURSE (ID number PRIMARY KEY,
                STUDENT_ID number,
                COURSE_ID number,
                FOREIGN KEY (STUDENT_ID) REFERENCES STUDENT (ID),
                FOREIGN KEY (COURSE_ID) REFERENCES COURSE (ID) );
```

2.2.3) Let's see tables **after inserting data** in RDBMS >

```
select * from STUDENT;
```

	ID	FIRST_NAME
1	1	Ankit
2	2	Sam

```
select * from COURSE;
```

	ID	COURSE_NAME
1	11	Hindi
2	12	English

```
select * from STUDENT_COURSE;
```

	ID	STUDENT_ID	COURSE_ID
1	21	1	11
2	22	1	12
3	23	2	11
4	24	2	12

Here, Many students study many courses.

3 > Now, let's read/query/find in above (2.1) MongoDB collections >

Query 3.1 > Query to find and show all students

```
db.STUDENT.find().pretty()
```

Output>

```
{ "_id" : 1, "FIRST_NAME" : "Ankit" }
{ "_id" : 2, "FIRST_NAME" : "Sam" }
```

Query 3.2 > Query to find and show all students

```
db.COURSE.find().pretty()
```

Output>

```
{ "_id" : 11, "COURSE_NAME" : "Hindi" }
{ "_id" : 12, "COURSE_NAME" : "English" }
```

Query 3.3 > Query to find and show all student_courses

```
db.STUDENT_COURSE.find().pretty()
```

Output>

```
{ "_id" : 21, "STUDENT_ID" : 1, "COURSE_ID" : 11 }
{ "_id" : 22, "STUDENT_ID" : 1, "COURSE_ID" : 12 }
{ "_id" : 23, "STUDENT_ID" : 2, "COURSE_ID" : 11 }
{ "_id" : 24, "STUDENT_ID" : 2, "COURSE_ID" : 12 }
```

B) *Now let's cover above point - (Inserting related documents in same collection) -* ***Many-to-Many*** *Relationships with* ***Document Reference*** *>*

create and insert in STUDENT collection (Also insert related data i.e. data (documents) of COURSE and STUDENT_COURSE in STUDENT collection) >

```
db.STUDENT.insert({
 "_id": 1,
 "FIRST_NAME": "Ankit"
})
db.STUDENT.insert({
 "_id": 2,
 "FIRST_NAME": "Sam"
})
```

```
db.STUDENT.insert({
 "_id": 11,
 "COURSE_NAME": "Hindi"
})
db.STUDENT.insert({
 "_id": 12,
 "COURSE_NAME": "English"
})


db.STUDENT.insert({
 "_id": 21,
 "STUDENT_ID": 1 ,
 "COURSE_ID": 11
})
db.STUDENT.insert({
 "_id": 22,
 "STUDENT_ID": 1 ,
 "COURSE_ID": 12
})
db.STUDENT.insert({
 "_id": 23,
 "STUDENT_ID": 2 ,
 "COURSE_ID": 11
})
db.STUDENT.insert({
 "_id": 24,
 "STUDENT_ID": 2 ,
 "COURSE_ID": 12,
})
```

C) *Summary -*

So in this mongoDB tutorial we learned how to create and manage Many-Many relationship in MongoDB**.**