```
-----------------------------------------------------------------------------------------------
01-important-note
-----------------------------------------------------------------------------------------------
    // Definitions
    /**
     * # store data as documents like JavaScript Object
       # noSQL database
       # store like JSON
       # BSON (b-> binary) Data Structure into server
     */


     No Schema!
     User Collection
     id: 1 "name": "Raju" "age": 22  ...

    /**
    Relations
    No / Few Relations
    #Relational Data needs to be merger manually-- > kind of

    #flow
     1. Database
     2. Collections (like tables)
     3. Documents(like JSON Object values)-these are schemaless

    #MongoDB Ecosystem

    MongoDB Database
    1. Self - Managed / Enterprise     2. Atlas(Cloud)      3. Mobile
              |
              |
      CloudManager / OpsManager
    // other options
    1. Compass
    2. BI Connectors
    3. MongoDb Charts
     */

     Stich --> Basically a serverless backend solution
     This gives  :
      1. serverless query api -> a tool sets or tools to directly database query from inside client apps.
      2. Server Functions(in the cloud-> related to js)-> like  google cloud function or AWS Lamda.
      3. Database Triggers -> that allows us to events in a database, like document was inserted and then
         execute a function in  response to that and that function could then maybe send an e-mail.
      4. Real-Time Sync -> basically is built to synchronize a database in a cloud with that mobile offline supporting database.

    Working with MongoDb
    Backend Server -> Drivers(Node.js, Java, Python) or MongoDb Shell->(queries)<->(communicate) MongoDb Server->Storage Engine(File/Data Access)

                   | Read + Write Data to Files(slow) ->when Database
    Storage Engine |
                   | Read + Write Data to Files(fast) ->when Memory

    # documents created implicity


    JSON data converts into BSON data

    {
      "name" : "MAX",
      "age" : 29
    }

    it converts into BSON

    BSON 1. Binary data 2. Extends JSON Types(e.g more detailed Number Types) 3.
    Efficient Storage
```

```
---------------------------------------------------------------------------------------------------
02-database-operations-basic
---------------------------------------------------------------------------------------------------
```

```
  # show all database
  show dbs

  # create a new database
  > use shop
  switched to db shop

  > db.products.insertOne({name: "A Book", price: 12.99})

  > db.products.find()
  { "_id" : ObjectId("5f12a043feace8e519f293ed"), "name" : "A Book", "price" : 12.99 }

  > db.products.find().pretty()
  {
      "_id" : ObjectId("5f12a043feace8e519f293ed"),
      "name" : "A Book",
      "price" : 12.99
  }


  > db.products.insertOne({name: "A T-shirt", price: 29.99, description: "This is
  high quality T-shirt"})
  {
      "acknowledged" : true,
      "insertedId" : ObjectId("5f12e3b1fa9f127dd8b9c7b4")
  }
  > db.products.find().pretty()
  {
      "_id" : ObjectId("5f12a043feace8e519f293ed"),
      "name" : "A Book",
      "price" : 12.99
  }
  {
      "_id" : ObjectId("5f12e3b1fa9f127dd8b9c7b4"),
      "name" : "A T-shirt",
      "price" : 29.99,
      "description" : "This is high quality T-shirt"
  }

  > db.products.insertOne({name: "A Computer", price:34829.99, description: "This is
  high quality computer", details:{cpu: "Intel i7 8770",memory: 32}})
  {
      "acknowledged" : true,
      "insertedId" : ObjectId("5f12e5edfa9f127dd8b9c7b5")
  }
  > db.products.find().pretty()
  {
      "_id" : ObjectId("5f12a043feace8e519f293ed"),
      "name" : "A Book",
      "price" : 12.99
  }
  {
      "_id" : ObjectId("5f12e3b1fa9f127dd8b9c7b4"),
      "name" : "A T-shirt",
      "price" : 29.99,
      "description" : "This is high quality T-shirt"
  }
  {
      "_id" : ObjectId("5f12e5edfa9f127dd8b9c7b5"),
      "name" : "A Computer",
      "price" : 34829.99,
      "description" : "This is high quality computer",
      "details" : {
          "cpu" : "Intel i7 8770",
          "memory" : 32
      }
  }

  // to create a different port
  sudo mongod --port 27018
  // to start port
  mongo --port 27018


  > db.flightData.insertOne( {     "departureAirport": "MUC",     "arrivalAirport":
  "SFO",     "aircraft": "Airbus A380",     "distance": 12000,
  "intercontinental": true   })
  {
      "acknowledged" : true,
      "insertedId" : ObjectId("5f130951d022deabe244f26c")
  }
  > db.flightData.find().pretty()
  {
      "_id" : ObjectId("5f130951d022deabe244f26c"),
      "departureAirport" : "MUC",
      "arrivalAirport" : "SFO",
      "aircraft" : "Airbus A380",
      "distance" : 12000,
      "intercontinental" : true
  }


  -----------------CRUD Operations-----------------------

  // Create
  insertOne(data, options)
  insertMany(data, options)

  > db.flightData.insertMany([
  ...   {
  ...     "departureAirport": "MUC",
  ...     "arrivalAirport": "SFO",
  ...     "aircraft": "Airbus A380",
  ...     "distance": 12000,
  ...     "intercontinental": true
  ...   },
  ...   {
  ...     "departureAirport": "LHR",
  ...     "arrivalAirport": "TXL",
  ...     "aircraft": "Airbus A320",
  ...     "distance": 950,
  ...     "intercontinental": false
  ...   }
  ... ])
  {
      "acknowledged" : true,
      "insertedIds" : [
          ObjectId("5f132aebd022deabe244f26d"),
          ObjectId("5f132aebd022deabe244f26e")
      ]
  }
  > db.flightData.find().pretty()
```

```
{
    "_id" : ObjectId("5f132aebd022deabe244f26d"),
    "departureAirport" : "MUC",
    "arrivalAirport" : "SFO",
    "aircraft" : "Airbus A380",
    "distance" : 12000,
    "intercontinental" : true
}
{
    "_id" : ObjectId("5f132aebd022deabe244f26e"),
    "departureAirport" : "LHR",
    "arrivalAirport" : "TXL",
    "aircraft" : "Airbus A320",
    "distance" : 950,
    "intercontinental" : false
}
>


// Read
find(filter, options)
findOne(filter, options)

find gives us a cursor object not an array

> db.flightData.find({intercontinental: true}).pretty()
{
    "_id" : ObjectId("5f132aebd022deabe244f26d"),
    "departureAirport" : "MUC",
    "arrivalAirport" : "SFO",
    "aircraft" : "Airbus A380",
    "distance" : 12000,
    "intercontinental" : true
}

> db.flightData.find({distance: {$gt: 10000}}).pretty()
{
    "_id" : ObjectId("5f132aebd022deabe244f26d"),
    "departureAirport" : "MUC",
    "arrivalAirport" : "SFO",
    "aircraft" : "Airbus A380",
    "distance" : 12000,
    "intercontinental" : true
}

> db.flightData.findOne({distance: {$gt: 900}})
{
    "_id" : ObjectId("5f132aebd022deabe244f26d"),
    "departureAirport" : "MUC",
    "arrivalAirport" : "SFO",
    "aircraft" : "Airbus A380",
    "distance" : 12000,
    "intercontinental" : true
}

> db.passengers.find().toArray()
[
    {
        "_id" : ObjectId("5f1339f7d022deabe244f26f"),
        "name" : "Max Schwarzmueller",
        "age" : 29
    },
    {
        "_id" : ObjectId("5f1339f7d022deabe244f270"),
        "name" : "Manu Lorenz",
        "age" : 30
    },
    {
        "_id" : ObjectId("5f1339f7d022deabe244f271"),
            "name" : "Chris Hayton",
            "age" : 35
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f272"),
            "name" : "Sandeep Kumar",
            "age" : 28
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f273"),
            "name" : "Maria Jones",
            "age" : 30
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f274"),
            "name" : "Alexandra Maier",
            "age" : 27
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f275"),
            "name" : "Dr. Phil Evans",
            "age" : 47
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f276"),
            "name" : "Sandra Brugge",
            "age" : 33
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f277"),
            "name" : "Elisabeth Mayr",
            "age" : 29
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f278"),
            "name" : "Frank Cube",
            "age" : 41
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f279"),
            "name" : "Karandeep Alun",
            "age" : 48
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f27a"),
            "name" : "Michaela Drayer",
            "age" : 39
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f27b"),
            "name" : "Bernd Hoftstadt",
            "age" : 22
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f27c"),
            "name" : "Scott Tolib",
            "age" : 44
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f27d"),
            "name" : "Freddy Melver",
            "age" : 41
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f27e"),
            "name" : "Alexis Bohed",
            "age" : 35
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f27f"),
            "name" : "Melanie Palace",
            "age" : 27
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f280"),
            "name" : "Armin Glutch",
            "age" : 35
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f281"),
            "name" : "Klaus Arber",
            "age" : 53
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f282"),
            "name" : "Albert Twostone",
            "age" : 68
        },
        {
            "_id" : ObjectId("5f1339f7d022deabe244f283"),
            "name" : "Gordon Black",
            "age" : 38
        }
    ]


    > db.passengers.find().forEach((passengerData) => {printjson(passengerData)})
    {
        "_id" : ObjectId("5f1339f7d022deabe244f26f"),
        "name" : "Max Schwarzmueller",
        "age" : 29
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f270"),
        "name" : "Manu Lorenz",
        "age" : 30
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f271"),
        "name" : "Chris Hayton",
        "age" : 35
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f272"),
        "name" : "Sandeep Kumar",
        "age" : 28
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f273"),
        "name" : "Maria Jones",
        "age" : 30
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f274"),
        "name" : "Alexandra Maier",
        "age" : 27
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f275"),
        "name" : "Dr. Phil Evans",
        "age" : 47
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f276"),
        "name" : "Sandra Brugge",
        "age" : 33
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f277"),
        "name" : "Elisabeth Mayr",
        "age" : 29
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f278"),
        "name" : "Frank Cube",
        "age" : 41
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f279"),
        "name" : "Karandeep Alun",
        "age" : 48
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f27a"),
        "name" : "Michaela Drayer",
        "age" : 39
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f27b"),
        "name" : "Bernd Hoftstadt",
        "age" : 22
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f27c"),
        "name" : "Scott Tolib",
        "age" : 44
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f27d"),
        "name" : "Freddy Melver",
        "age" : 41
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f27e"),
        "name" : "Alexis Bohed",
        "age" : 35
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f27f"),
        "name" : "Melanie Palace",
        "age" : 27
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f280"),
        "name" : "Armin Glutch",
        "age" : 35
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f281"),
        "name" : "Klaus Arber",
        "age" : 53
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f282"),
        "name" : "Albert Twostone",
        "age" : 68
    }
    {
        "_id" : ObjectId("5f1339f7d022deabe244f283"),
        "name" : "Gordon Black",
        "age" : 38
    }


    // Update
    updateOne(filter, data, options)
    updateMany(filter, data, options)
```

```
replaceOne(filter, data, options)

> db.flightData.updateOne({distance: 1200},{$set:{marker: "delete"}})
> db.flightData.updateMany({},{$set:{marker: "toDelete"}})

> db.flightData.updateOne({_id :
ObjectId("5f132aebd022deabe244f26d")},{$set:{delayed: true}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
> db.flightData.find().pretty()
{
    "_id" : ObjectId("5f132aebd022deabe244f26d"),
    "departureAirport" : "MUC",
    "arrivalAirport" : "SFO",
    "aircraft" : "Airbus A380",
    "distance" : 12000,
    "intercontinental" : true,
    "delayed" : true
}
{
    "_id" : ObjectId("5f132aebd022deabe244f26e"),
    "departureAirport" : "LHR",
    "arrivalAirport" : "TXL",
    "aircraft" : "Airbus A320",
    "distance" : 950,
    "intercontinental" : false
}

> db.flightData.update({_id : ObjectId("5f132aebd022deabe244f26d")},{$set:{delayed:
false}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.flightData.find().pretty()
{
    "_id" : ObjectId("5f132aebd022deabe244f26d"),
    "departureAirport" : "MUC",
    "arrivalAirport" : "SFO",
    "aircraft" : "Airbus A380",
    "distance" : 12000,
    "intercontinental" : true,
    "delayed" : false
}
{
    "_id" : ObjectId("5f132aebd022deabe244f26e"),
    "departureAirport" : "LHR",
    "arrivalAirport" : "TXL",
    "aircraft" : "Airbus A320",
    "distance" : 950,
    "intercontinental" : false
}

> db.flightData.update({_id : ObjectId("5f132aebd022deabe244f26d")},{delayed: false})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.flightData.find().pretty()
{ "_id" : ObjectId("5f132aebd022deabe244f26d"), "delayed" : false }
{
    "_id" : ObjectId("5f132aebd022deabe244f26e"),
    "departureAirport" : "LHR",
    "arrivalAirport" : "TXL",
    "aircraft" : "Airbus A320",
    "distance" : 950,
    "intercontinental" : false
}
// same can be done by replaceOne
> db.flightData.replaceOne({_id : ObjectId("5f132aebd022deabe244f26d")},{
...     "departureAirport": "MUC",
...     "arrivalAirport": "SFO",
...     "aircraft": "Airbus A380",
...     "distance": 12000,
...     "intercontinental": true
...   })
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.flightData.find().pretty()
{
    "_id" : ObjectId("5f132aebd022deabe244f26d"),
    "departureAirport" : "MUC",
    "arrivalAirport" : "SFO",
    "aircraft" : "Airbus A380",
    "distance" : 12000,
    "intercontinental" : true
}
{
    "_id" : ObjectId("5f132aebd022deabe244f26e"),
    "departureAirport" : "LHR",
    "arrivalAirport" : "TXL",
    "aircraft" : "Airbus A320",
    "distance" : 950,
    "intercontinental" : false
}


Delete
deleteOne(filter, options)
deleteMany(filter, options)

db.flightData.deleteOne({departureAirport :"MUC"})
> db.flightData.deleteMany({marker: "toDelete"})


------------------- Projection-------------------

> db.passengers.find({},{name: 1}).pretty()
{
    "_id" : ObjectId("5f1339f7d022deabe244f26f"),
    "name" : "Max Schwarzmueller"
}
{ "_id" : ObjectId("5f1339f7d022deabe244f270"), "name" : "Manu Lorenz" }
{ "_id" : ObjectId("5f1339f7d022deabe244f271"), "name" : "Chris Hayton" }
{ "_id" : ObjectId("5f1339f7d022deabe244f272"), "name" : "Sandeep Kumar" }
{ "_id" : ObjectId("5f1339f7d022deabe244f273"), "name" : "Maria Jones" }
{ "_id" : ObjectId("5f1339f7d022deabe244f274"), "name" : "Alexandra Maier" }
{ "_id" : ObjectId("5f1339f7d022deabe244f275"), "name" : "Dr. Phil Evans" }
{ "_id" : ObjectId("5f1339f7d022deabe244f276"), "name" : "Sandra Brugge" }
{ "_id" : ObjectId("5f1339f7d022deabe244f277"), "name" : "Elisabeth Mayr" }
{ "_id" : ObjectId("5f1339f7d022deabe244f278"), "name" : "Frank Cube" }
{ "_id" : ObjectId("5f1339f7d022deabe244f279"), "name" : "Karandeep Alun" }
{ "_id" : ObjectId("5f1339f7d022deabe244f27a"), "name" : "Michaela Drayer" }
{ "_id" : ObjectId("5f1339f7d022deabe244f27b"), "name" : "Bernd Hoftstadt" }
{ "_id" : ObjectId("5f1339f7d022deabe244f27c"), "name" : "Scott Tolib" }
{ "_id" : ObjectId("5f1339f7d022deabe244f27d"), "name" : "Freddy Melver" }
{ "_id" : ObjectId("5f1339f7d022deabe244f27e"), "name" : "Alexis Bohed" }
{ "_id" : ObjectId("5f1339f7d022deabe244f27f"), "name" : "Melanie Palace" }
{ "_id" : ObjectId("5f1339f7d022deabe244f280"), "name" : "Armin Glutch" }
{ "_id" : ObjectId("5f1339f7d022deabe244f281"), "name" : "Klaus Arber" }
{ "_id" : ObjectId("5f1339f7d022deabe244f282"), "name" : "Albert Twostone" }


> db.passengers.find({},{name: 1,_id: 0}).pretty()
{ "name" : "Max Schwarzmueller" }
{ "name" : "Manu Lorenz" }
{ "name" : "Chris Hayton" }
{ "name" : "Sandeep Kumar" }
{ "name" : "Maria Jones" }
{ "name" : "Alexandra Maier" }
{ "name" : "Dr. Phil Evans" }
{ "name" : "Sandra Brugge" }
{ "name" : "Elisabeth Mayr" }
{ "name" : "Frank Cube" }
{ "name" : "Karandeep Alun" }
{ "name" : "Michaela Drayer" }
{ "name" : "Bernd Hoftstadt" }
{ "name" : "Scott Tolib" }
{ "name" : "Freddy Melver" }
{ "name" : "Alexis Bohed" }
{ "name" : "Melanie Palace" }
{ "name" : "Armin Glutch" }
{ "name" : "Klaus Arber" }
{ "name" : "Albert Twostone" }
```

-------------------------Embedded Documents---------------------

nested documents
have size limit limitations (50mb) up to 100 times

```
> db.flightData.updateMany({},{$set:{status:{description: "on-time", lastupdated: "i Hour ago"}}})
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 0 }
> db.flightData.find().pretty()
{
    "_id" : ObjectId("5f132aebd022deabe244f26d"),
    "departureAirport" : "MUC",
    "arrivalAirport" : "SFO",
    "aircraft" : "Airbus A380",
    "distance" : 12000,
    "intercontinental" : true,
    "status" : {
        "description" : "on-time",
        "lastupdated" : "i Hour ago"
    }
}
{
    "_id" : ObjectId("5f132aebd022deabe244f26e"),
    "departureAirport" : "LHR",
    "arrivalAirport" : "TXL",
    "aircraft" : "Airbus A320",
    "distance" : 950,
    "intercontinental" : false,
    "status" : {
        "description" : "on-time",
        "lastupdated" : "i Hour ago"
    }
}

> db.flightData.updateMany({},{$set:{status:{description: "on-time", lastupdated: "i Hour ago",details:{responsible: "RAJU"}}}})
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
> db.flightData.find().pretty()
{
    "_id" : ObjectId("5f132aebd022deabe244f26d"),
    "departureAirport" : "MUC",
    "arrivalAirport" : "SFO",
    "aircraft" : "Airbus A380",
    "distance" : 12000,
    "intercontinental" : true,
    "status" : {
        "description" : "on-time",
        "lastupdated" : "i Hour ago",
        "details" : {
            "responsible" : "RAJU"
        }
    }
}
{
    "_id" : ObjectId("5f132aebd022deabe244f26e"),
    "departureAirport" : "LHR",
    "arrivalAirport" : "TXL",
    "aircraft" : "Airbus A320",
    "distance" : 950,
    "intercontinental" : false,
    "status" : {
        "description" : "on-time",
        "lastupdated" : "i Hour ago",
        "details" : {
            "responsible" : "RAJU"
        }
    }
}

// query an documents
> db.flightData.find({"status.description": "on-time"}).pretty()
{
    "_id" : ObjectId("5f132aebd022deabe244f26d"),
    "departureAirport" : "MUC",
    "arrivalAirport" : "SFO",
    "aircraft" : "Airbus A380",
    "distance" : 12000,
    "intercontinental" : true,
    "status" : {
        "description" : "on-time",
        "lastupdated" : "i Hour ago",
        "details" : {
            "responsible" : "RAJU"
        }
    }
}
{
    "_id" : ObjectId("5f132aebd022deabe244f26e"),
    "departureAirport" : "LHR",
    "arrivalAirport" : "TXL",
    "aircraft" : "Airbus A320",
    "distance" : 950,
    "intercontinental" : false,
    "status" : {
        "description" : "on-time",
        "lastupdated" : "i Hour ago",
        "details" : {
            "responsible" : "RAJU"
        }
    }
}


------------------------Arrays-----------------------

Array of imbedded documents
Array can hold any data

> db.passengers.updateOne({name: "Albert Twostone"}, {$set:{hobbies:["sports",
"cooking"]}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 0 }
> db.passengers.find().pretty()
{
    "_id" : ObjectId("5f1339f7d022deabe244f26f"),
    "name" : "Max Schwarzmueller",
    "age" : 29
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f270"),
    "name" : "Manu Lorenz",
    "age" : 30
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f271"),
    "name" : "Chris Hayton",
    "age" : 35
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f272"),
    "name" : "Sandeep Kumar",
    "age" : 28
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f273"),
    "name" : "Maria Jones",
    "age" : 30
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f274"),
    "name" : "Alexandra Maier",
    "age" : 27
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f275"),
    "name" : "Dr. Phil Evans",
    "age" : 47
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f276"),
    "name" : "Sandra Brugge",
    "age" : 33
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f277"),
    "name" : "Elisabeth Mayr",
    "age" : 29
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f278"),
    "name" : "Frank Cube",
    "age" : 41
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f279"),
    "name" : "Karandeep Alun",
    "age" : 48
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f27a"),
    "name" : "Michaela Drayer",
    "age" : 39
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f27b"),
    "name" : "Bernd Hoftstadt",
    "age" : 22
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f27c"),
    "name" : "Scott Tolib",
    "age" : 44
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f27d"),
    "name" : "Freddy Melver",
    "age" : 41
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f27e"),
    "name" : "Alexis Bohed",
    "age" : 35
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f27f"),
    "name" : "Melanie Palace",
    "age" : 27
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f280"),
    "name" : "Armin Glutch",
    "age" : 35
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f281"),
    "name" : "Klaus Arber",
    "age" : 53
}
{
    "_id" : ObjectId("5f1339f7d022deabe244f282"),
    "name" : "Albert Twostone",
    "age" : 68,
    "hobbies" : [
        "sports",
        "cooking"
    ]
}

> db.passengers.find({name: "Albert Twostone"}).pretty()
{
    "_id" : ObjectId("5f1339f7d022deabe244f282"),
    "name" : "Albert Twostone",
    "age" : 68,
    "hobbies" : [
        "sports",
        "cooking"
    ]
}
> db.passengers.find({name: "Albert Twostone"}).hobbies
> db.passengers.findOne({name: "Albert Twostone"}).hobbies
[ "sports", "cooking" ]

// query an array
> db.passengers.find({hobbies: "sports"}).pretty()
{
    "_id" : ObjectId("5f1339f7d022deabe244f282"),
    "name" : "Albert Twostone",
    "age" : 68,
    "hobbies" : [
        "sports",
        "cooking"
    ]
}




-----------------------------------------------------------------------------------------------------
03-schemas-relations-how-to-structure-documents
-----------------------------------------------------------------------------------------------------

> db.companies.insertOne({name: "Freash Apples Inc", isStartup: true, employees: 33, funding: 12345678987654321 9, details: {cea: "Mark Super"},tags: [{title: "super"},{title: "perfect"}], foundingData: new Date(), insertedAt: new Timestamp()})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f13ec400249b11a6aa5e37f")
}
> db.companies.find()
{ "_id" : ObjectId("5f13ec400249b11a6aa5e37f"), "name" : "Freash Apples Inc", "isStartup" : true, "employees" : 33, "funding" : 12345678987654322 0, "details" : { "cea" : "Mark Super" }, "tags" : [ { "title" : "super" }, { "title" : "perfect" } ], "foundingData" : ISODate("2020-07-19T06:46:24.175Z"), "insertedAt" : Timestamp(1595141184, 1) }
> db.companies.find().pretty()
{
    "_id" : ObjectId("5f13ec400249b11a6aa5e37f"),
    "name" : "Freash Apples Inc",
    "isStartup" : true,
    "employees" : 33,
    "funding" : 123456789876543220,
    "details" : {
        "cea" : "Mark Super"
    },
    "tags" : [
        {
            "title" : "super"
        },
        {
            "title" : "perfect"
        }
    ],
    "foundingData" : ISODate("2020-07-19T06:46:24.175Z"),
    "insertedAt" : Timestamp(1595141184, 1)
}

--------------------get the statistics of database-------------------------
> db.numbers.insertOne({a: 1})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f13ed5a0249b11a6aa5e380")
}
> db.numbers.findOne()
{ "_id" : ObjectId("5f13ed5a0249b11a6aa5e380"), "a" : 1 }
> db.stats
function (scale) {
        return this.runCommand({dbstats: 1, scale: scale});
    }
> db.stats()
{
    "db" : "companyData",
    "collections" : 2,
    "views" : 0,
    "objects" : 2,
    "avgObjSize" : 135,
    "dataSize" : 270,
    "storageSize" : 20480,
    "numExtents" : 0,
    "indexes" : 2,
    "indexSize" : 20480,
    "fsUsedSize" : 54183743488,
    "fsTotalSize" : 61754699776,
    "ok" : 1
}

// differentiate data size
> db.numbers.insertOne({a: 1})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f13eee00249b11a6aa5e381")
}
> db.stats()
{
    "db" : "companyData",
    "collections" : 1,
    "views" : 0,
    "objects" : 1,
    "avgObjSize" : 33,
    "dataSize" : 33,
    "storageSize" : 4096,
    "numExtents" : 0,
    "indexes" : 1,
    "indexSize" : 4096,
    "fsUsedSize" : 54183714816,
    "fsTotalSize" : 61754699776,
    "ok" : 1
}

> db.numbers.drop()
true
> db.numbers.insertOne({a: NumberInt(1)})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f13ef440249b11a6aa5e382")
}
> db.stats()
{
    "db" : "companyData",
    "collections" : 1,
    "views" : 0,
    "objects" : 1,
    "avgObjSize" : 29,
    "dataSize" : 29,
    "storageSize" : 4096,
    "numExtents" : 0,
    "indexes" : 1,
    "indexSize" : 4096,
    "fsUsedSize" : 54183739392,
    "fsTotalSize" : 61754699776,
    "ok" : 1
}

---------------getting the datatype----------------

> db.numbers.insertOne({a: 1.5,b: "r"})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f13f19e0249b11a6aa5e386")
}
> typeof db.numbers.findOne({b: "r"}).a
number

---------------------------Relations---------------------
one-to-one
one-to-many
many-to-many

--------------------------Joining with $lookup--------------------

> use bookStore
switched to db bookStore
> cls

> db.authors.insertMany([{name: 'Max Scwarz',age: 29, address:{street:
'Main'}},{name: 'Manuel Lor',age: 30, address:{street: 'Tree'}}])
{
    "acknowledged" : true,
    "insertedIds" : [
        ObjectId("5f145a7c231893e15e9e53fe"),
        ObjectId("5f145a7c231893e15e9e53ff")
    ]
}
> db.authors.find().pretty()
{
    "_id" : ObjectId("5f145a7c231893e15e9e53fe"),
    "name" : "Max Scwarz",
    "age" : 29,
    "address" : {
        "street" : "Main"
    }
}
{
    "_id" : ObjectId("5f145a7c231893e15e9e53ff"),
    "name" : "Manuel Lor",
    "age" : 30,
    "address" : {
        "street" : "Tree"
    }
}
> db.books.insertOne({name: 'My favorite
Book',authors:[ObjectId("5f145a7c231893e15e9e53fe"),ObjectId("5f145a7c231893e15e9e53f
f")]})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f145b5f231893e15e9e5400")
}
> db.authors.findOne()
{
    "_id" : ObjectId("5f145a7c231893e15e9e53fe"),
    "name" : "Max Scwarz",
    "age" : 29,
    "address" : {
        "street" : "Main"
    }
}
> db.books.findOne()
{
    "_id" : ObjectId("5f145b5f231893e15e9e5400"),
    "name" : "My favorite Book",
    "authors" : [
        ObjectId("5f145a7c231893e15e9e53fe"),
        ObjectId("5f145a7c231893e15e9e53ff")
    ]
}


> db.books.aggregate([{$lookup:{from:
'authors',localField:'authors',foreignField:"_id", as:'creators'}}])
{ "_id" : ObjectId("5f145b5f231893e15e9e5400"), "name" : "My favorite Book",
"authors" : [ ObjectId("5f145a7c231893e15e9e53fe"),
ObjectId("5f145a7c231893e15e9e53ff") ], "creators" : [ { "_id" :
ObjectId("5f145a7c231893e15e9e53fe"), "name" : "Max Scwarz", "age" : 29, "address"
: { "street" : "Main" } }, { "_id" : ObjectId("5f145a7c231893e15e9e53ff"), "name" :
"Manuel Lor", "age" : 30, "address" : { "street" : "Tree" } } ] }
> db.books.aggregate([{$lookup:{from:
'authors',localField:'authors',foreignField:"_id", as:'creators'}}]).pretty()
{
    "_id" : ObjectId("5f145b5f231893e15e9e5400"),
    "name" : "My favorite Book",
    "authors" : [
        ObjectId("5f145a7c231893e15e9e53fe"),
        ObjectId("5f145a7c231893e15e9e53ff")
    ],
    "creators" : [
        {
            "_id" : ObjectId("5f145a7c231893e15e9e53fe"),
            "name" : "Max Scwarz",
            "age" : 29,
            "address" : {
                "street" : "Main"
            }
        },
        {
            "_id" : ObjectId("5f145a7c231893e15e9e53ff"),
            "name" : "Manuel Lor",
            "age" : 30,
            "address" : {
                "street" : "Tree"
            }
        }
    ]
}
```

```
    > db.books.find().pretty()
    {
        "_id" : ObjectId("5f145b5f231893e15e9e5400"),
        "name" : "My favorite Book",
        "authors" : [
            ObjectId("5f145a7c231893e15e9e53fe"),
            ObjectId("5f145a7c231893e15e9e53ff")
        ]
    }




-------------------------------------------------------------------------------------------------
04-simple-project-blog
-------------------------------------------------------------------------------------------------

    -------------------Blog Project-------------------

    > use blog
    switched to db blog
    > db.users.insertMany([{name: 'Max Schwaezmuller', age: 29,email:'max23@gmail.com'},{name: 'Raju', age:'22', email: 'mijanur231@gmail.com'}])
    {
        "acknowledged" : true,
        "insertedIds" : [
            ObjectId("5f146535231893e15e9e5401"),
            ObjectId("5f146535231893e15e9e5402")
        ]
    }
    > db.users.find().pretty()
    {
        "_id" : ObjectId("5f146535231893e15e9e5401"),
        "name" : "Max Schwaezmuller",
        "age" : 29,
        "email" : "max23@gmail.com"
    }
    {
        "_id" : ObjectId("5f146535231893e15e9e5402"),
        "name" : "Raju",
        "age" : "22",
        "email" : "mijanur231@gmail.com"
    }
    > db.post.insertOne({title: 'My first post is', text:'This is very important post, I hope you like it!',tags:['new', 'tech'], creator: ObjectId("5f146535231893e15e9e5402"), comments: [{text: 'I like this post', author: ObjectId("5f146535231893e15e9e5401")}]})
    {
        "acknowledged" : true,
        "insertedId" : ObjectId("5f146bba231893e15e9e5403")
    }
    > db.posts.findOne()
    null
    > db.post.findOne()
    {
        "_id" : ObjectId("5f146bba231893e15e9e5403"),
        "title" : "My first post is",
        "text" : "This is very important post, I hope you like it!",
        "tags" : [
            "new",
            "tech"
        ],
        "creator" : ObjectId("5f146535231893e15e9e5402"),
        "comments" : [
            {
                "text" : "I like this post",
                "author" : ObjectId("5f146535231893e15e9e5401")
            }
        ]
    }

    --------------------validation check----------------------

    > db.post.drop()
    true
    > db.post.findOne()
    null
    > db.createCollection('post', {
    ...   validator: {
    ...     $jsonSchema: {
    ...       bsonType: 'object',
    ...       required: ['title', 'text', 'creator', 'comments'],
    ...       properties: {
    ...         title: {
    ...           bsonType: 'string',
    ...           description: 'must be a string and is required'
    ...         },
    ...         text: {
    ...           bsonType: 'string',
    ...           description: 'must be a string and is required'
    ...         },
    ...         creator: {
    ...           bsonType: 'objectId',
    ...           description: 'must be an objectid and is required'
    ...         },
    ...         comments: {
    ...           bsonType: 'array',
    ...           description: 'must be an array and is required',
    ...           items: {
    ...             bsonType: 'object',
    ...             required: ['text', 'author'],
    ...             properties: {
    ...               text: {
    ...                 bsonType: 'string',
    ...                 description: 'must be a string and is required'
    ...               },
    ...               author: {
    ...                 bsonType: 'objectId',
    ...                 description: 'must be an objectid and is required'
    ...               }
    ...             }
    ...           }
    ...         }
    ...       }
    ...     }
    ...   }
    ... });
    { "ok" : 1 }
    > db.post.insertOne({title: 'My first post is', text:'This is very important post,
    I hope you like it!',tags:['new', 'tech'], creator:
    ObjectId("5f146535231893e15e9e5402"), comments: [{text: 'I like this post', author:
    ObjectId("5f146535231893e15e9e5401")}]})
    {
        "acknowledged" : true,
        "insertedId" : ObjectId("5f148c00231893e15e9e5404")
    }
    > db.post.findOne()
    {
        "_id" : ObjectId("5f148c00231893e15e9e5404"),
        "title" : "My first post is",
        "text" : "This is very important post, I hope you like it!",
        "tags" : [
            "new",
            "tech"
        ],
        "creator" : ObjectId("5f146535231893e15e9e5402"),
        "comments" : [
            {
                "text" : "I like this post",
                "author" : ObjectId("5f146535231893e15e9e5401")
            }
        ]
    }


    > db.post.insertOne({title: 'My first post is', text:'This is very important post, I hope you like it!',tags:['new', 'tech'], creator: ObjectId("5f146535231893e15e9e5402"), comments: [{text: 'I like this post', author:1234}]})
    2020-07-20T00:12:38.402+0600 E QUERY    [thread1] WriteError: Document failed validation :
    WriteError({
        "index" : 0,
        "code" : 121,
        "errmsg" : "Document failed validation",
        "op" : {
            "_id" : ObjectId("5f148d16231893e15e9e5406"),
            "title" : "My first post is",
            "text" : "This is very important post, I hope you like it!",
            "tags" : [
                "new",
                "tech"
            ],
            "creator" : ObjectId("5f146535231893e15e9e5402"),
            "comments" : [
                {
                    "text" : "I like this post",
                    "author" : 1234
                }
            ]
        }
    })
    WriteError@src/mongo/shell/bulk_api.js:466:48
    Bulk/mergeBatchResults@src/mongo/shell/bulk_api.js:846:49
    Bulk/executeBatch@src/mongo/shell/bulk_api.js:910:13
    Bulk/this.execute@src/mongo/shell/bulk_api.js:1154:21
    DBCollection.prototype.insertOne@src/mongo/shell/crud_api.js:252:9
    @(shell):1:1

    > db.post.find().pretty()
    {
        "_id" : ObjectId("5f148d87231893e15e9e5407"),
        "title" : "My first post is",
        "text" : "This is very important post, I hope you like it!",
        "tags" : [
            "new",
            "tech"
        ],
        "creator" : ObjectId("5f146535231893e15e9e5402"),
        "comments" : [
            {
                "text" : "I like this post",
                "author" : ObjectId("5f146535231893e15e9e5401")
            }
        ]
    }

    ------------------------administrative check-------------------------
    // collMod -> collections mode

    > db.runCommand({collMod: 'post', validator: {
    ...     $jsonSchema: {
    ...       bsonType: 'object',
    ...       required: ['title', 'text', 'creator', 'comments'],
    ...       properties: {
```

```
...          title: {
...            bsonType: 'string',
...            description: 'must be a string and is required'
...          },
...          text: {
...            bsonType: 'string',
...            description: 'must be a string and is required'
...          },
...          creator: {
...            bsonType: 'objectId',
...            description: 'must be an objectid and is required'
...          },
...          comments: {
...            bsonType: 'array',
...            description: 'must be an array and is required',
...            items: {
...              bsonType: 'object',
...              required: ['text', 'author'],
...              properties: {
...                text: {
...                  bsonType: 'string',
...                  description: 'must be a string and is required'
...                },
...                author: {
...                  bsonType: 'objectId',
...                  description: 'must be an objectid and is required'
...                }
...              }
...            }
...          }
...        }
...      }
...   }})
{ "ok" : 1 }


> db.runCommand({
  collMod: 'posts',
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      required: ['title', 'text', 'creator', 'comments'],
      properties: {
        title: {
          bsonType: 'string',
          description: 'must be a string and is required'
        },
        text: {
          bsonType: 'string',
          description: 'must be a string and is required'
        },
        creator: {
          bsonType: 'objectId',
          description: 'must be an objectid and is required'
        },
        comments: {
          bsonType: 'array',
          description: 'must be an array and is required',
          items: {
            bsonType: 'object',
            required: ['text', 'author'],
            properties: {
              text: {
                bsonType: 'string',
                description: 'must be a string and is required'
              },
              author: {
                bsonType: 'objectId',
                description: 'must be an objectid and is required'
```

```
                    }
                  }
                }
              }
            }
          }
        },
        validationAction: 'warn'
      });
      { "ok" : 1 }
      > db.post.insertOne({title: 'My first post is', text:'This is very important post,
      I hope you like it!',tags:['new', 'tech'], creator:
      ObjectId("5f146535231893e15e9e5402"), comments: [{text: 'I like this post',
      author:1234}]})
      {
          "acknowledged" : true,
          "insertedId" : ObjectId("5f14930f231893e15e9e5409")
```

```
----------------------------------------------------------------------------------------------------------------
05-shell-important-commands
----------------------------------------------------------------------------------------------------------------
```

```
      mongod --help

      > help
          db.help()                    help on db methods
          db.mycoll.help()             help on collection methods
          sh.help()                    sharding helpers
          rs.help()                    replica set helpers
          help admin                   administrative help
          help connect                 connecting to a db help
          help keys                    key shortcuts
          help misc                    misc things to know
          help mr                      mapreduce

          show dbs                     show database names
          show collections             show collections in current database
          show users                   show users in current database
          show profile                 show most recent system.profile entries with time
          >= 1ms
          show logs                    show the accessible logger names
          show log [name]              prints out the last segment of log in memory,
          'global' is default
          use <db_name>                set current database
          db.foo.find()                list objects in collection foo
          db.foo.find( { a : 1 } )     list objects in foo where a == 1
          it                           result of the last line evaluated; use to further
          iterate
          DBQuery.shellBatchSize = x   set default number of items to display on shell
          exit                         quit the mongo shell



      > use shop
      switched to db shop
      > db.help()
      DB methods:
          db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs command
          [just calls db.runCommand(...)]
          db.aggregate([pipeline], {options}) - performs a collectionless aggregation on
          this database; returns a cursor
          db.auth(username, password)
```

```
        db.cloneDatabase(fromhost)
        db.commandHelp(name) returns the help for the command
        db.copyDatabase(fromdb, todb, fromhost)
        db.createCollection(name, {size: ..., capped: ..., max: ...})
        db.createView(name, viewOn, [{$operator: {...}}, ...], {viewOptions})
        db.createUser(userDocument)
        db.currentOp() displays currently executing operations in the db
        db.dropDatabase()
        db.eval() - deprecated
        db.fsyncLock() flush data to disk and lock server for backups
        db.fsyncUnlock() unlocks server following a db.fsyncLock()
        db.getCollection(cname) same as db['cname'] or db.cname
        db.getCollectionInfos([filter]) - returns a list that contains the names and
        options of the db's collections
        db.getCollectionNames()
        db.getLastError() - just returns the err msg string
        db.getLastErrorObj() - return full status object
        db.getLogComponents()
        db.getMongo() get the server connection object
        db.getMongo().setSlaveOk() allow queries on a replication slave server
        db.getName()
        db.getPrevError()
        db.getProfilingLevel() - deprecated
        db.getProfilingStatus() - returns if profiling is on and slow threshold
        db.getReplicationInfo()
        db.getSiblingDB(name) get the db at the same server as this one
        db.getWriteConcern() - returns the write concern used for any operations on
        this db, inherited from server object if set
        db.hostInfo() get details about the server's host
        db.isMaster() check replica primary status
        db.killOp(opid) kills the current operation in the db
        db.listCommands() lists all the db commands
        db.loadServerScripts() loads all the scripts in db.system.js
        db.logout()
        db.printCollectionStats()
        db.printReplicationInfo()
        db.printShardingStatus()
        db.printSlaveReplicationInfo()
        db.dropUser(username)
        db.repairDatabase()
        db.resetError()
        db.runCommand(cmdObj) run a database command.  if cmdObj is a string, turns it
        into {cmdObj: 1}
        db.serverStatus()
        db.setLogLevel(level,<component>)
        db.setProfilingLevel(level,slowms) 0=off 1=slow 2=all
        db.setWriteConcern(<write concern doc>) - sets the write concern for writes to
        the db
        db.unsetWriteConcern(<write concern doc>) - unsets the write concern for writes
        to the db
        db.setVerboseShell(flag) display extra information in shell output
        db.shutdownServer()
        db.stats()
        db.version() current version of the server




    > show collections
    products
    > db.products.help()
    DBCollection help
        db.products.find().help() - show DBCursor help
        db.products.bulkWrite( operations, <optional params> ) - bulk execute write
        operations, optional parameters are: w, wtimeout, j
        db.products.count( query = {}, <optional params> ) - count the number of
        documents that matches the query, optional parameters are: limit, skip, hint,
        maxTimeMS
        db.products.copyTo(newColl) - duplicates collection by copying all documents to
```

```
newColl; no indexes are copied.
	db.products.convertToCapped(maxBytes) - calls {convertToCapped:'products', size:maxBytes}} command
	db.products.createIndex(keypattern[,options])
	db.products.createIndexes([keypatterns], <options>)
	db.products.dataSize()
	db.products.deleteOne( filter, <optional params> ) - delete first matching document, optional parameters are: w, wtimeout, j
	db.products.deleteMany( filter, <optional params> ) - delete all matching documents, optional parameters are: w, wtimeout, j
	db.products.distinct( key, query, <optional params> ) - e.g. db.products.distinct( 'x' ), optional parameters are: maxTimeMS
	db.products.drop() drop the collection
	db.products.dropIndex(index) - e.g. db.products.dropIndex( "indexName" ) or db.products.dropIndex( { "indexKey" : 1 } )
	db.products.dropIndexes()
	db.products.ensureIndex(keypattern[,options]) - DEPRECATED, use createIndex() instead
	db.products.explain().help() - show explain help
	db.products.reIndex()
	db.products.find([query],[fields]) - query is an optional query filter. fields is optional set of fields to return.
	                                              e.g. db.products.find( {x:77} , {name:1, x:1} )
	db.products.find(...).count()
	db.products.find(...).limit(n)
	db.products.find(...).skip(n)
	db.products.find(...).sort(...)
	db.products.findOne([query], [fields], [options], [readConcern])
	db.products.findOneAndDelete( filter, <optional params> ) - delete first matching document, optional parameters are: projection, sort, maxTimeMS
	db.products.findOneAndReplace( filter, replacement, <optional params> ) - replace first matching document, optional parameters are: projection, sort, maxTimeMS, upsert, returnNewDocument
	db.products.findOneAndUpdate( filter, update, <optional params> ) - update first matching document, optional parameters are: projection, sort, maxTimeMS, upsert, returnNewDocument
	db.products.getDB() get DB object associated with collection
	db.products.getPlanCache() get query plan cache associated with collection
	db.products.getIndexes()
	db.products.group( { key : ..., initial: ..., reduce : ...[, cond: ...] } )
	db.products.insert(obj)
	db.products.insertOne( obj, <optional params> ) - insert a document, optional parameters are: w, wtimeout, j
	db.products.insertMany( [objects], <optional params> ) - insert multiple documents, optional parameters are: w, wtimeout, j
	db.products.mapReduce( mapFunction , reduceFunction , <optional params> )
	db.products.aggregate( [pipeline], <optional params> ) - performs an aggregation on a collection; returns a cursor
	db.products.remove(query)
	db.products.replaceOne( filter, replacement, <optional params> ) - replace the first matching document, optional parameters are: upsert, w, wtimeout, j
	db.products.renameCollection( newName , <dropTarget> ) renames the collection.
	db.products.runCommand( name , <options> ) runs a db command with the given name where the first param is the collection name
	db.products.save(obj)
	db.products.stats({scale: N, indexDetails: true/false, indexDetailsKey: <index key>, indexDetailsName: <index name>})
	db.products.storageSize() - includes free space allocated to this collection
	db.products.totalIndexSize() - size in bytes of all the indexes
	db.products.totalSize() - storage allocated for all data and indexes
	db.products.update( query, object[, upsert_bool, multi_bool] ) - instead of two flags, you can pass an object with fields: upsert, multi
	db.products.updateOne( filter, update, <optional params> ) - update the first matching document, optional parameters are: upsert, w, wtimeout, j
	db.products.updateMany( filter, update, <optional params> ) - update all matching documents, optional parameters are: upsert, w, wtimeout, j
	db.products.validate( <full> ) - SLOW
	db.products.getShardVersion() - only for use with sharding
```

```
        db.products.getShardDistribution() - prints statistics about data distribution
        in the cluster
        db.products.getSplitKeysForChunks( <maxChunkSize> ) - calculates split points
        over all chunks and returns splitter function
        db.products.getWriteConcern() - returns the write concern used for any
        operations on this collection, inherited from server/db if set
        db.products.setWriteConcern( <write concern doc> ) - sets the write concern for
        writes to the collection
        db.products.unsetWriteConcern( <write concern doc> ) - unsets the write concern
        for writes to the collection
        db.products.latencyStats() - display operation latency histograms for this
        collection




------------------------------------------------------------------------------------
-----------------
06-crud-operations-advanced
------------------------------------------------------------------------------------
-----------------

    --------------------------CREATE-----------------------

    insert() --> insert also exist one many document.But it's  not recommended to use
    it anymore - it also does not return the inserted id's

    > use user
    switched to db user
    > db.persons.insert({name: 'Phil', age: 35})
    WriteResult({ "nInserted" : 1 })
    > db.persons.find()
    { "_id" : ObjectId("5f151e97e3242ab6a2f87b4e"), "name" : "Phil", "age" : 35 }
    > db.persons.find().pretty()
    {
        "_id" : ObjectId("5f151e97e3242ab6a2f87b4e"),
        "name" : "Phil",
        "age" : 35
    }


    > db.persons.insert([{name: 'Khil', age: 45},{name: 'RAJU', age: 22}])
    BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
    })

    > db.persons.find().pretty()
    {
        "_id" : ObjectId("5f151e97e3242ab6a2f87b4e"),
        "name" : "Phil",
        "age" : 35
    }
    {
        "_id" : ObjectId("5f15209ae3242ab6a2f87b4f"),
        "name" : "Khil",
        "age" : 45
    }
    {
        "_id" : ObjectId("5f15209ae3242ab6a2f87b50"),
        "name" : "RAJU",
        "age" : 22
    }
```

-----------------------working with order insert--------------------------------

1. By default, when using insertMany(), inserts are ordered, that means, that the inserting process stops if an error occurs.
2. Can changes this by switching to 'unordered inserts' - inserting process will then continue, even if errors occurred.
3. In both cases, no successful inserts (before the error) will be rolled back.
4. Successful insert will not roll back.

// bulk process

```
> db.hobbies.insertMany([{_id: 'sports', name: 'Sports'},{_id:'cooking',name:'Cooking'},{_id:'cars',name: 'Cars'}])
{
    "acknowledged" : true,
    "insertedIds" : [
        "sports",
        "cooking",
        "cars"
    ]
}
> db.hobbies.find().pretty()
{ "_id" : "sports", "name" : "Sports" }
{ "_id" : "cooking", "name" : "Cooking" }
{ "_id" : "cars", "name" : "Cars" }

> db.hobbies.insertMany([{_id: 'yago', name: 'Yoga'},{_id:'cooking',name:'Cooking'}])
2020-07-20T11:19:04.791+0600 E QUERY    [thread1] BulkWriteError: write error at item 1 in bulk operation :
BulkWriteError({
    "writeErrors" : [
        {
            "index" : 1,
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection: user.hobbies index: _id_ dup key: { : \"cooking\" }",
            "op" : {
                "_id" : "cooking",
                "name" : "Cooking"
            }
        }
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 1,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
})
BulkWriteError@src/mongo/shell/bulk_api.js:369:48
BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:333:24
Bulk/this.execute@src/mongo/shell/bulk_api.js:1177:1
DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:314:5
@(shell):1:1
```

// every element is inserted and standalone but if an error creates then exact element and after are is exited.

```
> db.hobbies.find().pretty()
{ "_id" : "sports", "name" : "Sports" }
{ "_id" : "cooking", "name" : "Cooking" }
{ "_id" : "cars", "name" : "Cars" }
{ "_id" : "yago", "name" : "Yoga" }
```

```
> db.hobbies.insertMany([{_id: 'yago', name:
'Yoga'},{_id:'cooking',name:'Cooking'},{_id:'hiking',name:'Hiking'}],{ordered: true})
2020-07-20T11:33:12.922+0600 E QUERY    [thread1] BulkWriteError: write error at
item 0 in bulk operation :
BulkWriteError({
    "writeErrors" : [
        {
            "index" : 0,
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection: user.hobbies index:
            _id_ dup key: { : \"yago\" }",
            "op" : {
                "_id" : "yago",
                "name" : "Yoga"
            }
        }
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 0,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
})
BulkWriteError@src/mongo/shell/bulk_api.js:369:48
BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:333:24
Bulk/this.execute@src/mongo/shell/bulk_api.js:1177:1
DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:314:5
@(shell):1:1

// now one element is inserted

> db.hobbies.insertMany([{_id: 'yago', name:
'Yoga'},{_id:'cooking',name:'Cooking'},{_id:'hiking',name:'Hiking'}],{ordered:
false})
2020-07-20T11:33:46.532+0600 E QUERY    [thread1] BulkWriteError: 2 write errors in
bulk operation :
BulkWriteError({
    "writeErrors" : [
        {
            "index" : 0,
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection: user.hobbies index:
            _id_ dup key: { : \"yago\" }",
            "op" : {
                "_id" : "yago",
                "name" : "Yoga"
            }
        },
        {
            "index" : 1,
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection: user.hobbies index:
            _id_ dup key: { : \"cooking\" }",
            "op" : {
                "_id" : "cooking",
                "name" : "Cooking"
            }
        }
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 1,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
})
BulkWriteError@src/mongo/shell/bulk_api.js:369:48
BulkWriteResult/this.toError@src/mongo/shell/bulk_api.js:333:24
Bulk/this.execute@src/mongo/shell/bulk_api.js:1177:1
DBCollection.prototype.insertMany@src/mongo/shell/crud_api.js:314:5
@(shell):1:1

> db.hobbies.find().pretty()
{ "_id" : "sports", "name" : "Sports" }
{ "_id" : "cooking", "name" : "Cooking" }
{ "_id" : "cars", "name" : "Cars" }
{ "_id" : "yago", "name" : "Yoga" }
{ "_id" : "hiking", "name" : "Hiking" }

-----------------------------Write concern--------------------

## Control the "level of guarantee"

client---> MongoDB Server ---> Storage Engine ---> 1. Memory 2.Data on Disk

e.g insertOne()| --> {w: 1, j: undefined}
               | --> {w: 1, j: true} --> greater security that this will happen
               |                         and succeed even if the server should face
               issues
               | --> {w: 1, wtimeout: 200, j: true} --> this simply means which
               time frame do you give your
                                                        server to report a success
                                                        for this write before you
                                                        cancel it


w--> write -> write : 1 means should accepted to write
## In write the number means how many instances you want this write to be
acknowledged. With 1 is the default. So the storage engine is aware of it and will
eventually write to the disk.

j--> Journal('Todos') --> the journal is an additional file which the storage
engine manages is like a To-Do file. It works when if server is down for some
reason then file is still there.If the restart the server or if it recovers
basically.

## Backup todo list if server is down

> db.persons.insertOne({name: 'Chrissy', age: 44},{ writeConcern: {w: 0} })
{ "acknowledged" : false }

> db.persons.find()
{ "_id" : ObjectId("5f151e97e3242ab6a2f87b4e"), "name" : "Phil", "age" : 35 }
{ "_id" : ObjectId("5f15209ae3242ab6a2f87b4f"), "name" : "Khil", "age" : 45 }
{ "_id" : ObjectId("5f15209ae3242ab6a2f87b50"), "name" : "RAJU", "age" : 22 }
{ "_id" : ObjectId("5f154012e3242ab6a2f87b52"), "name" : "Chrissy", "age" : 44 }


// data is stored but acknowledged is false.You sent the request but you don't know
if it reached the server.If any network connections issue create. W:0 is super fast
but obviously, it tells you nothing about whether this succeed or not.

// write : the default is true

> db.persons.insertOne({name: 'Alex', age: 35},{writeConcern: {w: 1}})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f15415de3242ab6a2f87b53")
}
> db.persons.find()
{ "_id" : ObjectId("5f151e97e3242ab6a2f87b4e"), "name" : "Phil", "age" : 35 }
{ "_id" : ObjectId("5f15209ae3242ab6a2f87b4f"), "name" : "Khil", "age" : 45 }
{ "_id" : ObjectId("5f15209ae3242ab6a2f87b50"), "name" : "RAJU", "age" : 22 }
{ "_id" : ObjectId("5f154012e3242ab6a2f87b52"), "name" : "Chrissy", "age" : 44 }
{ "_id" : ObjectId("5f15415de3242ab6a2f87b53"), "name" : "Alex", "age" : 35 }

// journal : default is false or undefined

> db.persons.insertOne({name: 'Michel', age: 35},{writeConcern: {w: 1, j: false}})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f154215e3242ab6a2f87b54")
}

// if journal is true then it could be little bit slower

> db.persons.insertOne({name: 'Michela', age: 35},{writeConcern: {w: 1, j: true}})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f154255e3242ab6a2f87b55")
}

> db.persons.insertOne({name: 'Aliya', age: 35},{writeConcern: {w: 1, j: true,wtimeout: 200}})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f1542cee3242ab6a2f87b56")
}

// it super fast
// cause an issue if network connection is slow
> db.persons.insertOne({name: 'Aliya', age: 35},{writeConcern: {w: 1, j: true,wtimeout: 1}})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f154309e3242ab6a2f87b57")
}


--------------------------Atomicity-------------------------

1. Operation (e.g. insertOne()) --> Error --> Rolled Back(i.e NOTHING is saved)
2. Operation (e.g. insertOne()) --> Success --> Saved as Whole

## The Atomicity guarantees that an atomic transaction which means the transaction either succeeds as a whole or it fails as a whole.

## I it fails during the write, everything is rolled back for this document that are inserted.

## Its on a per document level, that means the top level document, it includes all embedded documents, all arrays so that is all included.

## MongoDB CRUD operation are Atomic on the Document Level(including Embedded Documents).


----------------------------------------------------------------------------------------------------

## If you use insert many with multiple documents being inserted, then you don't get this.

## If you have multiple documents in one operation, like insert many, the only each document on its own is guaranteed to either fail or succeed but not insert many.

## Here does not roll back if any occurs create in one document.

-------------------------Importing Data-------------------------

mongoimport tv-shows.json -d movieData -c movies --jsonArray --drop

```

```
    --jsonArray -> to make the mongo import command aware of this.
    --drop -> collection should already exist, it will dropped and then re-added
    otherwise it we'll append the data to the existing collection and that might also
    be what you want.




------------------------------------------------------------------------------------------
-----------------
06-crud-operations-advanced
7-read(part-1)
------------------------------------------------------------------------------------------
-----------------

    -------------------------READ-------------------------

    1. Methods, Filters & Operations
    2. Query Selectors (READ)
    3. Projection Operators(READ)

    Sample Example :

    1. db.myCollection.find({age: 30)
    here {age: 30 } --> Filter . age -> Field, 32 --> Value

    2. db.myCollection.find({age: { $gt: 30}})
    {age: { $gt: 30}} --> Filter(Range) &gt --> Operator

    ------------------- Operator--------------------

           Read                Update

     Query & Projection        Update         Query Modifiers          Aggregation

      Query Selectors--------->Fields-------->Change Query------------>Pipeline
      Stages
                                                Behaviors
     Projection Operators----->Arrays--------->   |       ----------->Pipeline Operators
                                                  |
                                           This is Deprecated now

    ----------------How Operators Impact Our Data-----------------

          Type                 Purpose                    Changes Data?        Example

      Query Operator--------->Locate Data------------------> blocked-------------> $eq
     Projection Operator---->Modify data presentation------> blocked-------------> $
      Update Operator-------> Modify + add additional------> not blocked---------> &inc
                                     data


    ----------------------Query Selectors types----------------------

    1. Comparison 2. Logical 3. Element 4. Evaluation 5. Array 6. Comments 7.
    Geaspatial(special)

    ------------------- Projections Operators---------------------

    1. $ 2. $elemMatch 3. $meta 4. $slice


    > use movieData
    switched to db movieData
    > cls

    > db.movies.findOne()
    {
        "_id" : ObjectId("5f15a22a9bfbc37d06f66616"),
        "id" : 1,
        "url" : "http://www.tvmaze.com/shows/1/under-the-dome",
        "name" : "Under the Dome",
        "type" : "Scripted",
        "language" : "English",
        "genres" : [
            "Drama",
            "Science-Fiction",
            "Thriller"
        ],
        "status" : "Ended",
        "runtime" : 60,
        "premiered" : "2013-06-24",
        "officialSite" : "http://www.cbs.com/shows/under-the-dome/",
        "schedule" : {
            "time" : "22:00",
            "days" : [
                "Thursday"
            ]
        },
        "rating" : {
            "average" : 6.5
        },
        "weight" : 91,
        "network" : {
            "id" : 2,
            "name" : "CBS",
            "country" : {
                "name" : "United States",
                "code" : "US",
                "timezone" : "America/New_York"
            }
        },
        "webChannel" : null,
        "externals" : {
            "tvrage" : 25988,
            "thetvdb" : 264492,
            "imdb" : "tt1553656"
        },
        "image" : {
            "medium" : "http://static.tvmaze.com/uploads/images/medium_portrait/0/1.jpg",
            "original" :
            "http://static.tvmaze.com/uploads/images/original_untouched/0/1.jpg"
        },
        "summary" : "<p><b>Under the Dome</b> is the story of a small town that is
        suddenly and inexplicably sealed off from the rest of the world by an enormous
        transparent dome. The town's inhabitants must deal with surviving the
        post-apocalyptic conditions while searching for answers about the dome, where
        it came from and if and when it will go away.</p>",
        "updated" : 1529612668,
        "_links" : {
            "self" : {
                "href" : "http://api.tvmaze.com/shows/1"
            },
            "previousepisode" : {
                "href" : "http://api.tvmaze.com/episodes/185054"
            }
        }
    }

    ---------------------- 1.Comparison------------------------------

    $ne, $eq, $lt, $lte, $gt, $gte, $in, $nin

    > db.movies.find({runtime: 60}).pretty()
    > db.movies.findOne({runtime: 60})

    // exactly the same
    >db.movies.findOne({runtime: {$eq: 60}})
```

```
> db.movies.find({runtime: {$ne: 60}}).pretty()
> db.movies.find({runtime: {$lt: 40}}).pretty()
> db.movies.find({runtime: {$lte: 40}}).pretty()
> db.movies.find({runtime: {$gt: 40}}).pretty()
> db.movies.find({runtime: {$gte: 40}}).pretty()

// query into imbedded documents
N.B : In imbedded documents have to use must quotes

> db.movies.find({"rating.average": {$gt: 7}}).pretty()

// query into imbedded array

> db.movies.find({genres:"Drama"}).pretty()
// to exact query
> db.movies.find({genres:["Drama"]}).pretty()

// it will find us all documents that have a runtime of 30 or 42 but not 60

[30,42] --> this is set of values not a range
> db.movies.find({runtime: {$in:[30,42]}}).pretty()
> db.movies.find({runtime: {$nin:[30,42]}}).pretty()


------------------------ 2.Logical--------------------------

$or, $and, $not, $nor

// multiple query

> db.movies.find({$or: [{"rating.average": {$lt: 5}},{"rating.average": {$gt: 
9.3}}]}).count()
> db.movies.find({$or: [{"rating.average": {$lt: 5}},{"rating.average": {$gt: 
9.3}}]}).pretty()

> db.movies.find({$nor: [{"rating.average": {$lt: 5}},{"rating.average": {$gt: 
9.3}}]}).count()

// this is the older command
> db.movies.find({$and: [{"rating.average": {$gt: 9}},{genres : "Drama"}]}).count()
> db.movies.find({$and: [{"rating.average": {$gt: 9}},{genres : "Drama"}]}).pretty()

// latest command (using only document)
> db.movies.find({"rating.average": {$gt: 9}, genres : "Drama"}).count()

// this basically not work, does not give right value
// same object is not permitted int this way
// here issue is create by same json key genres, this keys value replace the first 
one when execute second

> db.movies.find({genres : "Drama", genres: 'Horror'}).count()

// this also have same result
> db.movies.find({genres: 'Horror'}).count()



// we have to use and in the same field
// in this issue we have to use $and must
> db.movies.find({$and: [{genres : "Drama"}, {genres: 'Horror'}]}).count()
> db.movies.find({runtime: {$not :{$eq: 60}}}).count()

// this is also equal to the $ne
> db.movies.find({runtime: {$ne: 60}}).count()

------------------------------ 3.Element---------------------------
$exists $type
```

```
// exists

> db.users.insertMany([{name: 'Max', hobbies: [{title: 'Sports', frequency: 3},{title: 'Cooking', frequency: 6}], phone: 0123495334},{name: 'Manuel', hobbies: [{title: 'Cooking', frequency: 5},{title: 'Cars', frequency: 6}], phone: '043453495334', age: 30}])
{
    "acknowledged" : true,
    "insertedIds" : [
        ObjectId("5f172a343a76a40cd42b836a"),
        ObjectId("5f172a343a76a40cd42b836b")
    ]
}
> db.users.find().pretty()
{
    "_id" : ObjectId("5f172a343a76a40cd42b836a"),
    "name" : "Max",
    "hobbies" : [
        {
            "title" : "Sports",
            "frequency" : 3
        },
        {
            "title" : "Cooking",
            "frequency" : 6
        }
    ],
    "phone" : 123495334
}
{
    "_id" : ObjectId("5f172a343a76a40cd42b836b"),
    "name" : "Manuel",
    "hobbies" : [
        {
            "title" : "Cooking",
            "frequency" : 5
        },
        {
            "title" : "Cars",
            "frequency" : 6
        }
    ],
    "phone" : "043453495334",
    "age" : 30
}

// here checking an element exists or not
> db.users.find({age: {$exists: true}}).pretty()
{
    "_id" : ObjectId("5f172a343a76a40cd42b836b"),
    "name" : "Manuel",
    "hobbies" : [
        {
            "title" : "Cooking",
            "frequency" : 5
        },
        {
            "title" : "Cars",
            "frequency" : 6
        }
    ],
    "phone" : "043453495334",
    "age" : 30
}

// can also check multiple logic
> db.users.find({age: {$exists: true, $gt: 30}}).pretty()
> db.users.find({age: {$exists: true, $gte: 30}}).pretty()
```

```
    // if element value is null then it also be exists
    > db.users.insertMany([{name: 'Anna', hobbies: [{title: 'Sports', frequency: 2},{title: 'Yoga', frequency: 3}], phone: 01234953345, age: null}])
    {
        "acknowledged" : true,
        "insertedIds" : [
            ObjectId("5f172c593a76a40cd42b836c")
        ]
    }
    > db.users.find({age: {$exists: true}}).pretty()

    // but if we check with exists value is false and if an element value has null but exist then it also does not show
    > db.users.find({age: {$exists: false}}).pretty()

    // checking exit and value not equal null
    > db.users.find({age: {$exists: true, $ne: null}}).pretty()

    // type
    Type                Number       Alias              Notes
    Double             1          "double"
    String             2     "string"
    Object             3     "object"
    Array                  4     "array"
    Binary data            5     "binData"
    Undefined          6     "undefined"           Deprecated.
    ObjectId           7     "objectId"
    Boolean            8     "bool"
    Date                   9     "date"
    Null                  10     "null"
    Regular Expression   11     "regex"
    DBPointer         12     "dbPointer"           Deprecated.
    JavaScript        13     "javascript"
    Symbol                14     "symbol"          Deprecated.
    JavaScript (with scope)15   "javascriptWithScope"
    32-bit integer       16     "int"
    Timestamp         17     "timestamp"
    64-bit integer       18     "long"
    Decimal128        19     "decimal"          New in version 3.4.
    Min key           -1     "minKey"
    Max key          127     "maxKey"

    // checking with alias
    > db.users.find({phone: {$type: 'number'}}).pretty()

    // as shell is based on JavaScript number and double would be the same answer
    // In database number is stored into floating point number as double
    // JS drivers only knows it always double

    > db.users.find({phone: {$type: 'double'}}).pretty()

    // also can add multiple type
    > db.users.find({phone: {$type: ['double','string']}}).pretty()




------------------------------------------------------------------------------------------------
06-crud-operations-advanced
7-read(part-2)
------------------------------------------------------------------------------------------------


    --------------------------4.Evaluation Operators---------------------

    $expr, $regex, $text, $where

```

```
// $regex allows us search for text

// return the document that found the word
// it is not  best way to find text in this way
> db.movies.find({summary: {$regex: /musical/}}).pretty()


$expr --> compare two fields inside one document then return that fields

> use financialDatalet retrieve_code = request.params.id;
switched to db financialData
> db.sales.insertMany([{volume: 100, target: 120},{volume: 89, target: 80},{volume:
200, target: 177}])

> db.sales.find().pretty()
{
    "_id" : ObjectId("5f17491c3a76a40cd42b836d"),
    "volume" : 100,
    "target" : 120
}
{
    "_id" : ObjectId("5f17491c3a76a40cd42b836e"),
    "volume" : 89,
    "target" : 80
}
{
    "_id" : ObjectId("5f17491c3a76a40cd42b836f"),
    "volume" : 200,
    "target" : 177
}

## we want to find all entries, all items in this collection where the volume is
above the target

// we have to use double quotes in to query, have to pass reference fields name
// have to use dollar sign before fields also

// this will not work
> db.sales.find({$expr: {$gt: ['volume', 'target']}}).pretty()

// this work successfully
> db.sales.find({$expr: {$gt: ['$volume', '$target']}}).pretty()

{
    "_id" : ObjectId("5f17491c3a76a40cd42b836e"),
    "volume" : 89,
    "target" : 80
}
{
    "_id" : ObjectId("5f17491c3a76a40cd42b836f"),
    "volume" : 200,
    "target" : 177
}

## we do not want to compare whether volume is greater than target and also (want
to find if volume is above 190 and the difference to target at least 10)

// this is more complex query
$cond --> conditional because we are in document

> db.sales.find({$expr: {$gt: [{$cond: {if: {$gte: ['$volume', 190]}, then:
{$subtract: ['$volume', 10]}, else: '$volume'}}, '$target']}}).pretty()
{
    "_id" : ObjectId("5f17491c3a76a40cd42b836e"),
    "volume" : 89,
    "target" : 80
}
{
    "_id" : ObjectId("5f17491c3a76a40cd42b836f"),
    "volume" : 200,
    "target" : 177
}

// if i increase the subtracted value logically then result might be changed

> db.sales.find({$expr: {$gt: [{$cond: {if: {$gte: ['$volume', 190]}, then: 
{$subtract: ['$volume', 30]}, else: '$volume'}}, '$target']}}).pretty()
{
    "_id" : ObjectId("5f17491c3a76a40cd42b836e"),
    "volume" : 89,
    "target" : 80
}


------------------------5.Array------------------------

$elemMatch, $size , $all

> use user
switched to db user

// find all hobbies that are sports
// this won't work
> db.users.find({hobbies: 'Sports'}).pretty()

// also find nothing when using nested document
> db.users.find({hobbies: {title:'Sports'}}).pretty()

// this also can not a perfect value
> db.users.find({hobbies: {title:'Sports', frequency: 2}}).pretty()

// act an embedded document
// this is path embedded approach not only on a directly embedded documents
// this is similar to multiple embedded documents query

> db.users.find({'hobbies.title': 'Sports'}).pretty()
{
    "_id" : ObjectId("5f172a343a76a40cd42b836a"),
    "name" : "Max",
    "hobbies" : [
        {
            "title" : "Sports",
            "frequency" : 3
        },
        {
            "title" : "Cooking",
            "frequency" : 6
        }
    ],
    "phone" : 123495334
}
{
    "_id" : ObjectId("5f172c593a76a40cd42b836c"),
    "name" : "Anna",
    "hobbies" : [
        {
            "title" : "Sports",
            "frequency" : 2
        },
        {
            "title" : "Yoga",
            "frequency" : 3
        }
    ],
    "phone" : 1234953345,
    "age" : null
}
```

```
## want to find all users who have exactly 3 hobbies
> db.users.insertOne({name: 'Chris', hobbies: ['Sports', 'Cooking', 'Hiking']})
> db.users.find({'hobbies': {$size: 3}}).pretty()

## if want to query like hobbies greater than 3 or smaller. It does not support mongoDB
> use boxOffice
> db.moviestarts.find().pretty()

## want to find movies that have a genre of exactly thriller and action
// this won't work perfectly, here basically work with index ordering and also exact
// we are not concern about ordering
> db.moviestarts.find({genre: ['action', 'thriller']}).pretty()

// $all basically find if array have all fields like 'action', 'thriller'
> db.moviestarts.find({genre: {$all: ['action', 'thriller']}}).pretty()


## want to find all users who have a hobby of sports and the frequency should be grate or equal to 2
> db.users.find({$and: [{'hobbies.title': 'Sports'},{'hobbies.frequency': {$gte:2}}]}).pretty()

// if we change the query replace 2 with 3. does not work properly

{
    "_id" : ObjectId("5f172a343a76a40cd42b836a"),
    "name" : "Max",
    "hobbies" : [
        {
            "title" : "Sports",
            "frequency" : 3
        },
        {
            "title" : "Cooking",
            "frequency" : 6
        }
    ],
    "phone" : 123495334
}
{
    "_id" : ObjectId("5f172c593a76a40cd42b836c"),
    "name" : "Anna",
    "hobbies" : [
        {
            "title" : "Sports",
            "frequency" : 2
        },
        {
            "title" : "Yoga",
            "frequency" : 3
        }
    ],
    "phone" : 1234953345,
    "age" : null
}
// this work with different embedded document but we do not want that
// does not give the exact result
> db.users.find({$and: [{'hobbies.title': 'Sports'},{'hobbies.frequency': {$gte:3}}]}).pretty()

{
    "_id" : ObjectId("5f172a343a76a40cd42b836a"),
    "name" : "Max",
    "hobbies" : [
        {
            "title" : "Sports",
            "frequency" : 3
        },
        {
            "title" : "Cooking",
            "frequency" : 6
        }
    ],
    "phone" : 123495334
}
{
    "_id" : ObjectId("5f172c593a76a40cd42b836c"),
    "name" : "Anna",
    "hobbies" : [
        {
            "title" : "Sports",
            "frequency" : 2
        },
        {
            "title" : "Yoga",
            "frequency" : 3
        }
    ],
    "phone" : 1234953345,
    "age" : null
}
> db.users.find({$and: [{'hobbies.title': 'Sports'},{'hobbies.frequency':
{$gte:3}}]}).pretty().count()

// we want to ensure that query have to perform into same document/element
// work properly
// perform query into single document

> db.users.find({hobbies: {$elemMatch:{title: 'Sports', frequency: {$gte:
3}}}}).pretty().pretty()
{
    "_id" : ObjectId("5f172a343a76a40cd42b836a"),
    "name" : "Max",
    "hobbies" : [
        {
            "title" : "Sports",
            "frequency" : 3
        },
        {
            "title" : "Cooking",
            "frequency" : 6
        }
    ],
    "phone" : 123495334
}


------------------------Understanding Cursors------------------------

// when we find() it basically getting the all data like 100 millions
// it can reduce if we include query

// here cursors basically a pointer
// cursor request batch to the server every time to get tha data

// If have a query that meets 1000 documents, but let's say you have a website
where you only display 10 items, let's say 10 products you fetched at a time
anyways, then there is no need to load all thousand results that matched your query
right at the start.Instead you would only fetch the first 10,display them on the
screen and then go ahead and fetch the next 10,when the user navigated to the next
page or anything like that. This is the idea behind a cursor.

> use movieData
> db.movies.find().count()
240

// basically it returns 20 elements
```

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

// type it for more
> it

// get exactly one document, because next() gives the next document
> db.movies.find().next()

// we can use JavaScript syntax in mongoShell
> const dataCursor = db.movies.find()
> dataCursor.next()
> dataCursor.next()

// printjson() is a mongoShell function that helps to print something into shell

// fetched all documents
> dataCursor.forEach(document => {printjson(document)})

// check have any next value
> dataCursor.hasNext()

// fetching data with sort()
// one means ascending
> db.movies.find().sort({'rating.average': 1}).pretty()

// minus one mean descending
> db.movies.find().sort({'rating.average': -1}).pretty()

// sort with multiple query
// here first sort the 'rating.average' and if 'rating.average' have same value
into particular indexes and then runtime execute with descending if may exist
> db.movies.find().sort({'rating.average': 1, runtime: -1}).pretty()

// next() also exist with sort()
> db.movies.find().sort({'rating.average': 1, runtime: -1}).next()

// we can skip certain amount of elements
// it effective in pagination
// when we work with pagination we can skip the previous 10 elements
> db.movies.find().sort({'rating.average': 1, runtime: -1}).skip(10).pretty()

// skip() basically limit the amount of elements the cursor should retrieve at a time
// can still have include limit

// limit return the exact number of element
> db.movies.find().sort({'rating.average': 1,runtime:
-1}).skip(100).limit(10).pretty()

// here order does not matter.
// Order check from right such(previousexample) : sort()->skip()->limit()
// what method we write first, it execute first


------------------Using Projection to Share our Results---------------------

## want to retrieve elements with specific fields
// we have no to check the other fields. they are executed by default

// here ID always include
> db.movies.find({}, {name: 1, genres: 1, runtime: 1, rating: 1}).pretty()

// to exclude the ID
> db.movies.find({}, {name: 1, genres: 1, runtime: 1, rating: 1, _id: 0}).pretty()

// can also use embedded document with path notation
> db.movies.find({}, {name: 1, genres: 1, runtime: 1, 'rating.average': 1, _id:
0}).pretty()

> db.movies.find({}, {name: 1, genres: 1, runtime: 1, 'rating.average': 1,
'schedule.time': 1,_id: 0}).pretty()

// can also add logic
> db.movies.find({'rating.average': {$gt: 8}}, {name: 1, genres: 1, runtime: 1,
'rating.average': 1, 'schedule.time': 1,_id: 0}).pretty()


-----------------------Projection in Arrays----------------------

// simply array query
> db.movies.find({genres: 'Drama'}).pretty()

// return the array projection of related query
> db.movies.find({genres: 'Drama'},{'genres.$': 1}).pretty()

// it does not work properly
> db.movies.find({genres: {$all: ['Drama', 'Horror']}},{'genres.$': 1}).pretty()

// this projection is element wise and exact query
// {$elemMatch: {$eq: 'Horror'}} --> this thing decide which item is displayed or not
> db.movies.find({genres: 'Drama'},{genres: {$elemMatch: {$eq: 'Horror'}}}).pretty()

// can also check with other element
> db.movies.find({'rating.average':{$gt: 9}},{genres: {$elemMatch: {$eq:
'Horror'}}}).pretty()


----------------------Projection Slice--------------------

// slicing array that i want
// can add any number
// $slice: 2 --> how many array elements we want to show from first
> db.movies.find({'rating.average':{$gt: 9}}, {genres: {$slice: 2}, name:
1}).pretty()

// can also be execute with array from
// 1--> What lengths of elements we want to skip (index - start from 1)
// 2--> How many element we want to show
> db.movies.find({'rating.average':{$gt: 9}}, {genres: {$slice: [1, 2]}, name:
1}).pretty()
// checking
> db.movies.find({_id: ObjectId("5f15a22a9bfbc37d06f66662")},{genres: 1}).pretty()

{
    "_id" : ObjectId("5f15a22a9bfbc37d06f66662"),
    "genres" : [
        "Drama",
        "Adventure",
        "Fantasy"
    ]
}

> db.movies.find({'rating.average':{$gt: 9}}, {genres: {$slice: [2, 2]}, name:
1}).pretty()

{
    "_id" : ObjectId("5f15a22a9bfbc37d06f6662d"),
    "name" : "Berserk",
    "genres" : [
        "Horror"
    ]
}
{
    "_id" : ObjectId("5f15a22a9bfbc37d06f66662"),
    "name" : "Game of Thrones",
    "genres" : [
        "Fantasy"
    ]
}
```

```
    {
        "_id" : ObjectId("5f15a22a9bfbc37d06f666b7"),
        "name" : "Breaking Bad",
        "genres" : [
            "Thriller"
        ]
    }
    {
        "_id" : ObjectId("5f15a22a9bfbc37d06f666c0"),
        "name" : "The Wire",
        "genres" : [ ]
    }
    {
        "_id" : ObjectId("5f15a22a9bfbc37d06f666c1"),
        "name" : "Firefly",
        "genres" : [
            "Western"
        ]
    }
    {
        "_id" : ObjectId("5f15a22a9bfbc37d06f666d8"),
        "name" : "Stargate SG-1",
        "genres" : [
            "Science-Fiction"
        ]
    }
    {
        "_id" : ObjectId("5f15a22a9bfbc37d06f666e2"),
        "name" : "Rick and Morty",
        "genres" : [
            "Science-Fiction"
        ]
    }



-------------------------------------------------------------------------------------------------------------
06-crud-operations-advanced
8-update
-------------------------------------------------------------------------------------------------------------

    --------------------------Update--------------------------

    1. Document Updating Operator 2. Updating Fields 3. Updating Arrays

    ---------------------1. Document Updating Operator-----------------------

    $min, $max, $mul, $inc, $set, $unset

    // set basically changed or added new document

    > use user
    switched to db user

    > db.users.find().pretty()

    > db.users.updateOne({_id: ObjectId("5f17c3d47122dce4fa46fb4a")}, {$set: {hobbies:[{title: 'Sports', frequency: 5},{title: 'Cooking', frequency: 3}, {title: 'Hiking', frequency: 1}]}})

    // $set basically add a new value or update existing value
    // update the users whose hobby is Sports
    > db.users.updateMany({'hobbies.title': 'Sports'}, {$set: {isSporty: true}})

    // adding multiple elements using $set
    > db.users.updateOne({_id: ObjectId("5f17c3d47122dce4fa46fb4a")}, {$set: {age: 40, phone: 082344289399}})
```

```
// can manually increment or decrement any number document
> db.users.updateOne({name: "Manuel"}, {$inc: {age: 2}})
> db.users.updateOne({name: "Manuel"}, {$inc: {age: -2}})

// also check with multiple query
> db.users.updateOne({name: "Manuel"}, {$inc: {age: -2}, $set:{isSporty: false}})

// two operations is not allowed into same fields.
> db.users.updateOne({name: "Manuel"}, {$inc: {age: -2}, $set:{age: 30}})

// update the age value with min value
> db.users.updateOne({name: "Chris"}, {$min: {age: 35}})
> db.users.updateOne({name: "Chris"}, {$max: {age: 39}})

// multiply age by a number specify 10 %
> db.users.updateOne({name: "Chris"}, {$mul: {age: 1.1}})

----------------------2. Updating Fields-----------------------------

$upsert

// Getting Rid of Fields

## want to drop all value on all persons who are sporty
> db.users.updateMany({isSporty: true}, {$set: {phone: null}})

// to get red of fields
> db.users.updateMany({isSporty: true}, {$unset: {phone: null}})

> db.users.updateMany({isSporty: true}, {$unset: {phone: ''}})

// Renaming Fields

$rename, $set, upsert

> db.users.updateMany({}, {$rename: {age: 'totalAge'}})
{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 3 }

## want to update some fields but does not know if its exist or not.
// if it is exist then override the document
// if it does not exist then create new document

// its normal insert and update
> db.users.updateOne({name: 'Maria'}, {$set: {age: 29, hobbies: [{title: 'Good
food', frequency: 3}], isSporty: true}})
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 }

> db.users.updateOne({name: 'Maria'}, {$set: {age: 29, hobbies: [{title: 'Good
food', frequency: 3}], isSporty: true}}, {upsert: false})

// this works perfectly
// work with filter
> db.users.updateOne({name: 'Maria'}, {$set: {age: 29, hobbies: [{title: 'Good
food', frequency: 3}], isSporty: true}}, {upsert: true})

-------------------------3. Updating Arrays---------------------------

.$, $[], $push, $pop, $pull, $each, $addToSet

## want to find a certain amount of persons and persons based on the hobbies array

// checking query into array same elements but not working perfectly
> db.users.find({$and: [{'hobbies.title': 'Sports'},{'hobbies.frequency':
{$gte:3}}]}).pretty()

// this is the exact query
> db.users.find({hobbies: {$elemMatch: {title: 'Sports',frequency:
{$gte:3}}}}).pretty()
```

```
// updating hole matched array elements
// here .$ -> this will automatically refer to the element in our filter  as i want to update the element in hobbies which matched the condition
// $--> dollar sign is a place holder here

// adding new field
> db.users.updateMany({hobbies: {$elemMatch: {title: 'Sports',frequency: {$gte:3}}}}, {$set : {"hobbies.$.highFrequency": true}})

// updating All Array Elements
> db.users.find({'hobbies.frequency': {$gt: 2}}).pretty()
> db.users.find({'hobbies.frequency': {$gt: 2}}).count()

// updating the elements
// but this won't work properly
> db.users.updateMany({'hobbies.frequency': {$gt: 2}}, {$set: {'hobbies.$.goodFrequency': true}})

## lets say if totalAge is greater than 30 than we want to update the every array elements
// do not override
// .$[] --> update all array elements and for each element because we have embedded document.
> db.users.updateMany({totalAge: {$gt: 30}}, {$inc: {'hobbies.$[].frequency': -1}})

## want to find all hobbies with a frequency greater than 2
> db.users.find({'hobbies.frequency': {$gt: 2}}).pretty()

// el --> is a identifier for every items of array into documents
// {'hobbies.frequency': {$gt: 2}} --> this filter identify documents
// {'el.frequency': {$gt: 2}} --> this filter which identify array elements
// these two are not equal

> db.users.updateMany({'hobbies.frequency': {$gt: 2}}, {$set: {'hobbies.$[el].goodFrequency': true}}, {arrayFilters: [{'el.frequency': {$gt: 2}}]})
{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 2 }

// Adding Elements to Arrays
// taking also old array
> db.users.updateOne({name: 'Maria'}, {$push: {hobbies: {title: 'Sports', frequency: 2}}})

// $push also used with more than one document
// also use $sort, $sort is related with every $each
> db.users.updateOne({name: 'Maria'}, {$push: {hobbies: {$each: [{title: 'Good Wine', frequency: 1}, {title: 'Good Wine', frequency: 2}], $sort: {frequency: -1}}}})

// Removing Elements from array
// $pull describe an object that what we want to pull
> db.users.updateOne({name: 'Maria'}, {$pull: {hobbies: {title: 'Hiking'}}})

// Remove the last element of an array
> db.users.updateOne({name: 'Chris'}, {$pop: {hobbies: 1}})

// Remove the first element of an array
> db.users.updateOne({name: 'Chris'}, {$pop: {hobbies: -1}})

// Understanding $addToSet
// $addToSet adds unique value only
// its basically add new element.But if the element already exist(have to exact same) it does not update.
> db.users.updateOne({name: 'Maria'}, {$addToSet: {hobbies: {title: 'Hiking', frequency: 2}}})



-----------------------------------------------------------------------------------------------
```

06-crud-operations-advanced
9-delete

```
    --------------------------Delete--------------------------

    > use user
    switched to db user
    > db.users.deleteOne({name: 'Chris'})

    // delete with matched query
    > db.users.deleteOne({totalAge: {$gt: 30}, isSporty: true})

    > db.users.deleteOne({totalAge: {$exists: false}, isSporty: true})

    // delete many
    > db.users.deleteMany({totalAge: {$gt: 30}, isSporty: true})

    // deleting all entries in a collection
    // {} --> this is simply is a filter that matches every document in the collection.
    > db.users.deleteMany({})

    // to delete the entire collection
    > db.users.drop()

    // to delete the entire dataBase
    > db.dropDataBase()
```

10-working-with-indexes
credit-rating

```
    conn = new Mongo();
    db = conn.getDB("credit");

    for (let i = 0; i < 1000000; i++) {
        db.ratings.insertOne({
            "person_id": i + 1,
            "score": Math.random() * 100,
            "age": Math.floor(Math.random() * 70) + 18
        })
    }
```

10-working-with-indexes

```
    ---------------------Index and others----------------------

    // different Types of Indexes
    // Using and Optimizing Indexes
    // indexes are order list of values
    // Its point related index just like a pointer indexes
    // Indexes are updated with every insert

    ----------- What ant Why -------------
    1. Indexes allow to retrieve data more efficiently (if used correctly) because
    queries only have to look at a subset of all documents.
```

```
2. Can use single-field, compound, multi-key(array) and text indexes.

3. Indexes don't come for free, they will slow down writes.
---------- Queries & Sorting -----------
4. Indexes can be used for both queries and efficient sorting.

5. Compound indexes can be used as a whole or in a 'left-to-right' (prefix) manner
(e.g only consider the 'name' of the 'name-age' compound index)


// Adding a Single Field Index
> use contactData
> db.contacts.find({'dob.age': {$gt: 60}}).pretty()

//analyze database with explain() method
> db.contacts.explain().find({'dob.age': {$gt: 60}})

// here also have a different types of plan --> 1. winningPlan 2. rejectedPlans
// getting the detailed query
> db.contacts.explain("executionStats").find({'dob.age': {$gt: 60}})

// creating index, index is defined as a document
// here one basically means i want to sort the data with ascending order
> db.contacts.createIndex({'dob.age': 1})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}

// if change the query then it works like before
// Without index every different query ha different values
> db.contacts.explain("executionStats").find({'dob.age': {$gt: 20}})

// dropping the index
> db.contacts.dropIndex({'dob.age': 1})
{ "nIndexesWas" : 2, "ok" : 1 }

// Understanding Index Restrictions
> db.contacts.explain("executionStats").find({'dob.age': {$gt: 20}})

// if have to retrieve large number of documents nearly 70 to 80% then index can
effect the query be slower,cause for using query we have to add an extra step.

// for retrieving 20-30% or lower can using index, query be faster
// creating a compound index with text
> db.contacts.createIndex({gender: 1})
// get explain after creating index
> db.contacts.explain("executionStats").find({gender: "male"})

## want to find all persons who are older than 30 and male or older than 40 and male
// when using multiple fields for query in index, basically one combined index is
created from multiple fields.
// here created one combined index from two fields
// every time have to drop if uses the same filed
> db.contacts.dropIndex({'gender': 1})
> db.contacts.createIndex({'dob.age': 1,'gender': 1})

// getting info from query
> db.contacts.explain().find({'dob.age': 35,'gender': 'male'})

// if want to execute single index from multiple combined index its work left to
right
"indexName" : "dob.age_1_gender_1"

// it works fine, means it works with index scan
> db.contacts.explain().find({'dob.age': 35})
```

```
// it does not work properly means, it works with colum scan
> db.contacts.explain().find({'gender': 'male'})

// using indexes for sorting
// this query also works like indexes
// this also use index scan
"indexName" : "dob.age_1_gender_1",
> db.contacts.explain().find({'dob.age': 35}).sort({gender: 1})

// mongo db reserves 32mb for fetched documents when using sort

// understanding the default index and find how many indexes into documents
> db.contacts.getIndexes()
[
    {
        "v" : 2,
        "key" : {
            "_id" : 1
        },
        "name" : "_id_",
        "ns" : "contactData.contacts"
    },
    {
        "v" : 2,
        "key" : {
            "dob.age" : 1,
            "gender" : 1
        },
        "name" : "dob.age_1_gender_1",
        "ns" : "contactData.contacts"
    }
]
---------------------configuring Indexes------------------------
// every indexes _id is a unique by default
// can not add same value into same document
> db.contacts.createIndex({email: 1}, {unique: true})
{
    "ok" : 0,
    "errmsg" : "E11000 duplicate key error collection: contactData.contacts index:
    email_1 dup key: { : \"abigail.clark@example.com\" }",
    "code" : 11000,
    "codeName" : "DuplicateKey"
}
> db.contacts.find({email: 'abigail.clark@example.com'}).count()
// by checking can find unique value exist or not

// Understanding Partial Filters
> db.contacts.dropIndex({'dob.age': 1, gender: 1})

// now create is an index on age, not on gender but on age but only for elements
where the underlying document is for a male.
> db.contacts.createIndex({'dob.age': 1}, {partialFilterExpression:{gender: 'male'}})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}

// can also use age query
> db.contacts.createIndex({'dob.age': 1}, {partialFilterExpression:{'dob.age':
{$gt: 60}}})
// this does not work. because as a partial index we have to also use gender
> db.contacts.explain().find({'dob.age': {$gt: 60}})
// this works with index scan
> db.contacts.explain().find({'dob.age': {$gt: 60}, gender: 'male'})


--------------------applying the Partial Index-----------------------
```

```
> db.users.insertMany([{name: 'Max', email: 'max@test.com'},{name: 'Manu'}])
// implementing unique key with email
> db.users.createIndex({email: 1}, {unique: true})

// if now want to add new user without email, it says duplicate index error, because no values store twice.
> db.users.insertOne({name: 'Anna'})

// but a person could not have email
> db.users.dropIndex({email: 1})

// now we create index a bit differently.
> db.users.createIndex({email: 1},{unique: true, partialFilterExpression: {email: {$exists: true}}})

// now add user without email, it works
> db.users.insertOne({name: 'Anna'})

// now we have three user one have email and others two without email
> db.users.find().pretty()

// as partial index already created we can not add new user with same email
> db.users.insertOne({name: 'Anna', email: 'max@test.com'})

// this section index options
------------------------Understanding the Time-To-Live(TTL) index--------------------

// this works like session
// clear data after some duration
// self destroying data
> db.sessions.insertOne({data: 'Sample data', createdAt: new Date()})
> db.sessions.find().pretty()

// now add time to live index, can create with normal ascending text
> db.sessions.createdIndex({createdAt: 1})
> db.sessions.dropIndex({createdAt: 1})

// add indexes with differently
> db.sessions.createIndex({createdAt: 1}, {expireAfterSeconds: 10})

// after 10 seconds the document will be destroyed
> db.sessions.find().pretty()

-----------------------Query Diagnosis and Query Planing-------------------

// explain() it contains three types of parameter
1. 'queryPlanner' --> Show Summary for Executed Query + Winning Plan
2. 'executionsStats' --> Show Detailed Summary for Executed Query + Winning Plan + Possibly Rejected Plans
3. 'allPlanExecution' --> Show Detailed Summary for Executed Query + Winning Plan + Winning Plan Decision Process

// Efficient Queries and Covered Queries
// Milliseconds Process Time
IXSCAN typically beats(1. of keys(in index) Examined 2. of Documents Examined 3. of Documents Returns) COLLSCAN

// Understanding Covered Queries
> db.customers.insertMany([{name: 'Max', age: 29, salary: 3000}, {name: 'Manu', age: 30, salary: 4000}])

// creating index
> db.customers.createIndex({name: 1})
> db.customers.getIndexes()

// get info
> db.customers.explain('executionStats').find({name: 'Max'})

// lets implement covered queries
// if can optimize query, than have to reach that covered query state
// useful when typically return the specific fields
> db.customers.explain('executionStats').find({name: 'Max'},{_id: 0, name: 1})

------------------How mongoDB rejects a plan------------------

// creating a compound index
// order is important for compound index
// name index here wouldn't make much sense
// if age comes first, we can also filter just for age and take advantage of this index.
// if filtered for just name and didn't have that index, name could not be supported by index.

// here we can use just age or combination of age and name.
> db.customers.createIndex({age: 1, name: 1})

// let execute query, when execute query order does not matter in compound index
> db.customers.explain().find({age: 30, name: 'Max'})
> db.customers.explain().find({name: 'Max', age: 30})

// wining plan
 1. Approach 1
 2. Approach 2
 3. Approach 3 --> winning Plan --> Cached --> Cache --> but cache is not there forever

// Clearing the Winning Plan from Cache

                 | 1. Write Threshold (currently 1,000)
Stored Forever?--| 2. Index is Rebuilt
                 | 3. Other Indexes are Added or Removed
                 | 4. MongoDB Server is Restarted

> db.customers.insertOne({name:'Raju', age:22, salary: 1000})

// get details of all plan
// here we get the all details of plan which be good and execution time
> db.customers.explain('allPlansExecution').find({age: 30, name: 'Max'})

------------------------ Using Multi Key Indexes ------------------------

// insert new data into new table
> db.contactsinfo.insertOne({name:'Max', hobbies:['Cooking', 'Sports'], assress: [{street: 'Main Street'}, {street: 'Second Street'}]})

> db.contactsinfo.findOne()
// create an index
> db.contactsinfo.createIndex({hobbies: 1})
> db.contactsinfo.find({hobbies: 'Sports'}).pretty()

// execute explain
// here multi key is true, it is created when documents into array
> db.contactsinfo.explain('executionsStats').find({hobbies: 'Sports'})

// lets create another index
> db.contactsinfo.createIndex({addresses: 1})

// here index does not work,cause it does not work on nested documents query
> db.contactsinfo.explain('executionStats').find({'addresses.street': 'Main Street'})

// it works when query like
// Basically it works like normal
> db.contactsinfo.explain('executionStats').find({addresses: {street: 'Main Street'}})

// if the index is created like then it works, it also have a multi key index
> db.contactsinfo.createIndex({'addresses.street': 1})
// this is now index scan
```

```
> db.contactsinfo.explain('executionStats').find({'addresses.street': 'Main Street'})

// Still multi key index is super helpful if have queries that regularly target
array values or even nested values or values in an embedded document in arrays.
// There are a couple of restrictions or one important restriction to be precise
when using multi key indexes

// create a multi key compound index, it is also possible, when have one multi key
> db.contactsinfo.createIndex({name: 1, hobbies: 1})

// but parallel arrays can not create multiple compound index
> db.contactsinfo.createIndex({addresses: 1, hobbies: 1})

------------------------- Understanding 'text' indexes----------------------

// this is a special type of multi key index
this product is a must-buy for all fans of modern fiction!
// from the sentence the text index : product,must,buy,fans,modern,fiction
// if an array of single words or array of keywords essentially to search text.

// create a new collections
> db.products.insertMany([{title: 'A book', description: 'This is an awesome book
about a young artist!'}, {title: 'Red T-Shirt', description: 'This T-Shirt is red
and it is pretty awesome'}])

// create an index
// this is a single field index and can search with exact text
> db.products.createIndex({description: 1})

// to create text index to split the sentence
// so drop the previous index
> db.products.dropIndex({description: 1})
// create text index --> special kind of index
// in text index remove all the stop words and store all the keyword into array
essentially
> db.products.createIndex({description: 'text'})

// Now might be wondering why do not need to specify the field in which want to
search pretty expensive as can imagine.
// if have a lot of long text that has to be split up,don't want to do this like 10
times per collection and therefore,only have one text index where this could look
into.

// can actually merge multiple fields into one text index.
// everything is stored as lowercase.
> db.products.find({$text: {$search: 'awesome'}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
    "title" : "Red T-Shirt",
    "description" : "This T-Shirt is red and it is pretty awesome"
}
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book about a young artist!"
}

> db.products.find({$text: {$search: 'book'}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book about a young artist!"
}

// here red into second document and book into first document
> db.products.find({$text: {$search: 'red book'}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book about a young artist!"
}
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
    "title" : "Red T-Shirt",
    "description" : "This T-Shirt is red and it is pretty awesome"
}

// can search with exactly phrase
> db.products.find({$text: {$search: "\"red book\""}}).pretty()
> db.products.find({$text: {$search: "\"awesome book\""}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book about a young artist!"
}

---------------------Text Indexes Sorting---------------------

// it works in new version automatically
> db.products.find({$text: {$search: "awesome t-shirt"}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
    "title" : "Red T-Shirt",
    "description" : "This T-Shirt is red and it is pretty awesome"
}
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book about a young artist!"
}

// but in previous version
> db.products.find({$text: {$search: "awesome t-shirt"}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book about a young artist!"
}
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
    "title" : "Red T-Shirt",
    "description" : "This T-Shirt is red and it is pretty awesome"
}

// lets add sorting query. in this query check how many words match with each
documents.
// score increase with the number of matching words
// which score is higher comes into first position

> db.products.find({$text: {$search: "awesome t-shirt"}}, {score: {$meta:
'textScore'}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book about a young artist!",
    "score" : 0.6
}
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
    "title" : "Red T-Shirt",
    "description" : "This T-Shirt is red and it is pretty awesome",
    "score" : 1.7999999999999998
}

// if sort does not work automatically add sort function and sort by score.
> db.products.find({$text: {$search: "awesome t-shirt"}}, {score: {$meta:
'textScore'}}).sort({score: {$meta: 'textScore'}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
    "title" : "Red T-Shirt",
    "description" : "This T-Shirt is red and it is pretty awesome",
    "score" : 1.7999999999999998
}
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book about a young artist!",
    "score" : 0.6
}

--------------------------Combining Text Indexes---------------------

> db.products.getIndexes()
> db.products.findOne()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book about a young artist!"
}

// if we now add text indexes with title like that it would be an error.
// already text index is added with description in the document.
// index option conflict
// in every document we can must add only one text index
> db.products.createIndex({title: 'text'})

// can merge the text of multiple fields together into one text index.
// now drop the previous description text index, dropping text index is little bit
different
// have include the text index name
> db.products.dropIndex('description_text')
{ "nIndexesWas" : 2, "ok" : 1 }

// now add two fields like title and description to create combined text index
> db.products.createIndex({title:'text',description: 'text'})
// insert a new element
> db.products.insertOne({title: 'A Ship', description: 'Floats perfectly!'})
// let execute query
> db.products.find({$text: {$search: 'ship'}})
> db.products.find({$text: {$search: 'awesome'}}).pretty()

// search with multiple text
> db.products.find({$text: {$search: 'ship t-shirt'}}).pretty()
> db.products.find({$text: {$search: 'awesome'}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
    "title" : "Red T-Shirt",
    "description" : "This T-Shirt is red and it is pretty awesome"
}
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book about a young artist!"
}

--------------------Using Text Index To Exclude Words------------------

// to exclude words in search just add '-' before word
// here want to search awesome but in the sentence if get awesome then exclude
t-shirt
> db.products.find({$text: {$search: 'awesome -t-shirt'}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a3"),
    "title" : "A book",
    "description" : "This is an awesome book abou a young artist!"
}
```

```
--------------------Setting the Default Language Using Weights-------------------

// first dropping the previous text index
> db.products.dropIndex('title_text_description_text')
// pass some config
> db.products.createIndex({title:'text',description: 'text'},{default_language:
'german', weights: {title: 1, description: 10}})
// can also work without weights, but without weights score value can be changed.
> db.products.createIndex({title:'text',description: 'text'},{default_language:
'english'})

> db.products.find({$text: {$search: '', $language: 'german'}}).pretty()
// caseSensitive default is false
> db.products.find({$text: {$search: '', $caseSensitive: true}}).pretty()
> db.products.createIndex({title:'text',description: 'text'},{default_language:
'english', weights: {title: 1, description: 10}})

// execute query
> db.products.find({$text: {$search: 'red',}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
    "title" : "Red T-Shirt",
    "description" : "This T-Shirt is red and it is pretty awesome"
}

> db.products.find({$text: {$search: 'red'}},{score: {$meta: 'textScore'}}).pretty()
{
    "_id" : ObjectId("5f2adb2fbcaaeedce48e55a4"),
    "title" : "Red T-Shirt",
    "description" : "This T-Shirt is red and it is pretty awesome",
    "score" : 6.666666666666667
}

--------------------- Building Indexes----------------------

1. Foreground:
   a) Collection is locked during index creation.
   b) Faster
2. Background
   a) Collection is accessible during index creation.
   b) Slower

// In the previous, discussed about Foreground index(basically access from core db)
// now create an index that basically a Background index
// first discuss why Background index needs

> use credit
switched to db credit
> show collections
ratings
> db.ratings.find().count()
1000000

> db.ratings.findOne()

// create an index with the age
// here time is important cause documents size 100000
> db.ratings.createIndex({age: 1})

// when creating an index into large scale documents or even a complex documents,
db or documents is locked for a few seconds or couple of minutes
// specially text indexes also need more time
// so this is not an alternative production database

// after creating index if want to insert a new document into a large scale
document then it also take a few lengthy time
> db.ratings.insertOne({person_id: 'a39djd', score: 55.2211, age: 90})
{
    "acknowledged" : true,
    "insertedId" : ObjectId("5f2ced58f48b8c5c77285c65")
}

// let examine the query
> db.ratings.explain('executionStats').find({age: {$gt: 80}})
"executionTimeMillis" : 156
> db.ratings.find({age: {$gt: 80}}).count()
99792

// let drop the index
> db.ratings.dropIndex({age: 1})

// let execute previous query
> db.ratings.explain('executionStats').find({age: {$gt: 80}})
"executionTimeMillis" : 367


// let create a Background index
// in Background index it takes a second argument
// background default is false
// so we have to set background to true
// and it's created immediately
> db.ratings.createIndex({age: 1}, {background: true})

// it happened in the background without locking the collection
```

11-working-with-geospatial-data

```
    ----------------- Working with geo-spatial data--------------

    // Storing Gea-spatial data in Geo-JSON Format
    // Querying Gea-spatial Data

    // GeoJson value is a embedded document, it contains two fields
    1. type --> specifies the GeoJson object type
    2. coordinates --> two values [longitude, latitude] in this format

    // some operations ($near) require such an index.
    // but other operations like $geoWithin does not require index.
    // still can be used to speed up queries

    GeoJson object Type: 1. Point 2. Line String 3. MultiLineString 4. Polygon 5. MultiPolygon 6. MultiPoint 7. GeometryCollection

    // $GeoSpatial Queries : $near, $geoWithin, $geoIntersects
    // GeoSpatial queries work with GeoJson data

    > use awesomeplaces
    switched to db awesomeplaces
    > db.places.insertOne({name:'California Academy of Science',location:{type:'Point',coordinates:[-122.4724356, 37.7672544]}})

    // this is the GeoJson object
    > db.places.findOne()
```

```
{
    "_id" : ObjectId("5f2f7ac7f82aee4b45288303"),
    "name" : "California Academy of Science",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -122.4724356,
            37.7672544
        ]
    }
}

// let execute query into GeoJson data
// to get our current location we have to use webApi or mobile api have to use any
other process so that user's can locate themselves

------------Finding nearest places from current location-------------

// i want to find some places near my current location (sherpur home)
// let's this is my location
latitude : 25.0218715, longitude : 90.0106577

// Sherpur Government University College position --> latitude: 25.017493,longitude
: 90.011495
> db.places.insertOne({name:'Sherpur Government University
College',location:{type:'Point',coordinates:[90.011495,25.017493]}})

// certain radius
// first we have to create GeoSpatial index to track the distance
// here the index name is '2dsphere'
> db.places.createIndex({location: '2dsphere'})

// let find my nearest places
> db.places.find({location: {$near: {$geometry: {type: 'Point', coordinates:
[90.0106577, 25.0218715]}}}}).pretty()
{
    "_id" : ObjectId("5f2f81bef82aee4b45288304"),
    "name" : "Sherpur Government University College",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            90.011495,
            25.017493
        ]
    }
}
{
    "_id" : ObjectId("5f2f7ac7f82aee4b45288303"),
    "name" : "California Academy of Science",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -122.4724356,
            37.7672544
        ]
    }
}

// we can add also max and min distance into the query
> db.places.find({location: {$near: {$geometry: {type: 'Point', coordinates:
[90.0106577, 25.0218715]}, $maxDistance: 30, $minDistance: 10}}}).pretty()
// we don't get any value, cause no places found with the distance that gives

// i calculate the distance between my current location and Sherpur Government
University College is 200.00m to 350m approximate

// so we have to add max distance a little bit large

> db.places.find({location: {$near: {$geometry: {type: 'Point', coordinates:
```

```
[90.0106577, 25.0218715]}, $maxDistance: 500, $minDistance: 20}}}).pretty()
{
    "_id" : ObjectId("5f2f81bef82aee4b45288304"),
    "name" : "Sherpur Government University College",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            90.011495,
            25.017493
        ]
    }
}

--------------Finding Points inside a covered area---------------------

// want to find all coordinates around the area
// could be sphere, any polygon
// which points are inside of the area.
// here consider 4 points
// previous California Academy of Science point also

// insert points
> db.places.insertOne({name:'Conservatory of Flowers',location:{type:'Point',coordinates:[-122.4615748, 37.7701756]}})
> db.places.insertOne({name:'Golden Gate Park Tennis',location:{type:'Point',coordinates:[-122.4593702, 37.7705046]}})
> db.places.insertOne({name:'Nopa',location:{type:'Point',coordinates:[-122.4389058, 37.7747415]}})

// show all documents
> db.places.find().pretty()

// lets draw a polygon with the four points to check the area points exist in database that covered by the area
> const p1 = [-122.4547, 37.77473]
> p1
[ -122.4547, 37.77473 ]
> const p2 = [-122.45303, 37.76641]
> const p3 = [-122.51026, 37.76411]
> const p4 = [-122.51088, 37.77131]

// here do not use $near instead use $geoWithin -->this can help to find all elements within a certain shape, within a certain object, typically something like polygon
// have to also add p1(first corner) again, cause to complete the polygon and also close the polygon

p1 ----------------- p2
   |               |
   |               |
   |               |
p4 ----------------- p3

> db.places.find({location: {$geoWithin: {$geometry: {type: 'Polygon', coordinates: [[p1, p2, p3, p4, p1]]}}}}).pretty()
{
    "_id" : ObjectId("5f2f9194f82aee4b45288305"),
    "name" : "Conservatory of Flowers",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -122.4615748,
            37.7701756
        ]
    }
}
{
    "_id" : ObjectId("5f2f931ff82aee4b45288306"),
    "name" : "Golden Gate Park Tennis",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -122.4593702,
            37.7705046
        ]
    }
}
{
    "_id" : ObjectId("5f2f7ac7f82aee4b45288303"),
    "name" : "California Academy of Science",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -122.4724356,
            37.7672544
        ]
    }
}


-----------------Finding Out if a User is Inside a Specific Area----------------

// Now another typical use case would be the opposite, that have an application
where want to find out whether the user is in a certain area.

// so don't want to find all places in an area but want to store a couple of
different areas potentially in the database

// let's say the neighborhoods of a city and then user sends some coordinates
because he located himself and want to find out in which neighborhood the user is.

// So essentially the same query as before, just the other way around.

// let insert data into new collections
> db.areas.insertOne({name: 'Golden Gate Park' ,area: {type: 'Polygon',
coordinates: [[p1, p2, p3, p4, p1]]}})
> db.areas.find().pretty()

// now create an index
> db.areas.createIndex({area: '2dsphere'})

// basically check $geoIntersects is true or false
// here .p --> means query point
p1 ----------------- p2
   |               |
   |      .p       |
   |               |
p4 ----------------- p3

// Golden Gate Park is in -122.49089, 37.76992
// result --> here can see all points intersect
> db.areas.find({area: {$geoIntersects: {$geometry: {type: 'Point', coordinates:
[-122.49089, 37.76992]}}}}).pretty()


// check with the outside point
// result --> do not find any point
> db.areas.find({area: {$geoIntersects: {$geometry: {type: 'Point', coordinates:
[-122.48446, 37.77776]}}}}).pretty()

----------------Finding Places Within a Certain Radius---------------------

// want to find all elements with unsorted order that are within certain radius
// want to find all places that are within a place or an area
// here have to use geoWithin not geoIntersects
// also can use $centerSphere operator instead of $geometry operator
// $centerSphere is a helpful operator that allows to quickly get a circle ar a point
// so essentially it use a radius and a center and gives the whole circle around that center therefore.
// $centerSphere first element--> the coordinates of the center of the circle want to draw.(-122.46203,37.77286)
// $centerSphere second element --> a radius length with meter(m) , now interested into kilometers
// here use one kilometer
// convert distance to radius, 6378.1 kilometer is a earth radius
// here use places collection
> db.places.find({location: {$geoWithin: {$centerSphere: [[-122.46203, 37.77286], 1 / 6378.1]}}}).pretty()
{
    "_id" : ObjectId("5f2f9194f82aee4b45288305"),
    "name" : "Conservatory of Flowers",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -122.4615748,
            37.7701756
        ]
    }
}
{
    "_id" : ObjectId("5f2f931ff82aee4b45288306"),
    "name" : "Golden Gate Park Tennis",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -122.4593702,
            37.7705046
        ]
    }
}

// California Academy of Science coordinates is falsely inserted, so update the query
> db.places.updateOne({_id: ObjectId("5f2f7ac7f82aee4b45288303")}, {$set: {location: {type: 'Point', coordinates: [-122.46636, 37.77014]}}})
> db.places.find({location: {$geoWithin: {$centerSphere: [[-122.46203, 37.77286], 1 / 6378.1]}}}).pretty()
{
    "_id" : ObjectId("5f2f7ac7f82aee4b45288303"),
    "name" : "California Academy of Science",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -122.46636,
            37.77014
        ]
    }
}
{
    "_id" : ObjectId("5f2f9194f82aee4b45288305"),
    "name" : "Conservatory of Flowers",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -122.4615748,
            37.7701756
        ]
    }
}
{
    "_id" : ObjectId("5f2f931ff82aee4b45288306"),
    "name" : "Golden Gate Park Tennis",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -122.4593702,
            37.7705046
        ]
    }
}

// result--> here get the data with unsorted order. To sort the data apply manuel approach
// $near is the solution of sorted list



---------------------------------------------------------------------------------------------------------------
12-understanding-the-aggregation-framework

---------------------------------------------------------------------------------------------------------------

---------------------Aggregation Framework-----------------------

// Retrieving Data Efficiently and In a Structured way

What is aggregation Framework
// pipeline stages
Steps for find (follow top to down)
                  Collection
                      |            /
                  { $match }      /
                      |          /    Every stage receives
                  { $sort }     /     the output of the
                      |         \     Previous stage
                  { $group }     \
                      |           \
                  { $project }     \
                      |
          Output (List of Documents)

----------------Short Description ---------------

// Stages and Operations
1. There are plenty of available stages and operations can choose from
2. Stages define the different steps of and data is funneled through
3. Each stage receives the output of the last stage as input
4. Operations can be used inside of stages to transform, limit or re-calculated data.

// Important Stages
1. The most important stages are $match, $group, $project, $sort and $unwind etc.
2. Whilst there are some common behaviors between find() filters + projection and $match + $project, the aggregation stages are more flexible.


mongoimport persons.json -d analytics -c persons --jsonArray
> use analytics
> db.persons.findOne()

// The aggregate method takes an array and it takes an array cause have to define a series of steps inside array.

> db.persons.aggregate([
... { $match: {gender: 'female'} }
... ]).pretty()

----------------Understanding the Group Stage--------------------

// group --> group stage allows to group data by a certain fields or by multiple fields
// have to add $ sign before selected query document
// here accumulate by 1 --> increasing value -1 -> decreasing value
// totalPersons is the value that how many person are into same state
// _id is unique value
// can not use group into find() method
// "$location.state" --> means iterating every element

db.persons.aggregate([
    { $match: { gender: 'female' } },
    { $group: { _id: { state: "$location.state" }, totalPersons: { $sum: 1}}}
]).pretty()

// this is group stage in action
// here we get the data with unsorted order
// can also be sorted

{ "_id" : { "state" : "berkshire" }, "totalPersons" : 1 }
{ "_id" : { "state" : "michigan" }, "totalPersons" : 1 }
{ "_id" : { "state" : "county down" }, "totalPersons" : 1 }
{ "_id" : { "state" : "loiret" }, "totalPersons" : 1 }
{ "_id" : { "state" : "cornwall" }, "totalPersons" : 2 }
{ "_id" : { "state" : "sivas" }, "totalPersons" : 1 }
{ "_id" : { "state" : "uşak" }, "totalPersons" : 1 }
{ "_id" : { "state" : "sinop" }, "totalPersons" : 3 }
{ "_id" : { "state" : "marne" }, "totalPersons" : 1 }
{ "_id" : { "state" : "northumberland" }, "totalPersons" : 1 }
{ "_id" : { "state" : "leicestershire" }, "totalPersons" : 1 }
{ "_id" : { "state" : "puy-de-dôme" }, "totalPersons" : 1 }
{ "_id" : { "state" : "maryland" }, "totalPersons" : 1 }
{ "_id" : { "state" : "ardèche" }, "totalPersons" : 1 }
{ "_id" : { "state" : "ankara" }, "totalPersons" : 3 }
{ "_id" : { "state" : "dordogne" }, "totalPersons" : 1 }
{ "_id" : { "state" : "antalya" }, "totalPersons" : 1 }
{ "_id" : { "state" : "corrèze" }, "totalPersons" : 1 }
{ "_id" : { "state" : "ardennes" }, "totalPersons" : 1 }
{ "_id" : { "state" : "bas-rhin" }, "totalPersons" : 2 }
Type "it" for more


// to check aggregation function work correctly
> db.persons.find({'location.state': 'sinop', gender: 'female'}).count()
3

// let also sort the group stage values according to totalPersons when execute query
// sorting done from to previous stage
> db.persons.aggregate([     { $match: { gender: 'female' } },     { $group: { _id:
{ state: "$location.state" }, totalPersons: { $sum: 1 } } },     { $sort: {
totalPersons: -1 } } ]).pretty()
{ "_id" : { "state" : "midtjylland" }, "totalPersons" : 33 }
{ "_id" : { "state" : "nordjylland" }, "totalPersons" : 27 }
{ "_id" : { "state" : "new south wales" }, "totalPersons" : 24 }
{
    "_id" : {
        "state" : "australian capital territory"
    },
    "totalPersons" : 24
}
{ "_id" : { "state" : "syddanmark" }, "totalPersons" : 24 }
{ "_id" : { "state" : "south australia" }, "totalPersons" : 22 }
{ "_id" : { "state" : "hovedstaden" }, "totalPersons" : 21 }
{ "_id" : { "state" : "danmark" }, "totalPersons" : 21 }
{ "_id" : { "state" : "queensland" }, "totalPersons" : 20 }
{ "_id" : { "state" : "overijssel" }, "totalPersons" : 20 }
{ "_id" : { "state" : "sjælland" }, "totalPersons" : 19 }
{ "_id" : { "state" : "nova scotia" }, "totalPersons" : 17 }
{ "_id" : { "state" : "canterbury" }, "totalPersons" : 16 }
{ "_id" : { "state" : "northwest territories" }, "totalPersons" : 16 }
{ "_id" : { "state" : "gelderland" }, "totalPersons" : 16 }
{ "_id" : { "state" : "yukon" }, "totalPersons" : 16 }
{ "_id" : { "state" : "bayern" }, "totalPersons" : 15 }
{ "_id" : { "state" : "northern territory" }, "totalPersons" : 15 }
{ "_id" : { "state" : "tasmania" }, "totalPersons" : 15 }
{ "_id" : { "state" : "noord-brabant" }, "totalPersons" : 14 }
Type "it" for more

// check if answer is correctly
> db.persons.find({'location.state': 'midtjylland', gender: 'female'}).count()
33

----------------- Working with Project Stage ---------------------

// project works in the same way as the projection works in the find method
"gender" : "male",
    "name" : {
        "title" : "mr",
        "first" : "harvey",
        "last" : "chambers"
    },
// full list to all
// want to convert name into one document
// project does not group multiple documents together, its just transform every
single document

> db.persons.aggregate([
    { $project: { _id: 0, gender: 1, fullName: { $concat: ['$name.first', '
    ','$name.last'] } } }
]).pretty()

// now want to first and last name start with Uppercase letter
> db.persons.aggregate([
    {
        $project: {
            _id: 0,
            gender: 1,
            fullName: {
                $concat: [{ $toUpper: '$name.first'}, ' ', { $toUpper: '$name.last'}]
            }
        }
    }
]).pretty()

// $substrCP --> substring part
// 0 -> means starting index
// 1 -> means how much character(length)
> db.persons.aggregate([
    {
        $project: {
            _id: 0,
            gender: 1,
            fullName: {
                $concat: [
                    { $toUpper: { $substrCP: ['$name.first', 0, 1] } },
                    ' ',
                    { $toUpper: { $substrCP: ['$name.last', 0, 1] } }
                ]
            }
        }
    }
]).pretty()

// the final output
> db.persons.aggregate([
    {
        $project: {
            _id: 0,
            gender: 1,
            fullName: {
                $concat: [
                    { $toUpper: { $substrCP: ['$name.first', 0, 1] } },
                    { $substrCP: ['$name.first', 1, { $subtract: [{ $strLenCP:
                    '$name.first' }, 1] }] },
                        ' ',
                        { $toUpper: { $substrCP: ['$name.last', 0, 1] } },
                        { $substrCP: ['$name.last', 1, { $subtract: [{ $strLenCP:
                        '$name.last' }, 1] }] },
                    ]
                }
            }
        }
    ]).pretty()


---------------------------------------------------------------------------------------------
12-understanding-the-aggregation-framework
using-the-aggregation-framework(part-2)
---------------------------------------------------------------------------------------------


    ---------------- Turning the Location Into a geoJSON Object----------------

    // using multiple aggregate function to get the next value from previous

    > db.persons.aggregate([
        {
            $project: {
                _id: 0,
                name: 1,
                email: 1,
                location: {
                    type: 'Point',
                    coordinates: [
                        '$location.coordinates.longitude',
                        '$location.coordinates.latitude',
                    ]
                }
            }
        },
        {
            $project: {
                email: 1,
                location: 1,
                gender: 1,
                fullName: {
                    $concat: [
                        { $toUpper: { $substrCP: ['$name.first', 0, 1] } },
                        { $substrCP: ['$name.first', 1, { $subtract: [{ $strLenCP:
                        '$name.first' }, 1] }] },
                        ' ',
                        { $toUpper: { $substrCP: ['$name.last', 0, 1] } },
                        { $substrCP: ['$name.last', 1, { $subtract: [{ $strLenCP:
                        '$name.last' }, 1] }] },
                    ]
                }
            }
        }
    ]).pretty()

    {
        "location" : {
            "type" : "Point",
            "coordinates" : [
                "168.9462",
                "-22.5329"
            ]
        },
        "email" : "harvey.chambers@example.com",
        "fullName" : "Harvey Chambers"
    }
    // here getting coordinates as a string, so have to convert into number
    > db.persons.aggregate([
        {
            $project: {
                _id: 0,
                name: 1,
                email: 1,
                location: {
                    type: 'Point',
                    coordinates: [
                        {
                            $convert: {
                                input: '$location.coordinates.longitude',
                                to: 'double',
                                onError: 0.0,
                                onNull: 0.0
                            }
                        },
                        {
                            $convert: {
                                input: '$location.coordinates.latitude',
                                to: 'double',
                                onError: 0.0,
                                onNull: 0.0
                            }
                        }
                    ]
                }
            }
        },
        {
            $project: {
                email: 1,
                location: 1,
                gender: 1,
                fullName: {
                    $concat: [
                        {
                            $toUpper: {
                                $substrCP: ['$name.first', 0, 1]
                            }
                        }, {
                            $substrCP: [
                                '$name.first', 1, {
                                    $subtract: [
                                        { $strLenCP: '$name.first' }, 1
                                    ]
                                }]
                        },
                        ' ',
                        {
                            $toUpper: {
                                $substrCP: ['$name.last', 0, 1]
                            }
                        },
                        {
                            $substrCP: [
                                '$name.last', 1, {
                                    $subtract: [
                                        { $strLenCP: '$name.last' }, 1
                                    ]
                                }]
                        }
                    ]
                }
            }
        }
```

```
]).pretty()

// transforming the BirthDate into data format
db.persons.aggregate([
    {
        $project: {
            _id: 0,
            name: 1,
            email: 1,
            birthdate: {
                $convert: {
                    input: '$dob.date',
                    to: 'date',
                    onError: 0.0,
                    onNull: 0.0
                }
            },
            age: '$dob.age',
            location: {
                type: 'Point',
                coordinates: [
                    {
                        $convert: {
                            input: '$location.coordinates.longitude',
                            to: 'double',
                            onError: 0.0,
                            onNull: 0.0
                        }
                    },
                    {
                        $convert: {
                            input: '$location.coordinates.latitude',
                            to: 'double',
                            onError: 0.0,
                            onNull: 0.0
                        }
                    }
                ]
            }
        }
    },
    {
        $project: {
            email: 1,
            location: 1,
            gender: 1,
            birthdate: 1,
            age: 1,
            fullName: {
                $concat: [
                    {
                        $toUpper: {
                            $substrCP: ['$name.first', 0, 1]
                        }
                    }, {
                        $substrCP: [
                            '$name.first', 1, {
                                $subtract: [
                                    { $strLenCP: '$name.first' }, 1
                                ]
                            }]
                    },
                    ' ',
                    {
                        $toUpper: {
                            $substrCP: ['$name.last', 0, 1]
                        }
                    },
                    {
                        $substrCP: [
                            '$name.last', 1, {
                                $subtract: [
                                    { $strLenCP: '$name.last' }, 1
                                ]
                            }]
                    }
                ]
            }
        }
    }
]).pretty()

// Using Shortcuts for Transformations
// But Shortcuts Transformations always not good. cause can not handle error in
this process

db.persons.aggregate([
    {
        $project: {
            _id: 0,
            name: 1,
            email: 1,
            birthdate: { $toDate: '$dob.date'},
            age: '$dob.age',
            location: {
                type: 'Point',
                coordinates: [
                    {
                        $convert: {
                            input: '$location.coordinates.longitude',
                            to: 'double',
                            onError: 0.0,
                            onNull: 0.0
                        }
                    },
                    {
                        $convert: {
                            input: '$location.coordinates.latitude',
                            to: 'double',
                            onError: 0.0,
                            onNull: 0.0
                        }
                    }
                ]
            }
        }
    },
    {
        $project: {
            email: 1,
            location: 1,
            gender: 1,
            birthdate: 1,
            age: 1,
            fullName: {
                $concat: [
                    {
                        $toUpper: {
                            $substrCP: ['$name.first', 0, 1]
                        }
                    }, {
                        $substrCP: [
                            '$name.first', 1, {
                                $subtract: [
                                    { $strLenCP: '$name.first' }, 1
                                ]
                            }]
                    },
                        ' ',
                        {
                            $toUpper: {
                                $substrCP: ['$name.last', 0, 1]
                            }
                        },
                        {
                            $substrCP: [
                                '$name.last', 1, {
                                    $subtract: [
                                    { $strLenCP: '$name.last' }, 1
                                    ]
                                }]
                        }
                    ]
                }
            }
        }
    ]).pretty()


-------------------------------------------------------------------------------------------------
12-understanding-the-aggregation-framework
using-the-aggregation-framework(part-3)
-------------------------------------------------------------------------------------------------


    --------------------Understanding the ISO Week Year Operator-----------------

    // $isoWeekYear retries the year out of date
    db.persons.aggregate([
        {
            $project: {
                _id: 0,
                name: 1,
                email: 1,
                birthdate: { $toDate: '$dob.date' },
                age: '$dob.age',
                location: {
                    type: 'Point',
                    coordinates: [
                        {
                            $convert: {
                                input: '$location.coordinates.longitude',
                                to: 'double',
                                onError: 0.0,
                                onNull: 0.0
                            }
                        },
                        {
                            $convert: {
                                input: '$location.coordinates.latitude',
                                to: 'double',
                                onError: 0.0,
                                onNull: 0.0
                            }
                        }
                    ]
                }
            }
        },
        {
            $project: {
                email: 1,
                location: 1,
                gender: 1,
                birthdate: 1,
                age: 1,
                fullName: {
                    $concat: [
                        {
                            $toUpper: {
                                $substrCP: ['$name.first', 0, 1]
                            }
                        }, {
                            $substrCP: [
                                '$name.first', 1, {
                                    $subtract: [
                                        { $strLenCP: '$name.first' }, 1
                                    ]
                                }]
                        },
                        ' ',
                        {
                            $toUpper: {
                                $substrCP: ['$name.last', 0, 1]
                            }
                        },
                        {
                            $substrCP: [
                                '$name.last', 1, {
                                    $subtract: [
                                        { $strLenCP: '$name.last' }, 1
                                    ]
                                }]
                        }
                    ]
                }
            }
        },
        { $group: { _id: { birthYear: { $isoWeekYear: '$birthdate' } }, numPersons: {
        $sum: 1 } } }
]).pretty()

// adding sort

db.persons.aggregate([
    {
        $project: {
            _id: 0,
            name: 1,
            email: 1,
            birthdate: { $toDate: '$dob.date' },
            age: '$dob.age',
            location: {
                type: 'Point',
                coordinates: [
                    {
                        $convert: {
                            input: '$location.coordinates.longitude',
                            to: 'double',
                            onError: 0.0,
                            onNull: 0.0
                        }
                    },
                    {
                        $convert: {
                            input: '$location.coordinates.latitude',
                            to: 'double',
                            onError: 0.0,
                            onNull: 0.0
                        }
                    }
                ]
            }
        }
    },
    {
        $project: {
            email: 1,
            location: 1,
            gender: 1,
            birthdate: 1,
            age: 1,
            fullName: {
                $concat: [
                    {
                        $toUpper: {
                            $substrCP: ['$name.first', 0, 1]
                        }
                    }, {
                        $substrCP: [
                            '$name.first', 1, {
                                $subtract: [
                                    { $strLenCP: '$name.first' }, 1
                                ]
                            }]
                    },
                    ' ',
                    {
                        $toUpper: {
                            $substrCP: ['$name.last', 0, 1]
                        }
                    },
                    {
                        $substrCP: [
                            '$name.last', 1, {
                                $subtract: [
                                    { $strLenCP: '$name.last' }, 1
                                ]
                            }]
                    }
                ]
            }
        }
    },
    { $group: { _id: { birthYear: { $isoWeekYear: '$birthdate' } }, numPersons: { $sum: 1 } } },
    { $sort: { numPersons: -1}}
]).pretty()
```

------------------$group vs $project---------------------

```
$group :
   1. grouping multiple documents into one document.
   2. n:1
   3. have multiple documents and return one grouped by one or more categories.
   4. do things like summing, counting, averaging, build array and so on

$project:
   1. get one document and then will return one document, that one document we'll just have changed.
   2. 1:1
   3. transform a single document, add new fields and so on.
   4. have a one to one relation, include/exclude fields.
```

-------------Pushing Elements Into Newly Created Arrays-------------------

// push Operator allows to push a new element into the all hobbies array for every incoming document.

```
> db.friends.aggregate([
...     { $group: { _id: { age: '$age' }, allHobbies: {$push: '$hobbies'}}}
... ]).pretty()
{
        "_id" : {
            "age" : 29
        },
        "allHobbies" : [
            [
                "Sports",
                "Cooking"
            ],
            [
                "Cooking",
                "Skiing"
            ]
        ]
    }
    {
        "_id" : {
            "age" : 30
        },
        "allHobbies" : [
            [
                "Eating",
                "Data Analytics"
            ]
        ]
    }

    ---------------Understanding the unwind Stage-------------------

    // do not want to insert into  nested array
    // The unwind stage is always a great stage when have an array of which want to
    pull out the elements.

    > db.friends.aggregate([
        { $unwind:'$hobbies' }
    ]).pretty()
    // result -> every array element has one document

    // now adding group to every document according to age
    > db.friends.aggregate([
    ...     { $unwind: '$hobbies' },
    ...     { $group: { _id: { age: '$age' }, allHobbies: { $push: '$hobbies' } } }
    ... ]).pretty()
    {
        "_id" : {
            "age" : 29
        },
        "allHobbies" : [
            "Sports",
            "Cooking",
            "Cooking",
            "Skiing"
        ]
    }
    {
        "_id" : {
            "age" : 30
        },
        "allHobbies" : [
            "Eating",
            "Data Analytics"
        ]
    }




------------------------------------------------------------------------------------------
----------------
12-understanding-the-aggregation-framework
```

```
using-the-aggregation-framework(part-4)
-------------------------------------------------------------------------------------------------


    ---------------------Eliminating Duplicate Values-----------------

    // but can see that have some duplicate values
    // instead of $push have to use $addToSet
    // $addToSet pushes but avoid duplicating element.

    db.friends.aggregate([
        {
            $unwind: '$hobbies'
        },
        {
            $group: {
                _id: {
                    age: '$age'
                },
                allHobbies: {
                    $addToSet: '$hobbies'
                }
            }
        }
    ]).pretty()
    {
        "_id" : {
            "age" : 29
        },
        "allHobbies" : [
            "Skiing",
            "Sports",
            "Cooking"
        ]
    }
    {
        "_id" : {
            "age" : 30
        },
        "allHobbies" : [
            "Eating",
            "Data Analytics"
        ]
    }

    --------------Using projection with Arrays---------------------

    // want to print first document from examScores arrays

    // 1 --> means first element from first and length 1
    > db.friends.aggregate([
        {
            $project: {
                _id: 0,
                examScore: {
                    $slice: ['$examScores', 1]
                }
            }
        }
    ]).pretty()
    { "examScore" : [ { "difficulty" : 4, "score" : 57.9 } ] }
    { "examScore" : [ { "difficulty" : 7, "score" : 52.1 } ] }
    { "examScore" : [ { "difficulty" : 3, "score" : 75.1 } ] }

    > db.friends.aggregate([
        {
            $project: {
                _id: 0,
                examScore: {
                    $slice: ['$examScores', 2]
                }
            }
        }
]).pretty()

// -1 means last

> db.friends.aggregate([
        {
            $project: {
                _id: 0,
                examScore: {
                    $slice: ['$examScores', -1]
                }
            }
        }
]).pretty()
{ "examScore" : [ { "difficulty" : 3, "score" : 88.5 } ] }
{ "examScore" : [ { "difficulty" : 5, "score" : 53.1 } ] }
{ "examScore" : [ { "difficulty" : 6, "score" : 61.5 } ] }

// last two scores
> db.friends.aggregate([
        {
            $project: {
                _id: 0,
                examScore: {
                    $slice: ['$examScores', -2]
                }
            }
        }
]).pretty()

// start at position two and give one element
> db.friends.aggregate([
        {
            $project: {
                _id: 0,
                examScore: {
                    $slice: ['$examScores', 2, 1]
                }
            }
        }
]).pretty()

-------------------Getting the length of and array--------------------

// $size calculate the length of an array

> db.friends.aggregate([
        {
            $project: {
                _id: 0,
                numScores: {
                    $size: '$examScores'
                }
            }
        }
]).pretty()



---------------------Using the Filter Operator-----------------

// $filter Operator allows to filter out certain elements an array and only return
the data according to condition
// filter score so the greater than 60
// here sc is a temporary variable for using condition
// sc is a temporary variable of examScores but filter function executes over and
over again all fields
// so have to use two dollar sign
// $cond --> condition

db.friends.aggregate([
    {
        $project: {
            _id: 0,
            scores: {
                $filter: {
                    input: '$examScores',
                    as: 'sc',
                    cond: {
                        $gt: ['$$sc.score', 60]
                    }
                }
            }
        }
    }
]).pretty()
{
    "scores" : [
        {
            "difficulty" : 6,
            "score" : 62.1
        },
        {
            "difficulty" : 3,
            "score" : 88.5
        }
    ]
}
{ "scores" : [ { "difficulty" : 2, "score" : 74.3 } ] }
{
    "scores" : [
        {
            "difficulty" : 3,
            "score" : 75.1
        },
        {
            "difficulty" : 6,
            "score" : 61.5
        }
    ]


-------------------Applying Multiple Operations to our Array---------------

wanted to transform our friend objects such that only output the highest exam score

> db.friends.aggregate([
    { $unwind: '$examScores' },
    { $sort: { 'examScores.score': -1 } }
]).pretty()

// can do same thing by projection

> db.friends.aggregate([
    { $unwind: '$examScores' },
    { $project: { _id: 1, name: 1, age: 1, score: '$examScores.score'}},
    { $sort: { score: -1 } },
    { $group: { _id: '$_id', maxScore: {$max: '$score'}}}
]).pretty()

// group by id but can also add anything
// if can group by name, it is bad choice. cause name can be duplicate
> db.friends.aggregate([
        { $unwind: '$examScores' },
        { $project: { _id: 1, name: 1, age: 1, score: '$examScores.score' } },
        { $group: { _id: '$_id', maxScore: { $max: '$score' } } }
    ]).pretty()

    { "_id" : ObjectId("5f318bb939e723820551436e"), "maxScore" : 74.3 }
    { "_id" : ObjectId("5f318bb939e723820551436f"), "maxScore" : 75.1 }
    { "_id" : ObjectId("5f318bb939e723820551436d"), "maxScore" : 88.5 }

    // show the name and sort with descending order
    // use the first value encounter

    // $first -->means want to get the name value
    > db.friends.aggregate([
        { $unwind: '$examScores' },
        { $project: { _id: 1, name: 1, age: 1, score: '$examScores.score' } },
        { $group: { _id: '$_id', name: { $first: '$name' }, maxScore: { $max: '$score'
        } } },
        { $sort: { maxScore: -1 } }
    ]).pretty()


    {
        "_id" : ObjectId("5f318bb939e723820551436d"),
        "name" : "Max",
        "maxScore" : 88.5
    }
    {
        "_id" : ObjectId("5f318bb939e723820551436f"),
        "name" : "Maria",
        "maxScore" : 75.1
    }
    {
        "_id" : ObjectId("5f318bb939e723820551436e"),
        "name" : "Manu",
        "maxScore" : 74.3
    }

    > db.friends.aggregate([
        { $unwind: '$examScores' },
        { $project: { _id: 1, name: 1, age: 1, score: '$examScores.score' } },
        { $group: { _id: '$_id', name: { $first: '$name' }, age: { $first: '$age' },
        maxScore: { $max: '$score' } } },
        { $sort: { maxScore: -1 } }
    ]).pretty()




-------------------------------------------------------------------------------------------------
-----------------
12-understanding-the-aggregation-framework
using-the-aggregation-framework(part-5)
-------------------------------------------------------------------------------------------------
-----------------


    --------------------Understanding bucket-------------------

    // let's prepare a bucket stage
    // using bucket can create a different categories and filter

    // boundaries means range/levels like 0-18, 18-30,30-50, 50-80, 80-120
    // in every range first value execute not last value, 18-30 --> means with 18 but
    not exist 30
    > db.persons.aggregate([
        {
            $bucket: {
                groupBy: '$dob.age',
                boundaries: [0, 18, 30, 50, 80, 120],
                output: {
                    numPersons: { $sum: 1 },
                    average: { $avg: '$dob.age' },
                }
            }
        }
    ]).pretty()

    // here we get the three bucket
    { "_id" : 18, "numPersons" : 868, "average" : 25.101382488479263 }
    { "_id" : 30, "numPersons" : 1828, "average" : 39.4917943107221 }
    { "_id" : 50, "numPersons" : 2304, "average" : 61.46440972222222 }

    > db.persons.find({'dob.age': {$gt: 17, $lt: 30}}).count()
    868

    > db.persons.find({'dob.age': {$gt: 49, $lt: 80}}).count()
    2304

    > db.persons.find({'dob.age': {$gt: 29, $lt: 50}}).count()
    1828


    // checking the validation
    // no data
    > db.persons.find({'dob.age': {$lt: 18}})
    > db.persons.find({'dob.age': {$gt: 80}})
    > db.persons.find({'dob.age': 80})

    // adding more levels

    > db.persons.aggregate([
        {
            $bucket: {
                groupBy: '$dob.age',
                boundaries: [18, 30, 40, 50, 60, 120],
                output: {
                    numPersons: { $sum: 1 },
                    average: { $avg: '$dob.age' },
                }
            }
        }
    ]).pretty()

    // can also create a auto bucket by defining how many buckets want
    // almost have equal distributions

    > db.persons.aggregate([
        {
            $bucketAuto: {
                groupBy: '$dob.age',
                buckets: 5,
                output: {
                    numPersons: { $sum: 1 },
                    average: { $avg: '$dob.age' },
                }
            }
        }
    ]).pretty()

    {
        "_id" : {
            "min" : 21,
            "max" : 32
        },
        "numPersons" : 1042,
        "average" : 25.99616122840691
    }
```

```
{
    "_id" : {
        "min" : 32,
        "max" : 43
    },
    "numPersons" : 1010,
    "average" : 36.97722772277228
}
{
    "_id" : {
        "min" : 43,
        "max" : 54
    },
    "numPersons" : 1033,
    "average" : 47.98838334946757
}
{
    "_id" : {
        "min" : 54,
        "max" : 65
    },
    "numPersons" : 1064,
    "average" : 58.99342105263158
}
{
    "_id" : {
        "min" : 65,
        "max" : 74
    },
    "numPersons" : 851,
    "average" : 69.11515863689776
}

------------------Diving into Additional Stages------------------

// want to find the 10 users, the 10 persons with the oldest birth date, so the
lowest birth date

> db.persons.aggregate([
    {
        $project: {
            _id: 0,
            name: 1,
            birthDate: {
                $toDate: '$dob.date'
            }
        }
    }
]).pretty()

> db.persons.aggregate([
    { $project: { _id: 0, name: 1, birthDate: { $toDate: '$dob.date' } } },
    { $sort: { birthDate: 1 } },
    { $limit: 10 }
]).pretty()

// adding some extra
> db.persons.aggregate([
    { $project: { _id: 0, name: {$concat:['$name.first', ' ', '$name.last']},
    birthDate: { $toDate: '$dob.date' } } },
    { $sort: { birthDate: 1 } },
    { $limit: 10 }
]).pretty()

// skip first 10
> db.persons.aggregate([
    { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
    birthDate: { $toDate: '$dob.date' } } },
    { $sort: { birthDate: 1 } },
    { $skip: 10},
    { $limit: 10 }
]).pretty()

// but after $skip into $limit it does not work
> db.persons.aggregate([
    { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
    birthDate: { $toDate: '$dob.date' } } },
    { $sort: { birthDate: 1 } },
    { $limit: 10 },
    { $skip: 10 }
]).pretty()

// if add sort into last can see the different result
> db.persons.aggregate([
    { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
    birthDate: { $toDate: '$dob.date' } } },
    { $limit: 10 },
    { $skip: 10 },
    { $sort: { birthDate: 1 } },
]).pretty()

// same also for $match
> db.persons.aggregate([
    { $match: { gender: 'male' } },
    { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
    birthDate: { $toDate: '$dob.date' } } },
    { $skip: 10 },
    { $limit: 10 },
    { $sort: { birthDate: 1 } }
]).pretty()

// if $match add after the project without projection,we do not get any result
> db.persons.aggregate([
    { $project: { _id: 0, name: { $concat: ['$name.first', ' ', '$name.last'] },
    birthDate: { $toDate: '$dob.date' } } },
    { $sort: { birthDate: 1 } },
    { $match: { gender: 'male' } },
    { $skip: 10 },
    { $limit: 10 },
]).pretty()

// if gender add into projection phase then will get results
> db.persons.aggregate([
    { $project: { _id: 0, gender: 1, name: { $concat: ['$name.first', ' ',
    '$name.last'] }, birthDate: { $toDate: '$dob.date' } } },
    { $sort: { birthDate: 1 } },
    { $match: { gender: 'male' } },
    { $skip: 10 },
    { $limit: 10 },
]).pretty()

// but best is, use $match before $project
> db.persons.aggregate([
    { $match: { gender: 'male' } },
    { $project: { _id: 0, gender: 1, name: { $concat: ['$name.first', ' ',
    '$name.last'] }, birthDate: { $toDate: '$dob.date' } } },
    { $sort: { birthDate: 1 } },
    { $skip: 10 },
    { $limit: 10 },
]).pretty()

-----------------Writing Pipeline Results Into a New Collection-----------

// by getting the output we can store into the another Collection
// can do work with the out stage
db.persons.aggregate([
  {
    $project: {
      _id: 0,
      name: 1,
      email: 1,
      birthdate: { $toDate: '$dob.date' },
      age: "$dob.age",
      location: {
        type: 'Point',
        coordinates: [
          {
            $convert: {
              input: '$location.coordinates.longitude',
              to: 'double',
              onError: 0.0,
              onNull: 0.0
            }
          },
          {
            $convert: {
              input: '$location.coordinates.latitude',
              to: 'double',
              onError: 0.0,
              onNull: 0.0
            }
          }
        ]
      }
    }
  },
  {
    $project: {
      gender: 1,
      email: 1,
      location: 1,
      birthdate: 1,
      age: 1,
      fullName: {
        $concat: [
          { $toUpper: { $substrCP: ['$name.first', 0, 1] } },
          {
            $substrCP: [
              '$name.first',
              1,
              { $subtract: [{ $strLenCP: '$name.first' }, 1] }
            ]
          },
          ' ',
          { $toUpper: { $substrCP: ['$name.last', 0, 1] } },
          {
            $substrCP: [
              '$name.last',
              1,
              { $subtract: [{ $strLenCP: '$name.last' }, 1] }
            ]
          }
        ]
      }
    }
  },
  { $out: "transformedPersons" }
]).pretty();

-----------------Working with the geoNear Stage------------------

// first create an index into the transformedPersons Collection

> db.transformedPersons.createIndex({location: '2dsphere'})
// create geo location aggregation pipeline stages

// have to specify and that is the distance field, because geoNear will actually
also give us back the distance that is calculated between our point and the document

// geoNear, it has to be the first element in the pipeline because it needs to use
that index and the first pipeline element is the only element with direct access to
the collection, other pipeline stages just get the output of the previous pipeline
stage, this is the only element with direct access to the collection.

// also can add query
> db.transformedPersons.aggregate([
    {
        $geoNear: {
            near: {
                type: 'Point',
                coordinates: [-18.4, -42.8]
            },
            maxDistance: 1000000,
            $limit: 10,
            query: { age: { $gt: 30 } },
            distanceField: 'distance'
        }
    }
]).pretty()

// can also add multiple pipeline stages
db.transformedPersons.aggregate([
    {
        $geoNear: {
            near: {
                type: 'Point',
                coordinates: [-18.4, -42.8]
            },
            maxDistance: 1000000,
            $limit: 10,
            query: { age: { $gt: 30 } },
            distanceField: 'distance'
        }
    },
    { $project: { _id: 1, email: 0, birthdate: 0 } },
    { $sort: { distanceField: 1 } },
]).pretty()



---------------------------------------------------------------------------------------------
13-working-with-numeric-data
---------------------------------------------------------------------------------------------

-------------------------Working With Numeric Data--------------------

// numeric data is most important in scientific calculation
// Number more complex than any other
// 3 types of number in mongoDB.(Integers,Longs,Doubles)

Integers(int32) -->
 1. Only full Numbers(+- 2^32).
 2. Use for 'normal' Integers

Longs(int64) -->
 1. Only full Numbers(+- 2^64).
 2. Use for large Integers

Doubles(64bit) -->
 1. Numbers with Decimal Places(Decimal values are approximated). 2. Use for floats
 where high precision is not required

High Precision Doubles(128bit) -->
 1. Numbers with Decimal Places(Decimal values are stored with high precision(34
 decimal digits)).
 2. Use for floats where high precision is required


// in mongoDB driver is a javaScript based driver.
// all numeric values stored as a double

> use numeric
switched to db numeric
> db.persons.insertOne({name: 'Max', age: 29})
> db.persons().find()

----------------WOrking with Int32--------------------

// here can see the size
> db.persons.stats()
"size" : 49,
"count" : 1,
> db.persons.deleteMany({})

// here can see that size is decrease
> db.persons.insertOne({ age: 29})
> db.persons.stats()
"size" : 35,
"count" : 1,
> db.persons.deleteMany({})

// here also can see that size now also more decrease
> db.persons.insertOne({ age: NumberInt(29)})
> db.persons.insertOne({ age: NumberInt("29")})
"size" : 31,
"count" : 1,

-----------------Working with Int64--------------------

// here can the output is a wrong value
> db.companies.insertOne({valuation: NumberInt('50000000000')})
> db.companies.findOne()
{ "_id" : ObjectId("5f3e81e7d0209e4d3a0ec072"), "valuation" : -1539607552

// if decrease one value then it works
> db.companies.insertOne({valuation: NumberInt('5000000000')})

// but if the data stored as a 64 bit double then it works
> db.companies.insertOne({valuation: 50000000000'})

> db.companies.find()
{ "_id" : ObjectId("5f3e81e7d0209e4d3a0ec072"), "valuation" : -1539607552 }
{ "_id" : ObjectId("5f3e8635d0209e4d3a0ec073"), "valuation" : 50000000000 }
{ "_id" : ObjectId("5f3e866cd0209e4d3a0ec074"), "valuation" : 705032704 }

// to store biggest possible number as Integers
// always have to use quotation marks
// basically it works as a string
> db.companies.insertOne({valuation: NumberLong('50000000000')})
> db.companies.find()
{ "_id" : ObjectId("5f3e81e7d0209e4d3a0ec072"), "valuation" : -1539607552 }
{ "_id" : ObjectId("5f3e8635d0209e4d3a0ec073"), "valuation" : 50000000000 }
{ "_id" : ObjectId("5f3e866cd0209e4d3a0ec074"), "valuation" : 705032704 }
{ "_id" : ObjectId("5f3e86ecd0209e4d3a0ec075"), "valuation" :
NumberLong("50000000000") }

----------------Doing Maths with Floats Int32 Int64--------------

> db.accounts.insertOne(name: 'Max', amount: '342342534583735345745245 24')
// add a small number
> db.accounts.insertOne(name: 'Max', amount: '10')

// $inc or any math calculation does not work with string value
```

```
> db.accounts.updateOne({}, {$inc: {amount: 10}})

// have to insert a Integers value
> db.accounts.deleteMany()
> db.accounts.insertOne(name: 'Max', amount: NumberInt('10'))

// here 10 as a double value mongoDB convert the sum as double
> db.accounts.updateOne({}, {$inc: {amount: 10}})

// if update the number with wrapping with NumberInt then the final output be a int
> db.accounts.updateOne({}, {$inc: {amount: NumberInt('10')}})

// let delete the document

> db.companies.deleteMany({})

// insert a large number
> db.companies.insertOne({valuation: NumberLong('3423425345837353457452452 4')})

// to calculate math operation with the large number NumberLong should be include
in that number
// this is incorrect
> db.companies.updateOne({}, {$inc: {valuation: 1}})
// this is correct
> db.companies.updateOne({}, {$inc: {valuation: NumberLong('1')}})

----------------What's Wrong With Normal Doubles------------------

> db.science.insertOne({a: 0.3, b: 0.1})

> db.science.findOne()

// let execute maths calculation
> db.science.aggregate([{$project: {result: {$subtract: ['$a', '$b']}}}])

// here should be the subtract value is 0.2
// but it's come
{ "_id" : ObjectId("5f3e9ec9d0209e4d3a0ec079"), "result" : 0.19999999999999998 }

// so have to fix the issue

-------------------Working With Decimal 128bit----------------------

// to get the exact subtract value, have to use NumberDecimal constructor

> db.companies.deleteMany({})
> db.science.insertOne({a: NumberDecimal("0.3"), b: NumberDecimal("0.1")})

> db.science.find().pretty()
{
    "_id" : ObjectId("5f3ea3ced0209e4d3a0ec07b"),
    "a" : NumberDecimal("0.3"),
    "b" : NumberDecimal("0.1")
}

// now getting the exact value
> db.science.aggregate([{$project: {result: {$subtract: ['$a', '$b']}}}])
{ "_id" : ObjectId("5f3ea3ced0209e4d3a0ec07b"), "result" : NumberDecimal("0.2") }

// let execute another query
> db.science.updateOne({}, {$inc: {a: 0.1}})

> db.science.updateOne({}, {$inc: {a: 0.1}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.science.find().pretty()
{
    "_id" : ObjectId("5f3ea3ced0209e4d3a0ec07b"),
    "a" : NumberDecimal("0.400000000000000"),
    "b" : NumberDecimal("0.1")
}

// so to get the right value
> db.science.updateOne({}, {$inc: {a: NumberDecimal("0.1")}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.science.find().pretty()
{
    "_id" : ObjectId("5f3ea3ced0209e4d3a0ec07b"),
    "a" : NumberDecimal("0.500000000000000"),
    "b" : NumberDecimal("0.1")
}

// NumberDecimal means getting high precision decimal

> db.number.insertOne({num: 0.1})
> db.number.stats()
"size" : 33

> db.number.deleteMany({})

> db.number.insertOne({num: NumberDecimal("0.1")})
> db.number.stats()
"size" : 41



-------------------------------------------------------------------------------------------------------------
16-transactions
-------------------------------------------------------------------------------------------------------------

------------------Transactions----------------------

                User deletes Accounts
    Users Collection              Posts Collection
--------------------------------------------------------
|   { User Document } -----------> { Post Document }   |
|                      \                               |
|             related  \                               |
|                        \--------> { Post Document }   |
|  Should be deleted together                          |
---------------------------------------------------------

// always have to change ip address into mongoDB cloud cluster

// first have access the mongoDB cloud
mongo "mongodb+srv://mytestingcluster.n7v1t.mongodb.net/<test1>" --username 
mijanur

> use blog
> db.users.insertOne({name: 'Max'})
> db.posts.insertMany([{title: 'A js post', views: 23, userId: 
ObjectId("5f4163d6526c4846e4c6fe1b")}, {title: 'Group discussion', views: 2, 
userId: ObjectId("5f4163d6526c4846e4c6fe1b")}])

// have to execute the mongo session to work with the Transactions

> const session = db.getMongo().startSession()
> session.startTransaction()

> const usersCol = session.getDatabase('blog').users
> const postsCol = session.getDatabase('blog').posts

// this is basically remove from session
> usersCol.deleteOne({_id: ObjectId("5f4163d6526c4846e4c6fe1b")})

// this command also successfully execute(this comes from cloud not session), but 
we deleted the user before
> postsCol.deleteMany({userId: ObjectId("5f4163d6526c4846e4c6fe1b")})

MongoDB Enterprise atlas-9fuf07-shard-0:PRIMARY> usersCol.deleteOne({_id:
ObjectId("5f4163d6526c4846e4c6fe1b")})
{ "acknowledged" : true, "deletedCount" : 1 }

// it basically deleted from cache but not from real server
> db.users.find().pretty()
{ "_id" : ObjectId("5f4163d6526c4846e4c6fe1b"), "name" : "Max" }

// to execute fully delete from cloud have to commit Transactions
> session.commitTransaction()

// now deleted from cloud
> db.users.find().pretty()

// can also abort --> all things are trying to fully delete

// so these actions either succeed together or they fail together. That is the idea
behind the transactions.

// this is basically comes from atomicity
// get an atomicity in operation level not just on a document level
// so need cross operation consistency


------------------------------------------------------------------------------------------------------------

------------------------------------------------------------------------------------------------------------
```