جامعة طرابلس

كلية تقنية المعلومات

قواعد البيانات النقالة والغير متجانسة

Heterogeneous and Mobile Databases

ITMC322

أستاذ المادة / محمد أوهيبة

المحاضرة السادسة

Review of topics

- ***Using ranges in your queries***
- ***Using logical operators to query data***
- ***Updating documents***
- ***Deleting data***
- ***Beyond basic data types***
  - ***Arrays***
  - ***Embedded documents***
- ***Some useful functions***
- ***Securing database access***

Using ranges in your queries

- queries will use some functions to restrict the range of the returned data, which is done in most SQL dialects and languages with the > and < or = operators.

- The equivalent operators in MongoDB terms are `$gt`, `$gte`, `$lt`, and `$lte`. Here is how to find users whose age is greater than *40* using the `$gt` operator:

```
> db.users.find({ age: { $gt: 40 } })

 { "_id" : ObjectId("5506d5988d7bd8471669e675"), "name" :
 "Ahmed", "age" : 44, "phone" : "092-456-789" }
```

- The `$gte` operator, on the other hand, is able to select keys that are greater than or equal (>=) to the one specified:

```
 > db.users.find({ age: { $gte: 32 } })

 { "_id" : ObjectId("5506d5988d7bd8471669e675"), "name" :
 "Ahmed", "age" : 44, "phone" : "092-456-789" }

 { "_id" : ObjectId("5506eea18d7bd8471669e676"), "name" :
 "Ali", "age" : 32, "phone" : "091-444-333" }
```

Using ranges in your queries(Cont.)

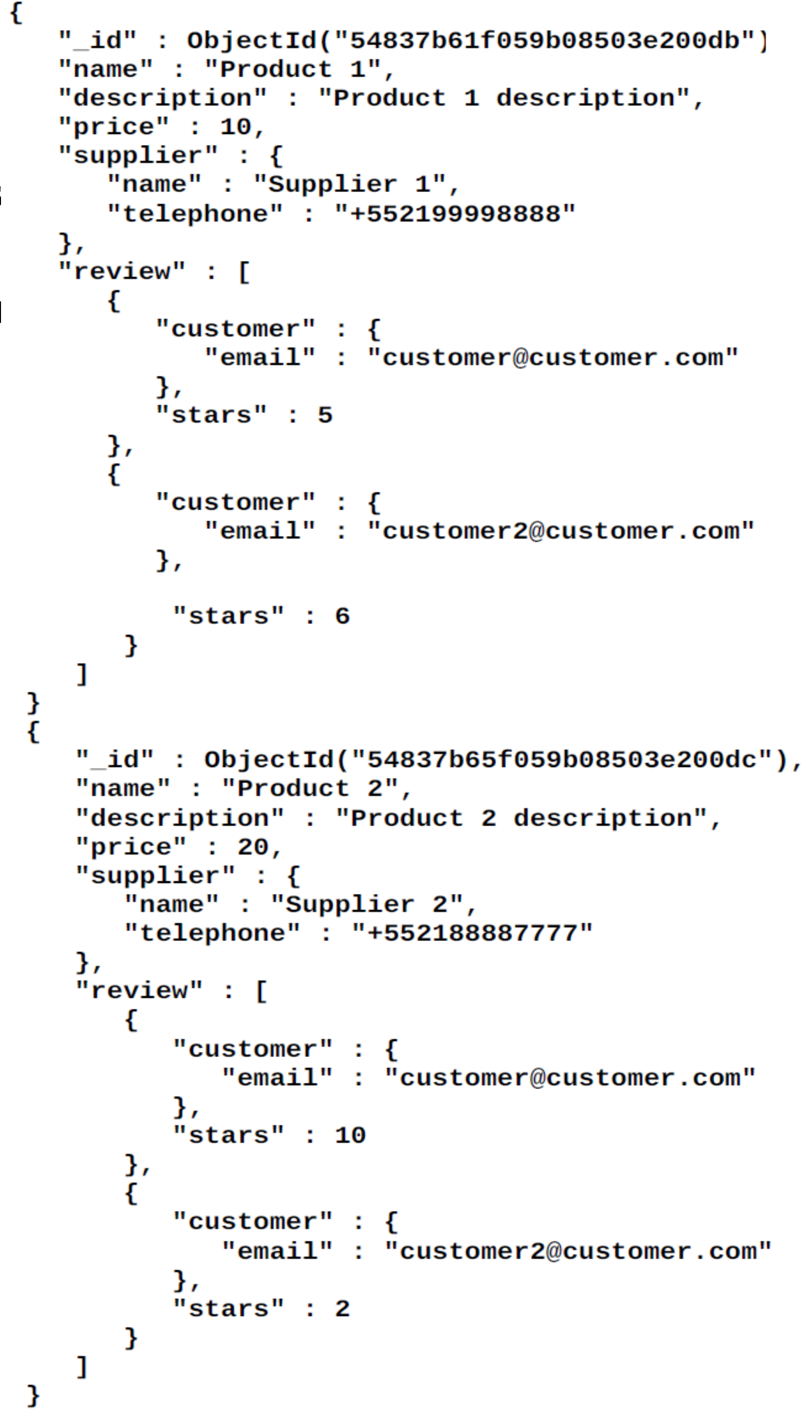
- The `$lt` and `$lte` operators, on the other hand, allow you to select keys which are smaller and smaller/equal to the value specified.

- With the $lt operator, it’s possible to search for values that are inferior to the requested value in the query. The query **db.products.find({price: {$lt: 20}})** will return:

```
{
   "_id" : ObjectId("54837b61f059b08503e200db"),
   "name" : "Product 1",
   "description" : "Product 1 description",
   "price" : 10,
   "supplier" : {
      "name" : "Supplier 1",
      "telephone" : "+552199998888"
   },
   "review" : [
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 5
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 6
      }
   ]
}
```

Using ranges in your

- The $lte operator searches for values that are les value in the query. If we execute the query **db.products.find({price: {$lte:20}})**, it will retur



```
{
    "_id" : ObjectId("54837b61f059b08503e200db")
    "name" : "Product 1",
    "description" : "Product 1 description",
    "price" : 10,
    "supplier" : {
        "name" : "Supplier 1",
        "telephone" : "+552199998888"
    },
    "review" : [
        {
            "customer" : {
                "email" : "customer@customer.com"
            },
            "stars" : 5
        },
        {
            "customer" : {
                "email" : "customer2@customer.com"
            },

             "stars" : 6
         }
     ]
 }
 {
     "_id" : ObjectId("54837b65f059b08503e200dc"),
     "name" : "Product 2",
     "description" : "Product 2 description",
     "price" : 20,
     "supplier" : {
         "name" : "Supplier 2",
         "telephone" : "+552188887777"
     },
     "review" : [
         {
             "customer" : {
                 "email" : "customer@customer.com"
             },
             "stars" : 10
         },
         {
             "customer" : {
                 "email" : "customer2@customer.com"
             },
             "stars" : 2
         }
     ]
 }
```

Using ranges in your queries(Cont...)

- The $in operator is able to search any document where the value of a field equals a value that is specified in the requested array in the query. The execution of the query **db.products.find({price:{$in: [5, 10, 15]}})** will return:

```
{
   "_id" : ObjectId("54837b61f059b08503e200db"),
   "name" : "Product 1",
   "description" : "Product 1 description",
   "price" : 10,
   "supplier" : {
      "name" : "Supplier 1",
      "telephone" : "+552199998888"
   },
   "review" : [
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 5
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 6
      }
   ]
```

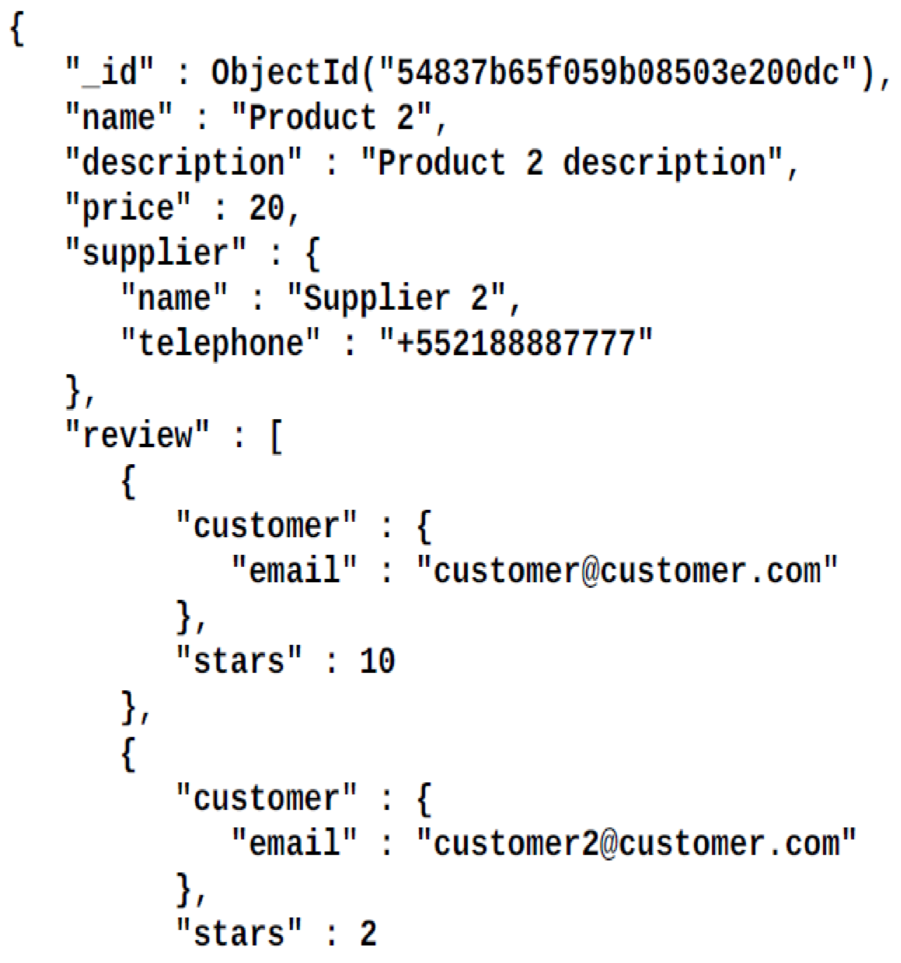
Using ranges in your queries(Cont....)

- The $nin operator will match values that are not included in the specified array. The execution of the **db.products.find({price:{$nin: [10, 20]}})** query will produce:

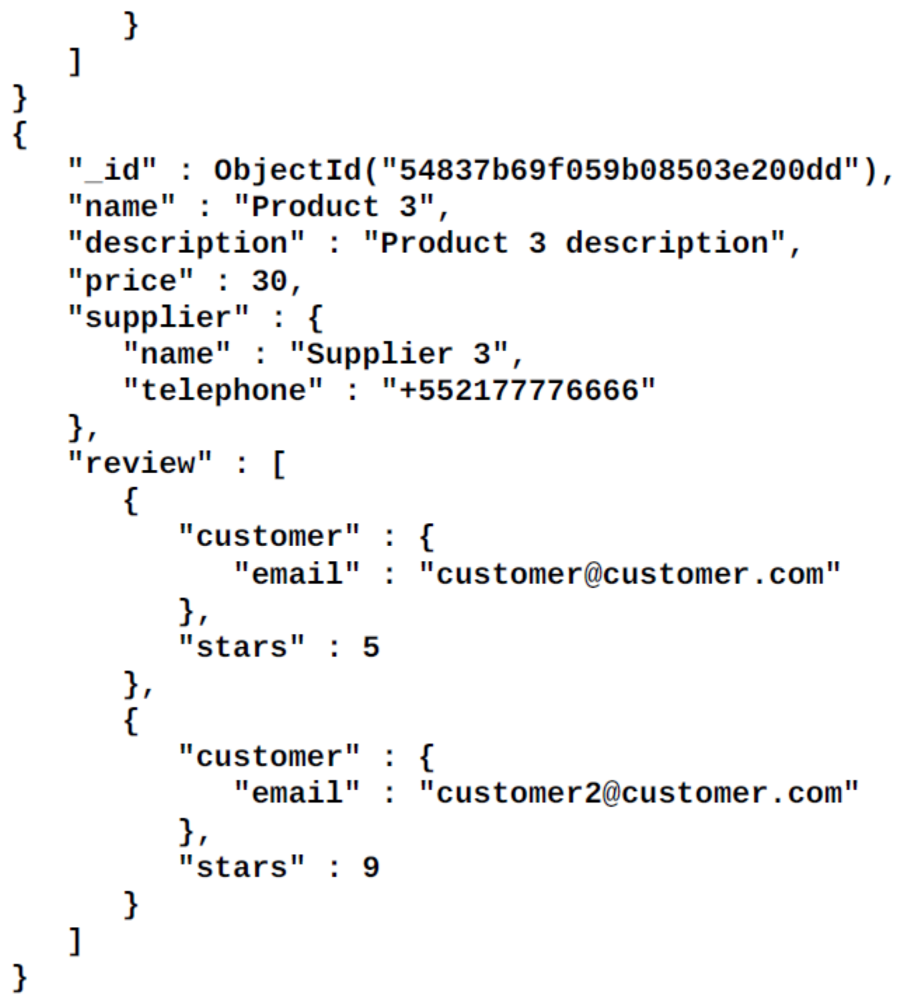
```
{
   "_id" : ObjectId("54837b69f059b08503e200dd"),
   "name" : "Product 3",
   "description" : "Product 3 description",
   "price" : 30,
   "supplier" : {
      "name" : "Supplier 3",
      "telephone" : "+552177776666"
   },
   "review" : [
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 5
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 9
      }
   ]
}
```

Using ranges in your queries(Cont.....)

- The $ne operator will match any values that are not equal to the specified value in the query. The execution of the **db.products.find({name: {$ne: "Product 1"}})** query will produce:



```
{
   "_id" : ObjectId("54837b65f059b08503e200dc"),
   "name" : "Product 2",
   "description" : "Product 2 description",
   "price" : 20,
   "supplier" : {
      "name" : "Supplier 2",
      "telephone" : "+552188887777"
   },
   "review" : [
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 10
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 2
```

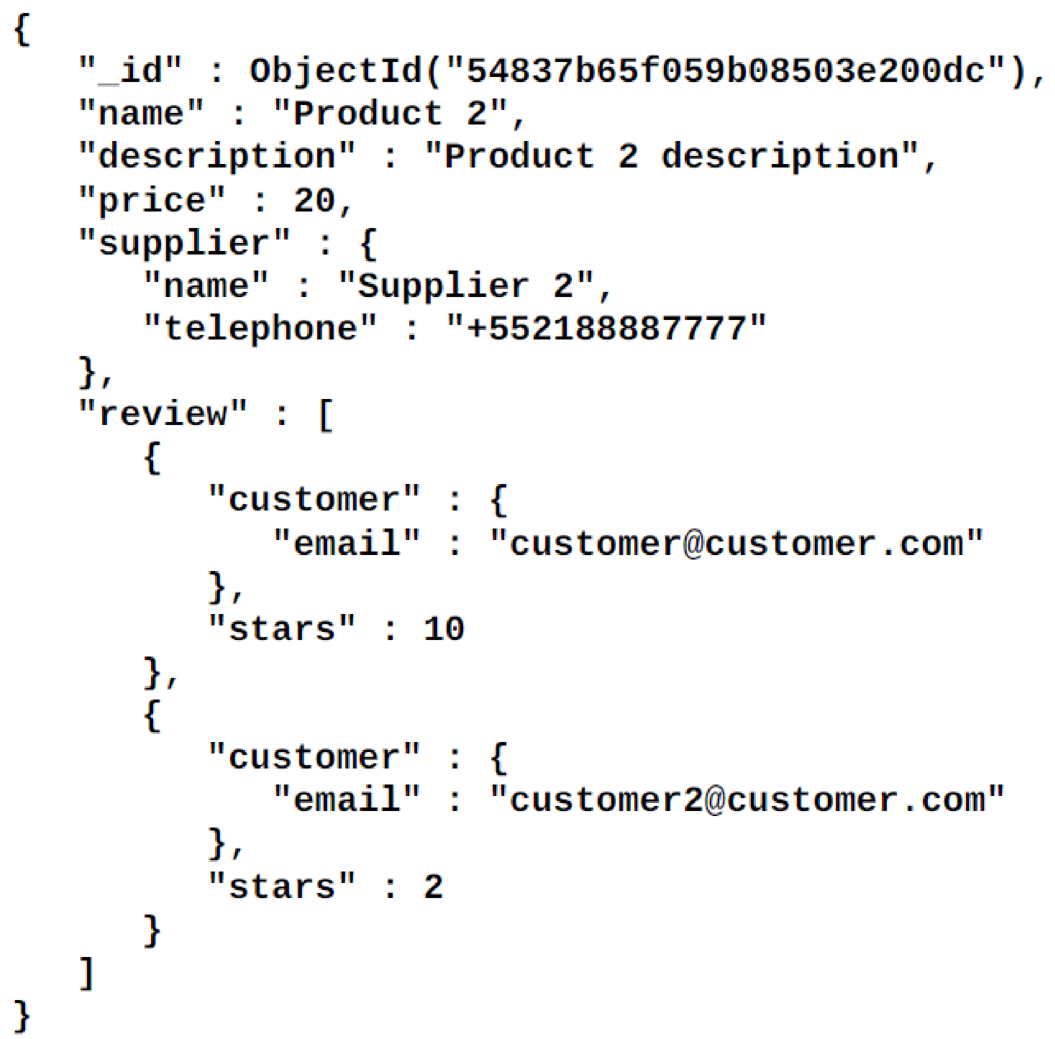


```
      }
   ]
}
{
   "_id" : ObjectId("54837b69f059b08503e200dd"),
   "name" : "Product 3",
   "description" : "Product 3 description",
   "price" : 30,
   "supplier" : {
      "name" : "Supplier 3",
      "telephone" : "+552177776666"
   },
   "review" : [
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 5
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 9
      }
   ]
}
```

Using logical operators to query data

- Logical operators are how we define the logic between values in MongoDB.
- These are derived from Boolean algebra, and the truth value of a Boolean value can be either true or false.
- The logical operators in MongoD

- The $and operator will make a lo
  will return the values that match
  The execution of the **db.products**
  query will produce:



```
{
    "_id" : ObjectId("54837b65f059b08503e200dc"),
    "name" : "Product 2",
    "description" : "Product 2 description",
    "price" : 20,
    "supplier" : {
        "name" : "Supplier 2",
        "telephone" : "+552188887777"
    },
    "review" : [
        {
            "customer" : {
                "email" : "customer@customer.com"
            },
            "stars" : 10
        },
        {
            "customer" : {
                "email" : "customer2@customer.com"
            },
            "stars" : 2
        }
    ]
}
```

Using logical operators to query data (Cont.)

The $or operator will make a logical OR operation in an expressions array, and will return all the values that match either of the specified criteria. The execution of the **db.products.find({$or: [{price: {$gt: 50}}, {name: "Product 3"}]})** query will produce:

```
   {
      "_id" : ObjectId("54837b69f059b08503e200dd"),
      "name" : "Product 3",
      "description" : "Product 3 description",
      "price" : 30,
      "supplier" : {
         "name" : "Supplier 3",
         "telephone" : "+552177776666"
      },
      "review" : [
         {
            "customer" : {
               "email" : "customer@customer.com"


         },
         "stars" : 5
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 9
      }
   ]
}
```

Using logical operators to query data (Cont..)

- The $not operator inverts the query effect and returns the values that do not match the specified operator expression. It is used to negate any operation. The execution of the **db.products.find({price: {$not: {$gt: 10}}})** query will produce:

```
{
   "_id" : ObjectId("54837b61f059b08503e200db"),
   "name" : "Product 1",
   "description" : "Product 1 description",
   "price" : 10,
   "supplier" : {
      "name" : "Supplier 1",
      "telephone" : "+552199998888"
   },
   "review" : [
      {
         "customer" : {
            "email" : "customer@customer.com"
         },
         "stars" : 5
      },
      {
         "customer" : {
            "email" : "customer2@customer.com"
         },
         "stars" : 6
      }
   ]
}
```

Using logical operators to query data (Cont...)

- The $nor operator will make a logical NOR operation in an expressions array, and will return all the values that fail to match all the specified expressions in the array. The execution of the **db.products.find({$nor:[{price:{$gt: 35}}, {price: {$lte: 20}}]})** query will produce:

```
{
   "_id" : ObjectId("54837b69f059b08503e200dd"),
   "name" : "Product 3",
   "description" : "Product 3 description",
   "price" : 30,
   "supplier" : {
      "name" : "Supplier 3",
      "telephone" : "+552177776666"
   },

    "review" : [
       {
          "customer" : {
             "email" : "customer@customer.com"
          },
          "stars" : 5
       },
       {
          "customer" : {
             "email" : "customer2@customer.com"
          },
          "stars" : 9
       }
    ]
  }
```

Updating documents

- In order to update an existing document, you need to provide two arguments:
- The document to update .
- How the selected documents should be modified .

- Example : supposing that you wanted to change the key `age` for the user `Ali` to be `39`:

```
> db.users.update({name: "owen"}, {$set: {"age": 39}})
  WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

- The outcome of the statement informs us that the update matched one document which was modified. A `find` issued on the `users` collection reveals that the change has been applied:

Be aware that executing an update without the `$set` operator won't update the fields but replace the whole document, while preserving the `_id` field.

```
> db.users.find()

{ "_id" : ObjectId("5506d5988d7bd8471669e675"), "name" : “Ahmed",
"age" : 44, "phone" : “092-456-789" }

{ "_id" : ObjectId("5506eea18d7bd8471669e676"), "name" : “Ali",
"age" : 39, "phone" : “091-444-333" }
```

Updating documents (Cont.)

- The update supports an additional option, which can be used to perform a more complex logic. For example, what if you wanted to update the record if it exists, and create it if it doesn't? This is called upsert and can be achieved by setting the `upsert` option to `true`, as in the following command line:

```
> db.users.update({user: "frank"}, {age: 40},{ upsert: true} )
   WriteResult({
   "nMatched" : 0,
   "nUpserted" : 1,
   "nModified" : 0,
   "_id" : ObjectId("55082f5ea30be312eb167fcb")
   })
```

- As you can see from the output, an upsert has been executed and a document with the `age` key has been added:

```
> db.users.find()
{ "_id" : ObjectId("5506d5988d7bd8471669e675"), "name" : "Ahmed",
"age" : 44, "phone" : "092-456-789" }
{ "_id" : ObjectId("5506eea18d7bd8471669e676"), "name" : "Ali",
"age" : 39, "phone" : "091-444-333" }
{ "_id" : ObjectId("55082f5ea30be312eb167fcb"), "age" : 40 }
```

Updating documents (Cont..)

- you can remove a single key from your collection by using the `$unset` option.

```
db.users.update({name: “Ali"}, {$unset : { "age" : 1} })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

- Executing the `find` on our collection confirms the update:

```
> db.users.find()
{ "_id" : ObjectId("5506d5988d7bd8471669e675"), "name" : “Ahmed",
"age" : 44, "phone" : “092-456-789" }
{ "_id" : ObjectId("5506eea18d7bd8471669e676"), "name" : “Ali",
"phone" : “091-444-333" }
{ "_id" : ObjectId("55082f5ea30be312eb167fcb"), "age" : 40 }
```

- The opposite of the `$unset` operator is `$push`, which allows you to append a value to a specified field. So here is how you can restore the `age` key for the user `owen`:

```
> db.users.update({name: “Ali"}, {$push : { "age" : 39} })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Deleting data

- To delete a whole set of documents, then you can use the `remove` operator. When used without any parameter, it is equivalent to the TRUNCATE command in SQL terms:

```
> db.users.remove()
```

- To be more selective when deleting documents as you might need to remove just a set of documents matching one or more conditions. For example, here is how to remove users older than `40`:

```
> db.users.remove({ "age": { $gt: 40 } })
  WriteResult({ "nRemoved" : 1 })
```

- Just like the TRUNCATE statement in SQL, it just removes documents from a collection. If you want to delete the collection, then you need to use the **drop()** method, which deletes the whole collection structure, including any associated index:

```
> db.users.drop()
```

Beyond basic data types

- Although the basic data types we have used so far will be fine for most use cases, there are a couple of additional types that are crucial to most applications, especially when mapping Mongo types to a language driver such as a Mongo driver for Java.

- **Arrays**

  MongoDB has a rich query language that supports storing and accessing documents as arrays. One of the great things about arrays in documents is that MongoDB understands their structure and knows how to reach inside arrays to perform operations on their content.

  - creating a couple of documents containing an array of items:

```
> db.restaurant.insert({"menu" : ["bread", "pizza", "coke"]})
  WriteResult({ "nInserted" : 1 })

> db.restaurant.insert({"menu" : ["bread", "omelette", "sprite"]})
      WriteResult({ "nInserted" : 1 })
```

  - query on the array selection to find the menu, which includes pizza:

```
> db.restaurant.find({"menu" : "pizza"})

{ "_id" : ObjectId("550abbfe89ef057ee0671650"), "menu" : [
"bread","pizza", "coke" ] }
```

Beyond basic data types (Cont.)

- to match arrays using more than one element, then you can use **`$all`**.
-

```
> db.restaurant.find({"menu" : {$all : ["pizza", "coke"]}})

{ "_id" : ObjectId("550abbfe89ef057ee0671650"), "menu" : [
"bread", "pizza", "coke" ] }
```

- **Embedded documents**
- You can use a document as a value for a key. This is called an embedded document.

- Embedded documents can be used to organize data in a more natural way than just a flat structure of key-value pairs. This matches well with most object-oriented languages, which holds a reference to another structure in their class.

Beyond basic data types (Cont..)

- defining a structure, which is assigned to a variable in the mongo shell:

```
x = {
    "_id":1234,
    "owner":"Frank's Car",
    "cars":[
    {
    "year":2011,
    "model":"Ferrari",
    price:250000
    },
    {
    "year":2013,
    "model":"Porsche",
    price:250000
    }
    ]
  }
```

- Since the Mongo shell is a JavaScript interface, it is perfectly fine to write something like the preceding code and even use functions in order to enhance objects in the shell. Having defined our variable, we can insert it into the `cars` collection as follows: **`> db.cars.insert(x);`**

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

Beyond basic data types (Cont...)

- We can query our subdocument by using the dot notation. For example, we can choose the list of cars whose model is `Ferrari` by using the `cars.model` criteria:

```
> db.cars.find( { "cars.model": "Ferrari" }).pretty()
{
"_id" : 1234,
"owner" : "Frank's Car",
"cars" : [
       {
       "year" : 2011,
       "model" : "Ferrari",
       "price" : 250000
       },
       {
       "year" : 2013,
       "model" : "Porsche",
       "price" : 250000
       }
    ]
}
```

Some useful functions

- You can use the `limit` function to specify the maximum number of documents returned by your query.
- By setting this parameter to `0`, all the documents will be returned:

```
> db.users.find().limit(10)
```

- The `sort` function, on the other hand, can be used to sort the results returned from the query in ascending (1) or descending (-1) order.

- This function is pretty much equivalent to the ORDER BY statement in SQL.

```
> db.users.find({}).sort({"name":1})

{ "_id" : ObjectId("5506d5708d7bd8471669e674"), "name" : “Ahmed",
"age" : 44, "phone" : “092-456-789" }

{ "_id" : ObjectId("550ad3ef89ef057ee0671652"), "name" : “Ali",
"age" : 32, "phone" : “091-444-333" }
```

Some useful functions

- the `skip` function, which skips the first *n* documents in a collection. For example, here is how to skip the first document in a search across the `users` collection:

```
> db.users.find().skip(1)
{ "_id" : ObjectId("550ad3ef89ef057ee0671652"), "name" : “Ali",
"age" : 32, "phone" : “091-444-333" }
```

Securing database access

- Actually, starting mongod without any additional option exposes the database to any user who is aware of the process.
- We will show how to provide secure access by means of the mongo shell. So, launch the mongo shell and connect to the `admin` database, which holds information about the users:

```
use admin
```

- let's use the `createUser` function to add a user named `administrator` with the password `mypassword` and grant unlimited privileges (the role `root`):

```
db.createUser(
{
user: "administrator",
pwd: "mypassword",
roles: [ "root" ]
}
)
```

- Now, shut down the server by using the following command:

```
db.shutdownServer()
```

- We will restart the database using the `--auth` option, which forces user authentication:

```
mongod --dbpath "C:\mongodb-win32-x86_64-3.0.3\data" --auth
```

Securing database access

- Now, the database is started in secure mode. You can connect from the mongo shell in two different ways.
- The first one should be used with caution on Linux/Unix systems, as it exposes the user/password in the process list:

```
mongo -u administrator -p mypassword --authenticationDatabase
admin
```

- As an alternative, you can start the mongo shell and authenticate it at the beginning of the session (you need to select the admin database at first as the authentication keys are stored on the `admin` DB):

```
use admin
db.auth('admin','mypassword')
use yourdb
….
```

Reference Book

- **MongoDB for Java Developers_ Design, build, and deliver efficient Java applications using the most advanced NoSQL database.**
  Francesco Marchioni
  Copyright © 2015 Packt Publishing

- **MongoDB Data Modeling** **Wilson da Rocha França**
  Copyright © 2015 Packt Publishing

The End

Thanks for listening ..

Any questions ?