MongoDB

All of us do not have equal talent. But, all of us have an equal opportunity to develop our talents.

A.P.J. Abdul Kalam

- `sudo apt install build-essential`

Database Technologies – MongoDB

```
Enterprise primaryDB> config.set("editor", "notepad++")
Enterprise primaryDB> config.set("editor", null)
```

Session 1

Big Data?

Big data is a term that describes the large volume of data – both structured and unstructured.

What is Big Data?

Big Data is also data but with a huge size. Big Data is a term used to describe a collection of data that is huge in size and yet growing with time. In short such data is so large and complex that none of the traditional data management tools are able to store it or process it efficiently.

Characteristics Of Big Data

Big data is often characterized by the 3Vs: the extreme ***VOLUME*** of data, the wide ***VARIETY*** of data and the ***VELOCITY*** at which the data must be processed.

NoSQL, which stands for "Not Only SQL" which is an alternative to traditional relational databases in which data is placed in tables and data schema is carefully designed before the database is built.

NoSQL

NoSQL database are primarily called as **non-SQL** or **non-relational** database. MongoDB is Scalable (able to be changed in size or scale), open-source, high-perform, document-oriented database.

Remember:

- **Horizontal scaling** means that you **scale** by adding more machines into your pool of resources.
- **Vertical scaling** means that you **scale** by adding more power (**CPU**, **RAM**) to an existing machine.

why NoSQL

Next Generation Databases

Not Only SQL

Non – Relational

Distributed Architecture

Open Source

Horizontally Scalable

when to use NoSQL

When should NoSQL be used:

- When huge amount of data need to be stored and retrieved .
- The relationship between the data you store is not that important
- The data changing over time and is not structured.
- Support of Constraints and Joins is not required at database level.
- The data is growing continuously and you need to scale the database regular to handle the data.

Remember:

- Data Persistence on Server-Side via NoSQL.
- Does not use SQL-like query language.
- Longer persistence
- Store massive amounts of data.
- Systems can be scaled.
- High availability.
- Semi-structured data.
- Support for numerous concurrent connections.
- Indexing of records for faster retrieval

NoSQL Categories

NoSQL Categories

There are 4 basic types of NoSQL databases.

Key-value stores	Key-value stores, or key-value databases, implement a simple data model that pairs a unique key with an associated value. e.g. • **Redis**
Column-oriented	Wide-column stores organize data tables as columns instead of as rows. e.g. • **hBase, Cassandra**
Document oriented	Document databases, also called document stores, store semi-structured data and descriptions of that data in document format. e.g. • **MongoDB, CouchDB**
Graph	Graph data stores organize data as nodes. e.g. • **Neo4j**

NoSQL Categories

Column-oriented

Graph

An object is an unordered set of name/value pairs.

An array is an ordered collection of values.

SQL vs NoSQL Database

Relational databases are commonly referred to as SQL databases because they use SQL (structured query language) as a way of storing and querying the data.

Difference:

- NoSQL databases are document based, key-value pairs, or wide-column stores. This means that SQL databases represent data in form of tables which consists of *n* number of rows of data whereas NoSQL databases are the collection of key-value pair, documents, or wide-column stores which do not have standard schema definitions.
- SQL databases have predefined schema whereas NoSQL databases have dynamic schema for unstructured data.
- SQL requires a predefined schema, meaning the structure of the data (tables, columns, data types) needs to be defined before data can be inserted whereas NoSQL typically has a dynamic or schema-less approach, allowing for flexibility in the data structure. New fields can be added without requiring a predefined schema.
- SQL databases are vertically scalable whereas the NoSQL databases are horizontally scalable.
- SQL databases uses SQL (structured query language) for defining and manipulating the data. In NoSQL database, queries are focused on collection of documents.

Types of Data

Structured

0.103	0.176	0.387	0.300	0.379
0.333	0.384	0.564	0.587	0.857
0.421	0.309	0.654	0.729	0.228
0.266	0.750	1.056	0.936	0.911
0.225	0.326	0.643	0.337	0.721
0.187	0.586	0.529	0.340	0.829
0.153	0.485	0.560	0.428	0.628

Semi-Structured

```
{
    "_id" : 1001,
    "Name" : "Saleel Bagde",
    "canVote" :  true
},
{
    "_id" : 1002,
    "Name" : "Sharmin Bagde",
    "canVote" : true,
    "canDrive" : false
}
```

Unstructured

MongoDB stores **documents** (objects) in a format called **BSON**.
BSON is a binary serialization of JSON

Types of Data

- ***Structured***
  The data that can be stored and processed in a fixed format is called as Structured Data. Data stored in a relational database management system (RDBMS) is one example of 'structured' data. It is easy to process structured data as it has a fixed schema. Structured Query Language (SQL) is often used to manage such kind of Data.

- ***Semi-Structured***
  Semi-Structured Data is a type of data which does not have a formal structure of a data model, i.e. a table definition in a relational DBMS, XML files or JSON documents are examples of semi-structured data.

- ***Unstructured***
  The data which have unknown form and cannot be stored in RDBMS and cannot be analyzed unless it is transformed into a structured format is called as unstructured data. Text Files and multimedia contents like images, audios, videos are example of unstructured data.

MongoDB

MongoDB is a cross-platform document-oriented database program. Classified as a NoSQL database.

Remember:

- MongoDB documents are similar to JSON (key/fields and value pairs) objects.
- The values of fields may include other documents, arrays, or an arrays of documents.

Core MongoDB Operations (CRUD), stands for create, read, update, and delete.

SQL/MongoDB Terms:

MongoDB stores data as BSON documents. BSON is a binary representation of JSON documents.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write.

3Vs

3Vs (volume, variety and velocity) are three defining properties or dimensions of big data.

- ***Volume*** refers to the amount of data.
- ***Variety*** refers to the number of types of data.
- ***Velocity*** refers to the speed of data processing.

3Vs

Volume refers to the 'amount of data', which is growing day by day at a very fast pace. The size of data generated by humans, machines and their interactions on social media itself is massive.

Velocity is defined as the pace at which different sources generate the data every day. This flow of data is massive.

As there are many sources which are contributing to Big Data, the type of data they are generating is different. It can be structured, semi-structured or unstructured. Hence, there is a variety of data which is getting generated every day. Earlier, we used to get the data from excel and databases, now the data are coming in the form of images, audios, videos, sensor data etc. as shown in below image. Hence, this variety of unstructured data creates problems in capturing, storage, mining and analyzing the data.

* MongoDB does not support duplicate field names

- `cls`
- `console.clear();`

The maximum size an individual document can be in MongoDB is **16MB with a nested depth of 100 levels**.

document

MongoDB stores data as BSON documents. BSON is a binary representation of JSON documents.

MongoDB's collections, by default, do not require their documents to have the same schema. That is:

- The documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- To change the structure of the documents in a collection, such as add new fields, remove existing fields, or change the field values to a new type, update the documents to the new structure.

document

PERSON

Pers_ID	First_Name	Last_Name	City
0	Paul	Miller	London
1	Alvaro	Ortega	Valencia
2	Bianca	Bertolini	Rome
3	Auriele	Jackson	Paris
4	Urs	Huber	Zurich

CAR

Car_ID	Model	Year	Value	Pers_ID
101	Bently	1973	100000	0
102	Renault	1993	2000	3
103	Smart	1999	2000	2
104	Ferrari	2005	150000	4
105	Rolls Royce	1965	350000	0
106	Renault	2001	7000	3
107	Peugeot	1993	500	3

People Collection

```
{
  id: 0,
  first_name: 'Paul',
  last_name: 'Miller',
  city: 'London',
  cars: [
    {
      model: 'Bently',
      year: 1973,
      color: 'gold',
      value: NumberDecimal ('100000.00'),
        currency: 'USD',
      owner: 0
    },
{
      model: 'Rolls Royce,
      year: 1965,
      color: 'brewster green',
      value: NumberDecimal ('350000.00'),
        currency: 'USD',
      owner: 0
    }
  ]
}
```

Referencing Documents

Articles Collection

```
{
    _id: “5”,
    title: “Title 5”,
    body: “Great text here.”,
    author: “USER_1234”,
    …
}
```

Users Collection

```
{
    _id: “USER_1234”,
    password: “superStrong”,
    registered: <DATE>,
    lang: “EN”,
    city: “Seattle”,
    social: [ … ],
    articles: [“5”, “17”, …],
    …
}
```

document

MongoDB documents are composed of ***field-and-value*** pairs. The value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents.

The ***field name*** **_id** is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.

```
{
    field1: value,
    field2: value,
    field3: [],
    field4: {},
    field5: [ {}, {}, ... ]
    ...
    fieldN: valueN
}
```

The primary key **_id** is automatically added, if **_id** field is not specified.

Note:

- The **_id** field is always the first field in the documents.
- MongoDB does not support duplicate field names.

db

In the mongo shell, **db** is the variable that references the current database. The variable is automatically set to the default database **test** or is set when you use the **use <db_name>** to switch current database.

	MongoDB	Redis	MySQL	Oracle
Database Server	mongod	./redis-server	mysqld	oracle
Database Client	mongo	./redis-cli	mysql	sqlplus

start db server

start server and client

To start MongoDB server, execute **mongod.exe**.

Note: Always give --dbpath in ""

- The --dbpath option points to your database directory.
- The --bind_ip_all option : bind to all ip addresses.
- The --bind_ip arg option : comma separated list of ip addresses to listen on, localhost by default.

```
--bind_ip <hostnames | ipaddresses>
```

```
mongod --dbpath "c:\dBase" --bind_ip_all --journal --port=27017
mongod --dbpath "c:\dBase" --bind_ip stp10 --journal --port=27017
mongod --dbpath "c:\dBase" --bind_ip 192.168.100.20 --journal --port=27017
mongod --dbpath="c:\dBase" --bind_ip=192.168.100.20 --journal --port=27017
mongod --auth --dbpath="c:\dBase" --bind_ip=192.168.100.20 --journal --
port=27017
mongod --storageEngine inMemory --dbpath="d:\tmp" --bind_ip=192.168.100.20 --
port=27017
```

must be empty folder

```
--port=[27107, 27018, 27019, 27020, 27021]
```

start server and client

Note: Always give --dbpath in ""

To start MongoDB client, execute **mongosh.exe**.

```
mongosh "192.168.100.20:27017/primaryDB“
mongosh --host 192.168.100.20 --port 27017
mongosh --host 192.168.100.20 --port 27017 primaryDB
mongosh --host=192.168.100.20 --port=27017 primaryDB
mongosh --host=192.168.100.20 --port=27017 -u user01 -p user01 --
authenticationDatabase primaryDB
```

```
--port=[27107, 27018, 27019, 27020, 27021]
```

- `db.version();` # version number
- `db.getMongo();` # connection to 192.168.100.20:27017
- `db.hostInfo();` # Returns a document with information about the mongoDB is runs on.
- `db.stats();` # Returns DB status
- `getHostName();` # stp5

comparison operator

comparison operator

$eq	Matches values that are equal to a specified value.
$gt	Matches values that are greater than a specified value.
$gte	Matches values that are greater than or equal to a specified value.
$lt	Matches values that are less than a specified value.
$lte	Matches values that are less than or equal to a specified value.
$ne	Matches all values that are not equal to a specified value.
$in	Matches any of the values specified in an array.
$nin	Matches none of the values specified in an array.

comparison operator

$eq

```
{ field: { $eq: value } }
```

$ne

```
{ field: { $ne: value } }
```

$gt

```
{ field: { $gt: value } }
```

$gte

```
{ field: { $gte: value } }
```

$lt

```
{ field: { $lt: value } }
```

$lte

```
{ field: { $lte: value } }
```

$in

```
{ field: { $in: [ <value1>, <value2>, ..., <valueN> ] } }
```

$nin

```
{ field: { $nin: [ <value1>, <value2>, ..., <valueN> ] } }
```

logical operator

logical operator

$or	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.
$and	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
$not	Inverts the effect of a query expression and returns documents that do not match the query expression.

- ```
  db.emp.find( { $or:[ { job: 'manager' }, { job: 'salesman' } ], $and: [ { sal: { $gt: 3000 }} ] }, { _id: false, ename: true, job: true, sal: true } );
  ```
- ```
  db.emp.find({ JOB: { $in: [ 'manager', 'salesman' ] } }, { _id: false, ename: true, job:  true });
  ```

logical operator

$or

```
{ $or: [ { <expr1> }, { <expr2> }, ... , { <exprN> } ] }
```

- `db.emp.find({$or: [{job: 'manager'}, {job: 'salesman'}]})`

$and

```
{ $and: [ { <expr1> }, { <expr2> }, ... , { <exprN> } ] }
```

- `db.emp.find({$and: [{job:'manager'}, {sal:3400}]})`

$not

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

- `db.emp.find({ job: {$not: {$eq: 'manager'}}})`

ObjectId values are 12 bytes in length.

- A 4-byte timestamp, representing the ObjectId's creation, measured in seconds.
- A 5-byte random value generated once per process. This random value is unique to the machine and process.
- A 3-byte incrementing counter, initialized to a random value.

ObjectId()

```
The ObjectId class is the default primary key for a MongoDB document
and is usually found in the _id field in an inserted document.
```

The **_id** field must have a unique value. You can think of the **_id** field as the document's primary key.

ObjectId()

MongoDB uses ObjectIds as the default value of _id field of each document, which is auto generated while the creation of any document.

```
ObjectId()
```

- `x = ObjectId()`

show databases

Print a list of all available databases.

show database

Print a list of all databases on the server.

```
show  { dbs | databases }
```

- `show dbs`
- `show databases    # returns: all database name.`
- `db.adminCommand({ listDatabases: 1, nameOnly:true })`

```
db.getName()
```

- `db`
- `db.getName()      # returns: the current database name.`

To access an element of an array by the zero-based index position, concatenate the array name with the dot (.) and zero-based index position, and enclose in quotes

use database

Switch current database to <db>. The mongo shell variable db is set to the current database.

use database

Switch current database to <db>. The mongo shell variable db is set to the current database.

```
use <db>
```

- `use db1`

db.dropDatabase()

db.dropDatabase()

Removes the current database, deleting the associated data files.

```
db.dropDatabase()
```

- `use db1`
- `db.dropDatabase()`

If not working then do changes in *my.ini* file.

secure_file_priv = ""

- SELECT * FROM emp INTO OUTFILE "d:/emp.csv" FIELDS TERMINATED BY ',';

mongoimport

mongoimport tool imports content from an Extended **JSON**, **CSV**, or **TSV** export created by mongoexport, or another third-party export tool.

mongoimport - JSON

The *mongoimport* tool imports content from an Extended JSON, CSV, or TSV export created by *mongoexport*.

```
mongoimport < --host > < --port > < --db > < --collection > < --type >
< --file > < --fields "Field-List" > < --mode { insert | upsert | merge
} > < --jsonArray > < --drop >
```

```
< --jsonArray > # if the documents are in array i.e. in [] brackets
< --drop >      # drops the collection if exists
```

- `C:\> mongoimport  --host 192.168.0.3 --port 27017  --db db1 --collection emp --type json --file "d:\emp.json"`
- `C:\> mongoimport --host 192.168.0.6 --port 27017 --db db1 --collection movies --type json --file "d:\movies.json" --jsonArray  --drop`

mongoimport - CSV

The *mongoimport* tool imports content from an Extended JSON, CSV, or TSV export created by *mongoexport*.

```
mongoimport < --host > < --port > < --db > < --collection > < --type > < --file >
< --fields "<field1>[,<field2>,...]*" > < --columnsHaveTypes > < --headerline > <
--useArrayIndexFields > < --fieldFile > < --drop >
```

- `C:\> mongoimport --host 192.168.100.20 --port 27017 --db db1 --collection emp --type csv --file d:\emp.csv --headerline`
- `C:\> mongoimport --host 192.168.100.20 --port 27017 --db db1 --collection emp --type csv --file d:\emp.csv --fields "EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPTNO,BONUSID,USERNAME,PWD"`
- `C:\> mongoimport --db db1 --collection o --type csv --file d:\emp.csv --fields "EMPNO.int(32),ENAME.string(),JOB.string(),MGR.int32(),HIREDATE.date(2006-01-02),SAL.int32(),COMM.int32(),DEPTNO.int32(),BONUSID.int32(),USERNAME.string(),PWD.string()"`

Note:

- There should be no blank space in the field list.

  e.g.

  _id, ename, salary #this is an error

mongoimport - CSV

```
mongoimport < --host > < --port > < --db > < --collection > < --type > < --file >
< --fields "<field1>[,<field2>,...]*" > < --columnsHaveTypes > < --headerline > <
--useArrayIndexFields > < --fieldFile > < --drop >
```

```
_id,course,duration,modules.0,modules.1,modules.2,modules.3
1,course1,6 months,c++,database,java,.net
2,course2,6 months,c++,database,python,R
3,course3,6 months,c++,database,awp,.net
```

- `C:\> mongoimport --host 192.168.100.20 --port 27017 --db db1 --collection course --type csv --file d:\course.csv --drop --headerline --useArrayIndexFields`

mongoimport - CSV

```
mongoimport < --host > < --port > < --db > < --collection > < --type > < --file >
< --fields "<field1>[,<field2>,...]*" > < --columnsHaveTypes > < --headerline > <
--useArrayIndexFields > < --fieldFile > < --drop >
```

```
fieldFile.txt
_id
course
duration
modules.0
modules.1
modules.2
modules.3
```

```
course.csv
1,course1,6 months,c++,database,java,.net
2,course2,6 months,c++,database,python,R
3,course3,6 months,c++,database,awp,.net
```

- `C:\> mongoimport --host 192.168.100.20 --port 27017 --db db1 --collection course --type csv --file d:\course.csv --drop --fieldFile d:\fieldFile.txt --useArrayIndexFields`

Note:

- The --fieldFile option allows you to specify a file that holds a list of field names if your CSV or TSV file does not include field names in the first line of the file (i.e. header). Place one field per line.

mongoexport

mongoexport is a utility that produces a **JSON** or **CSV** export of data stored in a MongoDB instance.

mongoexport

mongoexport is a utility that produces a JSON or CSV export of data stored in a MongoDB instance..

```
mongoexport < --host > < --port > < --db > < --collection > < --type > < --file >
< --out >
```

- `C:\> mongoexport  --host 192.168.0.6 --port 27017  --db db1 --collection emp --type JSON  --out "d:\emp.json"`
- `C:\> mongoexport  --host 192.168.0.6 --port 27017  --db db1 --collection emp --type JSON --out "d:\emp.json" --fields "empno,ename,job"`
- `C:\> mongoexport  --host 192.168.0.6 --port 27017  --db db1 --collection emp --type CSV --out "d:\emp.csv" --fields "empno,ename,job"`

Note:

- there should be no space in the field list.
  e.g.
  _id, ename, salary #this is an error

new Date()

TODO

new Date()

MongoDB uses ObjectIds as the default value of _id field of each document, which is auto generated while the creation of any document.

```
var variable_name = new Date()
```

- x = Date()

db.getCollectionNames() / db.getCollectionInfos()

Returns an array containing the names of all collections and views in the current database.

db.getCollectionNames() / db.getCollectionInfos()

getCollectionNames() returns an array containing the names of all collections in the current database.

```
show collection
db.getCollectionNames()
db.getCollectionInfos()
```

- `show collections`
- `db.getCollectionNames()`
- `db.getCollectionInfos()`

db.createCollection()

Creates a new collection or view.

db.createCollection()

Capped collections have maximum size or document counts that prevent them from growing beyond maximum thresholds. All capped collections must specify a maximum size and may also specify a maximum document count. **MongoDB removes older documents if a collection reaches the maximum size limit before it reaches the maximum document count.**

```
db.createCollection(name, { options1, options2, ... })
```

The options document contains the following fields:

- capped : boolean
- size : number
- max : number

- `db.createCollection("log")`
- `db.createCollection("log", { capped: true, size: 1, max: 2})  // This command creates a collection named log with a maximum size of 1 byte and a maximum of 2 documents.`

The size parameter specifies the size of the capped collection in bytes.

convertToCapped

```
To convert the collection to a capped collection.
```

The size parameter specifies the size of the capped collection in bytes.

convertToCapped

To convert a non-capped collection to a capped collection, use the *convertToCapped* database command.

```
db.runCommand({ convertToCapped: collectionName, size: bytes, max: totalDocuments })
```

- `db.runCommand({ convertToCapped: 'log', size: 100, max: 2 });`

```
db.runCommand( { collMod: "log", cappedSize: 200 } );
```

- `db.runCommand({ collMod: 'log', cappedSize: 200 });`

```
db.runCommand( { collMod: "log", cappedMax: 7 } );
```

- `db.createCollection("log");`
- `db.runCommand({ convertToCapped: 'log', cappedMax: 7 });`

db.collection.isCapped()

Returns **true** if the collection is a capped collection, otherwise returns **false**.

db.collection.isCapped()

isCapped() returns true if the collection is a capped collection, otherwise returns false.

```
db.collection.isCapped()
```

- `db.log.isCapped()`

db.createCollection - validator

Collections with validation compare each inserted or updated document against the criteria specified in the validator option.

db.createCollection - validator

The *$jsonSchema* operator matches documents that satisfy the specified JSON Schema.

```
{ $jsonSchema: <JSON Schema object> }
```

- ```
  db.createCollection( "product", { validator: { $jsonSchema: {
  bsonType: "object", required: [ "code", "product", "price",
  "status", "isAvailable"  ],
  properties: {
        code: { bsonType: "string" },
        product: { bsonType: "string" },
        price:{ bsonType: "double", minimum: 1000, maximum: 5000
  },
        status: { enum: [ "in-store", "in-warehouse" ] },
        isAvailable : { bsonType: "bool"}
  }}}})
  ```
  ```

db.createCollection - validator

The *$jsonSchema* operator matches documents that satisfy the specified JSON Schema.

```
{ $jsonSchema: <JSON Schema object> }
```

```
db.createCollection( "person", { validator: { $jsonSchema: { bsonType: "object",
    required: [ "countryCode", "phone", "mobile", "status" ],
    properties: {
      countryCode: {
        bsonType: "string",
        description: "countryCode must be a string and is required"
      },
       mobile: {
         bsonType: "double",
         description: "mobile must be a integer and is required"
      },
      status: {
        enum: [ "Working", "Not Working"],
        description: "status must be a either ['Working', 'Not Working']"
      }
}}}})
```

db.getCollection()

Returns a collection or a view object that is in the DB.

db.getCollection()

TODO

```
db.getCollection('name')
```

- `db.getCollection('emp').find()`

-
```
const auth = db.getCollection("author")
const doc = {
        usrName : "John Doe",
        usrDept : "Sales",
        usrTitle : "Executive Account Manager",
        authLevel : 4,
        authDept : ["Sales", "Customers"]
    }

auth.insertOne( doc )
```

db.getSiblingDB()

To access another database without switching databases.

db.getSiblingDB()

Used to return another database without modifying the db variable in the shell environment.

```
db.getSiblingDB(<database>).runCommand(<command>)
```

- `db.getSiblingDB('db1').getCollectionNames()`

db.collection.renameCollection()

Renames a collection.

db.collection.renameCollection()

TODO

```
db.collection.renameCollection(target, dropTarget)
```

- `db.emp.renameCollection("employee", false)`

dropTarget : If true, mongod drops the target of renameCollection prior to renaming the collection. The default value is false.

db.collection.drop()

Removes a collection or view from the database. The method also removes any indexes associated with the dropped collection.

db.collection.drop()

drop() removes a collection or view from the database. The method also removes any indexes associated with the dropped collection.

```
db.collection.drop()
```

- `db.emp.drop()`

Method

.pretty()

Embedded Field Specification

For fields in an embedded documents, you can specify the field using either:

dot notation; e.g. `"field.nestedfield": <value>`
nested form; e.g. `{ field: { nestedfield: <value> } }`

For query on array elements:

array; e.g. `'<array>.<index>' // db.emp.find( {"phone.0": 111 } )`

db.collection.find()

The `find()` method always returns the `_id` field unless you specify `_id: 0/false` to suppress the field.

By default, mongo prints the first 20 documents. The mongo shell will prompt the user to **Type "it" to continue** iterating the next 20 results.

```
Enterprise primaryDB> config.set("displayBatchSize", 3)
```

- `db.emp.find( { }, { _id: false, sal: true, Per : { $multiply: [ "$sal", .05 ] }, NewSalary: { $add: [ "$sal", { $multiply: [ "$sal",  .05 ] } ] } } )`

db.collection.find()

TODO

```
db['collection'].find({ query }, { projection })
db.collection.find({ query }, { projection })
db.getCollection('name').find({ query }, { projection })
```

query: Specifies selection filter using query operators. To return all documents in a collection, omit this parameter or pass an empty document ({}).

{ "<Field Name>": { "<Comparison Operator>": <Comparison Value> } }

projection: Specifies the fields to return in the documents that match the query filter. To return all fields in the matching documents, omit this parameter.

{ "<Field Name>": <Boolean Value> } }

Remember

- 1 or true to include the field in the return documents. Non-zero integers are also treated as true.
- 0 or false to exclude the field.

db.collection.find()

TODO

'<array>.<index>'

```
db['collection'].find({ query }, { projection })
db.collection.find({ query }, { projection })
db.getCollection('name').find({ query }, { projection })
```

- `db.emp.find();`
- `db ["emp"].find ()`
- `db.getCollection("emp").find()`
- `db.getSiblingDB("db1").getCollection("emp").find()`
- `db.emp.find({job: "manager"})`
- `db.emp.find({}, {ename: true, job: true})`
- `db.emp.find({sal:{ $gt: 4 }})`
- `db.emp.find({job: "manager"}, {ename: true, job: true})`
- `db.emp.find({job: "manager"}, {_id: false, ename: true, job: true})`

db.collection.find()

TODO

```
db['collection'].find({ query }, { projection })
db.collection.find({ query }, { projection })
db.getCollection('name').find({ query }, { projection })
```

- `const query1 = { "job": "manager" };`
- `const query2 = { "sal": { $gt: 6000, $lt: 6500 } };`
- `const projection = { "_id" : false, "ename": true, "job": true, "sal": true , "address": true };`

- `db.emp.find( query1, projection )`
- `db.emp.find( query2, projection )`

TODO

```
delete < variable_name >
```

- `delete query1`

$exits

The **$exists** operator matches documents that contain or do not contain a specified field, including documents where the field value is null.

```
{ field: { $exists: <boolean> } }
```

- `db.emp.find({ phone: { $exists: true } }, { _id: false, ename: true });`
- `db.movies.aggregate([ { $match: { movie_title: { $exists: true }, movie_title: 'The Dark Knight Rises' } }, { $project: { _id: false, movie_title: true } } ]);`

$where

The **$where** operator in MongoDB allows you to pass either a string containing a JavaScript expression or a full JavaScript function to the query system.

```
{ $where: <string | JavaScript Code> }
```

- `db.employee.find({ $where: "this.sal > 3000" }, { _id: false, ename: true, sal: true });`
- `db.employee.find({ $where: function() { return true; } });`
- `db.employee.find({ $where: function() { return this.sal > 3000; } }, { _id: true, ename: true, job: true, sal: true, phone: true });`
- `db.employee.find({ $where: function() { if ( this.sal > 4000 ) { return true; } } }, { _id: false, ename: true, sal: true });`

alias name

TODO

- `db.emp.find({}, {_id:false, "Employee Name" : "$ename" })`

pattern matching / $regex

For patterns that include anchors (i.e. ^ for the start, $ for the end), match at the beginning or end of each line for strings with multiline values. Without this option, these anchors match at beginning or end of the string.

```
{ <field>: /pattern/ }
```

- `db.movies.find({movie_title:/z/}, {_id:false, movie_title:true})`
- `db.movies.find({movie_title:/^z/}, {_id:false, movie_title:true})`
- `db.movies.find({movie_title:/z$/}, {_id:false, movie_title:true})`
- `db.movies.aggregate([ {$match:{movie_title:/z$/}}, {$project:{_id:false, movie_title:true}} ])`
- `db.movies.aggregate([ { $match:{ genres: /^Horror$/ }}, { $project: {_id: false, "Title": "$movie_title", "Genres": "$genres", "Director": "$director"}} ])`

```
{ <field>: { $regex: /pattern/<options> } }
{ <field>: { $regex: /pattern/, $options: '<options>' }
```

You cannot use $regex operator expressions inside an $in operator.

- `db.emp.find({ ename:{$regex: /saleel/i} }, { _id: false, ename: true } )`
- `db.emp.aggregate({ $match: { $or: [{ ename: { $regex: /^sa/ } }, { ename: { $regex: /^sh/ } }] } }, { $project: { _id: false, ename: true } })`

db.collection.find() .toArray()[<index_number>]

The **toArray()** method returns an array that contains all the documents from a cursor.

```
db['collection'].find({ query }, { projection }).toArray()[ <index> ][.field]
db.collection.find({ query }, { projection }) }).toArray()[<index> ][.field]
db.getCollection('name').find({ query }, { projection }) }).toArray()[<index>
][.field]
```

- `db.movies.find().toArray()[0]`
- `db.movies.find().toArray()[0].movie_title`
- `db.getCollection("movies").find().toArray()[0]`
- `db.movies.find().toArray()[db.movies.countDocuments({}) - 2]`

cursor with db.collection.find()

In the mongo shell, if the returned cursor is not assigned to a variable using the var keyword, the cursor is automatically iterated to access up to the first 20 documents that match the query.

```
var variable_name = db.collection.find({ query }, { projection })
```

The find() method returns a cursor.

```
var x = db["emp"].find()
x.forEach(printjson)
```

sort

Specifies the order in which the query returns matching documents. You must apply **sort()** to the cursor before retrieving any documents from the database.

db.collection.find().sort({ })

sort() specifies the order in which the query returns matching documents. You must apply ***sort()*** to the cursor before retrieving any documents from the database.

```
cursor.sort({ field: value })
db['collection'].find({ query }, { projection }).sort({ field: value })
db.collection.find({ query  }, { projection }).sort({ field: value })
```

Specify in the sort parameter

- 1 to specify an ascending sort.
- -1 to specify an descending sort.

- `db["emp"].find({}, {ename: true}).sort({ename: 1})`
- `db["emp"].find({}, {ename: true}).sort({ename: -1})`

limit

limit() method on a cursor to specify the maximum number of documents the cursor will return.

A limit() value of 0 (i.e. limit(0)) is equivalent to setting no limit.

db.collection.find().limit()

limit() method specify the maximum number of documents the cursor will return.

```
cursor.limit(<number>)
db['collection'].find({ query }, { projection }).limit(<number>)
db.collection.find({ query }, { projection }).limit(<number>)
```

- `db["emp"].find({}, { ename: true }).limit(0) # all documents`
- `db["emp"].find({}, { ename: true }).limit(2)`

skip

skip() method on a cursor to control where MongoDB begins returning results.

db.collection.find().skip()

skip() method is used for skipping the given number of documents in the Query result.

```
cursor.skip(<offset_number>)
db['collection'].find({ query }, { projection }).skip(<offset_number>)
db.collection.find({ query }, { projection }).skip( < offset_number > )
```

- `db.emp.find().skip(4)`
- `db.emp.find().skip(db.emp.countDocuments({}) - 1)`

count

Counts the number of documents referenced by a cursor. Append the **count()** method to a **find()** query to return the number of matching documents. The operation does not perform the query but instead counts the results that would be returned by the query.

db.collection.find().count()

count() counts the number of documents referenced by a cursor. Append the ***count()*** method to a ***find()*** query to return the number of matching documents. The operation does not perform the query but instead counts the results that would be returned by the query.

```
cursor.count()
db['collection'].find({ query }).count()
db.collection.find({ query }).count()
```

- `db.emp.find().count()`
- `db.emp.find({job: "manager"}).count()`

db.collection.distinct()

Finds the distinct values for a specified field across a single collection or view and returns the results in an array.

db.collection.distinct()

distinct() finds the distinct values for a specified field across a single collection or view and returns the results in an array.

```
db.collection.distinct("field", { query }, { options })
```

- `db.emp.distinct("job")`
- `db.emp.distinct("job").length  //count distinct job's`
- `db.emp.distinct("job", { sal: { $gt: 5000 } } )`

```
var x = db.emp.find()[10]
for (i in x) {
    print(i)
 }
```

db.collection.count[Documents]()

```
TODO
```

db.collection.count[Documents]()

countDocuments() returns the count of documents that match the query for a collection

```
db.collection.count[Documents]({ query }, { options })
```

Field	Description
limit	Optional. The maximum number of documents to count.
skip	Optional. The number of documents to skip before counting.

- `db.emp.count({})`
- `db.emp.countDocuments({})`
- `db.emp.countDocuments({job: "manager"})`
- `db.emp.countDocuments({job: "salesman"}, {skip: 1, limit: 3})`

findOne

find() method always returns the **_id field** unless you specify **_id: 0/false** to suppress the field.

db.collection.findOne()

findOne() returns one document that satisfies the specified query criteria on the collection. If multiple documents satisfy the query, this method returns the first document according to the order in which order the documents are stored in the disk. If no document satisfies the query, the method returns null.

```
db['collection'].findOne({ query } , { projection })
db.collection.findOne({ query } , { projection })
```

- `db.emp.findOne()`
- `db.emp.findOne({ job: "manager" })`
- `db.movies.findOne({ _id: 4 }, {}).movie_title`
- `typeof db.movies.findOne({}, {} ).movie_title`

- If the document does not contain an _id field, then the save() method calls the insert() method. During the operation, the mongo shell will create an ObjectId and assign it to the _id field.
- If the document contains an _id field, then the save() method is equivalent to an update with the upsert option set to true and the query predicate on the _id field.

db.collection.save()

Updates an existing document or inserts a new document, depending on its document parameter.

db.collection.save()

save() UPDATES an existing document or INSERTS a new document, depending on its document parameter.

```
db.collection.save({ document })
```

- ```
  db.x.save({_id: 10, firstName: "neel", sal: 5000, color: ["blue", "black" ], size: ["small", "medium", "large", "xx-large" ] })
  ```
  ```

db.collection.insert()

Inserts a document or documents into a collection.

db.collection.insert() or db.collection.insert([])

insert() inserts a **single-document** or **multiple-documents** into a collection.

```
db.collection.insert({<document>})
db.collection.insert([{<document 1>} , {<document 2>}, ... ])
```

- `db.x.insert({})`
- `db.x.insert({ ename: "ram", job: "programmer", salary: 42000})`
- `db.x.insert([ { ename: "sham"} , { ename: "y" } ]) # for multiple documents.`

- `const doc1 = { "name": "basketball", "category": "sports", "qty": 20, "rate": 3400, "reviews": [] };`
- `const doc2 = { "name": "football",  "category": "sports", "qty": 30, "rate": 4200, "reviews": [] };`
- `db.games.insert([ doc1, doc2 ])`

db.collection.insertOne() & db.collection.insertMany()

```
Inserts a document into a collection.
Inserts multiple documents into a collection.
```

db.collection.insertOne() & db.collection.insertMany([])

- ```
  db.shapes.insertMany([
  { _id: "◐", x: "■", y: "▲", val: 10, ord: 0}
  { _id: "◑", x: "■", y: "■", val: 60},
  { _id: "◒", x: "●", y: "■", val: 80},
  {_id: "◓", x: "▲", y: "▲", val: 85},
  { _id: "◔", x: "■", y: "▲", val: 90},
  { _id: "◕", x: "●", y: "■", val: 95, ord: 100}
  ]);
  ```

- ```
  db.lists.insertMany([
  { _id: "▤", a: "●", b: ["◰", "◱"]},
  { _id: "▥", a: "▲", b: ["◲"]},
  { _id: "▦", a: "▲", b: ["◰", "◳", "◱"]},
  { _id: "▧", a: "●", b: ["◰"]},
  { _id: "▨", a: "■", b: ["◳", "◱"]},
  ]);
  ```

db.collection.insertOne() & db.collection.insertMany([])

insertOne() inserts a single document into a collection.

insertMany() inserts a document or multiple documents into a collection.

```
db.collection.insertOne({<document>})
db.collection.insertMany([{<document 1>} , {<document 2>}, ... ])
```

- `db.emp.insertOne({ ename: 'ram', job: 'programmer', salary: 2000 })`
- `db.emp.insertMany([ { ename: 'sham', salary: 2000}, { ename : 'raj', job: 'programmer' } ])`

- `const doc1 = { "name": "basketball", "category": "sports", "quantity": 20, "reviews": [] }`
- `const doc2 = { "name": "football",   "category": "sports", "quantity": 30, "reviews": [] }`
- `db.games.insertMany([ doc1, doc2 ])`

one-to-one collection
and
one-to-many collection

```
Inserting record in bulk.
```

one-to-one collection – embedded pattern

Embedded Document Pattern.

person-passport Collection

```
• db.person.insertMany([ {
    _id: "saleel",
    name: "saleel",
    passport: {
        "passport number": "AXITUD1092",
        "country code": "IN",
        "issue date": "24-July-1988",
        "valid to": "24-July-2008"
    }
  }, {
    _id: "sharmin",
    name: "sharmin",
    passport: {
        "passport number": "DKSK100SK",
        "country code": "IN",
        "issue date": "04-May-1998",
        "valid to": "04-May-2018"
    }
  }])
```

one-to-one collection – subset pattern

Subset Pattern.

person Collection

```
• db.person.insertMany([
    { _id: "saleel", name: "saleel", city: "pune", state: "MH" },
    { _id: "sharmin", name: "sharmin", city: "pune", state: "MH" }
  ])
```

passport Collection

```
• db.passport.insertMany([
    { _id: "saleel", "passport number": "AXITUD1092", "country code": "IN",
      "issue date": "24-July-1988", "valid to": "24-July-2008" },
    { _id: "sharmin", "passport number": "DKSK100SK", "country code": "IN",
      "issue date": "04-May-1998", "valid to": "04-May-2018" }
   ])
```

one-to-many collection – embedded pattern

Embedded Document Pattern.

Order-details Collection

```
• db.orders.insertMany([
    { "_id": 1, "orderDay": "Mon", "cart": [
        { "item": "maggi", "price": 40, "quantity": 7 },
        { "item": "butter", "price": 125, "quantity": 12 },
        { "item": "cheese", "price": 225, "quantity": 12 }
      ]
    },
    { "_id": 2, "orderDay": "Sat", "cart": [
        { "item": "coffee", "price": 75, "quantity": 1 },
        { "item": "tea", "price": 175, "quantity": 3 },
        { "item": "jam", "price": 375, "quantity": 2 }
      ]
    },

    { "_id": 3, "orderDay": "Sat" }
  ])
```

one-to-many collection – subset pattern

Subset Pattern.

orders Collection

```
db.orders.insertMany([
  { "_id": 1, "orderDay": "Mon" },
  { "_id": 2, "orderDay": "Sat" },
  { "_id": 3, "orderDay": "Sat" }
])
```

orderdetails Collection

```
db.orderdetails.insertMany([
  { "_id": 1, "orderNo": 1, "item": "maggi", "price": 40, "quantity": 7 },
  { "_id": 2, "orderNo": 1, "item": "butter", "price": 125, "quantity": 12 },
  { "_id": 3, "orderNo": 1, "item": "cheese", "price": 225, "quantity": 12 },
  { "_id": 4, "orderNo": 2, "item": "coffee", "price": 75, "quantity": 1 },
  { "_id": 5, "orderNo": 2, "item": "tea", "price": 175, "quantity": 3 },
  { "_id": 6, "orderNo": 2, "item": "jam", "price": 375, "quantity": 2 }
])
```

one-to-many collection – subset pattern

Subset Pattern.

books Collection

```
db.books.insertMany([
  { _id: 1, title: "redis" },
  { _id: 2, title: "mongodb" },
  { _id: 3, title: "hbase" },
  { _id: 4, title: "pig" },
  { _id: 5, title: "python" },
  { _id: 6, title: "neo4j" },
  { _id: 7, title: "javascript" },
  { _id: 8, title: "c++" }
])
```

author Collection

```
db.author.insertMany([
  { _id : 1, name: "saleel", bookID: [ 1, 3, 5 ] },
  { _id: 2, name: "sharmin", bookID: [ 2, 4, 6, 8 ] },
  { _id: 3, name: "vrushali", bookID: [ 1, 3, 4, 6, 7 ] }
])
```

```
db.author.aggregate([{ $lookup: { from: "books",
localField: "bookID", foreignField: "_id", as:
"Book Information" }}])
```

array methods

- `db.orders.updateOne({ _id: 2 }, { $push: { cart: { item: "bread", price: 45, quantity: 2 } } })`
- `db.orderItems.updateOne({ _id: 1 }, {$unset: { "cart.3": 1 }})`
- `db.orderItems.updateOne({ _id: 1 }, { $pop: { "cart": 1 }})`

var bulk =
db.collection.initializeUnorderedBulkOp()

```
Inserting record in bulk.
```

var bulk = db.collection.initializeUnorderedBulkOp()

A huge number of documents can also be inserted in an unordered manner by executing ***initializeUnorderedBulkOp()*** methods.

```
var bulk = db.collectionName.initializeUnorderedBulkOp()
```

```
var bulk = db.dept.initializeUnorderedBulkOp();
bulk.insert({"deptno" : 50, "dname" : "purchase", "loc" : "new york" });
bulk.insert({"deptno" : 60, "dname" : "hrd", "loc" : "new york" });
bulk.insert({"deptno" : 70, "dname" : "r&d", "loc" : "chicago" });
bulk.execute();
```

Full Stack JavaScript Developer

- MEAN stack: MongoDB + Express + AngularJS - Node.js
- MERN stack: MongoDB + Express + React.js + Node.js
- MEVN stack: MongoDB + Express + Vue.js + Node.js

javascript object

```
TODO
```

javascript object

Inserts a document or documents into a collection using javascript object.

```
var obj = {}

> var doc = {};                          # JavaScript object
> doc.title = "MongoDB Tutorial"
> doc.url = "http://mongodb.org"
> doc.comment = "Good tutorial video"
> doc.tags = ['tutorial', 'noSQL']
> doc.saveondate = new Date ()
> doc.meta = {}                          # object within doc object{}
> doc.meta.browser = 'Google Chrome'
> doc.meta.os = 'Microsoft Windows7'
> doc.meta.mongodbversion = '2.4.0.0'
> doc

> db.book.insert(doc);

> doc                 -> will print entire document.
> doc.Title           -> will print only Title from document.
> print(doc)          -> will print -> [object Object].
> print(doc.Title)    -> will print only Title from document.
```

After executing a file with load(), **you may reference any functions or variables defined the file from the mongo shell environment.**

load ("app.js")

```
Loads and runs a JavaScript file into the current shell environment.
```

load(file.js)

Specifies the path of a JavaScript file to execute.

```
load(file)
cat(file)
```

- ```
  function app(x, y) {
      return (x + y);
  }
  ```
- ```
  function app1(x, y, z) {
      return (x + y + z);
  }
  ```

- load("scripts/app.js")
- cat("scripts/app.js")

javascript function

```
db.emp.find({$or:[ {job:'manager'}, {job:'salesman'} ]}, {}).forEach(function(doc) {
    print(doc.ename.padEnd(12, "-") + doc.job);
});
```

```
db.emp.find().forEach(function(doc) {
    if(doc.ename == 'saleel') {
        print(doc.ename, doc.job);
    } else {
        quit;
    };
});
```

```
db.emp.find().forEach(function(doc) {
    x = doc.job.split(" ");
    print(x[0]);
});
```

```
db.emp.find().forEach((doc) => {
    if (doc.ename.length >= 7) {
        print(doc.ename + ": " + doc.ename.length);
    };
});
```

```
db.emp.find().forEach(function(doc) {
  print("user:" + doc.ename.toUpperCase();)
});
```

javascript function

```
db.emp.find().forEach(function(doc) {
   if(doc.job.split(' ')[1]=='Programmer' || doc.job=='programmer') {
      print(doc.ename, doc.job);
   }
});
```

```
function findProductByID(_productID) {
   return db.products.find({productID: _productID}, {_id:false,
      productID:true, productname:true});
};
```

```
function fn() {
      var x = db.emp.count();
      return db.emp.find().limit(x > 10 ? 1 : 2)
};
```

```
db.getSiblingDB("primaryDB").movies.find().forEach((doc) => {
    db.movie.insertOne(doc)
});
```

javascript function

```
• function insertOnlyPune(id, _name, _sal, _comm, _city) {
    if(_city == 'pune') {
        db.abc.insertOne({
            _id: id,
            ename: _name,
            sal: _sal,
            comm: _comm,
            grandSalary: _sal + _comm
        })
      };
  };
```

```
• function insertProduct(_productID, _productName, _color, _rate, _qty) {
    db.product.insert({
        productID: _productID,
        productName: _productName,
        color: _color,
        rate: _rate,
        qty: _qty,
        total: _qty * _rate
    });
  };
```

javascript function

```
function fn() {
   db.books.find().forEach(function(doc) {
     db.books.updateOne(
       { _id: doc._id}, { $set: {total: doc.price + 2}})
   })
};
```

```
function fn() {
   db.movies.find().forEach(function(doc) {
      db.movies.updateMany({_id: doc._id},{$set:{"Running Time min" :
          (Math.floor(Math.random() * 700) + 99 ) }})
   })
};
```

```
function deleteProduct(_productID) {
   db.product.deleteOne({_id:_productID});
};
```

javascript function

```
function findProductByRangeID(_startID, _endID) {
   return db.products.find({$and:[{productID:{$gte: _startID}}, {
     productID:{$lte: _endID}}]}, {_id:false, productID:true,
     productname:true });
};
```

```
function productValidation(_productID) {
   var x = db.products.find({productID:_productID}).count();
   if(x != 0) {
      return db.products.find({productID: _productID}, {_id:false,
         productID:true, productname:true});
   } else {
     return ("Document not found!");
   };
};
```

javascript function

```
• let fn = () => {
    db.movies.aggregate([]).forEach((doc) => {
      db.movies.updateOne({_id: doc._id}, {$set:{r: Math.round(Math.random()*800)+100 }});
    });
  };
• fn();
```

```
• let auto_increment = (title, author, pages, language, rate) => {
      let a = db.books.count({}) + 1;
      db.books.insertOne({
         _id: a,
         title: title,
         author: author,
         pages: pages,
         language: language,
         rate: rate
      });
};
```

javascript function

```
• let split_rs = () => {
      /* split Rs.970  into Rs and 970 */
      db.books.aggregate([]).forEach((doc) => {
            for(key in doc) {
                  if(key == 'rate'){
                        print(doc.rate.split("."));
                  };
            };
      });
};
```

Null Test

```
if (variable === null)

- variable == ""; (false) typeof variable is = string

- variable == null; (true) typeof variable is = object

- variable == undefined; (false) typeof variable is = undefined

- variable == false; (false) typeof variable is = boolean

- variable == 0; (false) typeof variable is = number

- variable == NaN; (false) typeof variable is = number
```

Empty String Test

```
if (variable === '')

- variable = ''; (true) typeof variable is = string

- variable = null; (false) typeof variable is = object

- variable = undefined; (false) typeof variable is = undefined

- variable = false; (false) typeof variable is = boolean

- variable = 0; (false) typeof variable is = number

- variable = NaN; (false) typeof variable is = number
```

Undefined Test

```
if (typeof variable == "undefined")

-- or --

if (variable === undefined)

- variable = ''; (false) typeof variable is = string

- variable = null; (false) typeof variable is = object

- variable = undefined; (true) typeof variable is = undefined

- variable = false; (false) typeof variable is = boolean

- variable = 0; (false) typeof variable is = number

- variable = NaN; (false) typeof variable is = number
```

False Test

```
if (variable === false)

- variable = ''; (false) typeof variable is = string

- variable = null; (false) typeof variable is = object

- variable = undefined; (false) typeof variable is = undefined

- variable = false; (true) typeof variable is = boolean

- variable = 0; (false) typeof variable is = number

- variable = NaN; (false) typeof variable is = number
```

Zero Test

```
if (variable === 0)

- variable = ''; (false) typeof variable is = string

- variable = null; (false) typeof variable is = object

- variable = undefined; (false) typeof variable is = undefined

- variable = false; (false) typeof variable is = boolean

- variable = 0; (true) typeof variable is = number

- variable = NaN; (false) typeof variable is = number
```

NaN Test

```
if (typeof variable == 'number' && !parseFloat(variable) && variable !== 0)

-- or --

if (isNaN(variable))

- variable = ''; (false) typeof variable is = string

- variable = null; (false) typeof variable is = object

- variable = undefined; (false) typeof variable is = undefined

- variable = false; (false) typeof variable is = boolean

- variable = 0; (false) typeof variable is = number

- variable = NaN; (true) typeof variable is = number
```

x == y

x \ y	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[[]]	[0]	[1]	NaN
true	■		■					■												■	
false		■		■					■		■					■		■	■		
1	■		■					■												■	
0		■		■					■		■					■		■	■		
-1					■					■											
"true"						■															
"false"							■														
"1"	■		■					■												■	
"0"		■		■					■										■		
"-1"					■					■											
""		■		■							■					■		■			
null												■	■								
undefined												■	■								
Infinity														■							
-Infinity															■						
[]		■		■							■										
{}																					
[[]]		■		■							■										
[0]		■		■					■												
[1]	■		■					■												■	
NaN																					

x === y

	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[[]]	[0]	[1]	NaN
true	■																				
false		■																			
1			■																		
0				■																	
-1					■																
"true"						■															
"false"							■														
"1"								■													
"0"									■												
"-1"										■											
""											■										
null												■									
undefined													■								
Infinity														■							
-Infinity															■						
[]																					
{}																					
[[]]																					
[0]																					
[1]																					
NaN																					

db.collection.update()

Modifies an existing document or documents in a collection. The method can modify specific fields of an existing document or documents or replace an existing document entirely, depending on the update parameter. By default, the **update()** method updates a single document. Set the Multi Parameter to update all documents that match the query criteria.

db.collection.update()

By default, the update() method updates a single document. Set the multi Parameter to update all documents that match the query criteria, an upsert means an update than inserts a new document if no document matches the filter.

```
db.collection.update({ query }, { update }, { options })
db.collection.update({ query }, { $set:{ update }}, { options })
```

```
Options : { $set: { field: value } }, { multi: true, upsert: true }
```

- `db.emp.update({ job: "programmer" }, { job: "sales" }, { upsert: true } )`
- `db.emp.update({ job: "programmer" }, { $set: { job: "sales" } }, { upsert : true,  multi: true })`
- `db.emp.update({ ename: "ram" }, { $set : { size: "small", color: [ "red", "blue" ] } }, { multi: true } )`

- `db.emp.updateOne({ sal: { $gt : 2000 } }, { $set: { color : [ "red", "yellow", "green", "blue"] } });`
- `db.emp.updateMany({ sal: { $gt : 2000 } }, { $set: { color : [ "red", "yellow", "green", "blue" ] } });`

db.collection.updateOne()
db.collection.updateMany()

updateOne() operations can add fields to existing documents using the $set operator.

updateMany() operations can add fields to existing documents using the $set operator.

db.collection.updateOne() / .updateMany()

- *updateOne()* updates a single document within the collection based on the filter. an `upsert` means an update than inserts a new document if no document matches the filter.
- *updateMany()* updates multiple documents within the collection based on the filter. an `upsert` means an update than inserts a new document if no document matches the filter.

```
db.collection.updateOne({ filter }, { $set:{update} }, { options })

db.collection.updateMany({ filter }, { $set:{update} }, { options })

{ $set: { field: value }, { upsert: true }
```

Note:

- The $set operator replaces the value of a field with the specified value.
- If the field does not exist, $set will add a new field with the specified value.
- If you specify multiple field-value pairs, $set will update or create each field.
- To specify a <field> in an embedded document or in an array, use dot notation.

$set

The $set operator replaces the value of a field with the specified value.

$set

The *$set* operator replaces the value of a field with the specified value.

```
{ $set: { field1: value, field2: value >, ... }
[ { $set: { field1: value, field2: value >, ... } ]
```

- `db.x.updateOne({ }, { $set: { a: 100, b: 200, c: 300 } } );`
- `db.x.updateOne({ }, [ { $set: { a: 100, b: 200, c: 300 } } ] );`

Array Update

- `db.x.updateOne({ _id: 1 }, { $set: { "colors.2": "lime yellow" }});`
- `db.x.updateOne({ _id: 1 }, { $set: { "colors.2": "green", "colors.3": "pink" }});`
- `db.x.updateOne({ _id: 1 }, { $set: { "colors.2": { $exists: true }, "colors.6": { $exists: true } }, { "colors.2": "green", "colors.6": "pink" }});`

$inc

$inc operator increments a field by a specified value.

$inc

The *$inc* operator increments a field by a specified value.

```
{ $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }

{ $set: { field1: value, field2: value },
{ $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }
```

- `db.emp.updateOne({ sal: { $gt: 300 } }, { $inc: { sal: 1 } });`
- `db.emp.updateOne({ _id: 2 }, { $set: { comm: 111 } }, { $inc: { sal: 1 } });`

- `db.x.updateOne({ _id: 1 }, { $set: { "colors.2": "ABC" }, $inc: { salary: 1 }});`
  `// $set and $inc in single statement.`
- `db.x.updateOne({ _id: 1 }, { $set: { "colors.2": "ABC" }, $inc: { salary: 1 },`
  `$unset: { numbers: true }});  // $set, $inc, and $unsset`

$min / $max / $mul

$inc operator increments a field by a specified value.

$min / $max / $mul

TODO

```
{ $min: { <field1>: < value1 >, <field2>: < value2 >, ... } }
{ $max: { <field1>: < value1 >, <field2>: < value2 >, ... } }
{ $mul: { <field1>: < value1 >, <field2>: < value2 >, ... } }
```

- `db.x.updateOne({ _id : 1 }, { $min: { a: 35, b: 45 } });`
- `db.x.updateOne({ _id : 1 }, { $max: { a: 35, b: 45 } });`
- `db.x.updateOne({ _id : 1 }, { $mul: { a: 4, b: 5 } });`

Note:

- If the field does not exists, the $min operator sets the field to the specified value.
- If the field does not exists, the $max operator sets the field to the specified value.
- If the field does not exist in a document, $mul creates the field and sets the value to zero of the same numeric type as the multiplier.

$unset

$unset operator deletes a particular field.

$unset

The *$unset* operator deletes a particular field or multiple fields using [{ $unset }].

```
{ $unset: { <field1>: "", ... } }

[
  { $set: { field1: value, field2: value },
  { $unset: [ <field1>, <field2>, <field3>, ... ] }
]
```

Single field

- `db.emp.updateMany({_id: 1 }, { $unset: { comm: true, ename: '', sal: true }});`

Multiple fields

- `db.emp.updateOne({ _id: 1 }, [ { $unset: [ 'name', 'sal' ] }]);`
- `db.emp.updateOne({ _id: 1 }, [ { $set:{ x: 100 }}, { $unset: ['name', 'sal'] }]);`

$rename

$rename operator updates the name of a field.

$rename

The *$rename* operator updates the name of a field.

```
{ $rename: { <oldfield1>: <newName1>, <oldfield2>: <newName2>, ... } }
```

- `db.emp.update({}, { $rename: { "ename": "Employee Name", "sal": "Salary" }})`
- `db.emp.updateOne({}, { $rename: { "ename": "Employee Name", "sal": "Salary" } })`
- `db.emp.updateMany({}, { $rename: { "ename": "Employee Name", "sal": "Salary" } })`

array update

- `{ $push: { <field1>: { field: value, field: value, ... }, ... } }`
- `{ $push: { <field1>: { $each: [value1, value2, ...  ], $position: <num> } } }`
- `{ $pop: { <field>: <-1 | 1>, ... } }`
- `{ $pull: { <field1>: <value | condition>, <field2>: <value | condition>, ... } }`
- `{ $pullAll: { <field1>: [ <value1>, <value2> ... ], ... } }`
- `{ $addToSet: { <field1>: <value1>, ... } }`

Note:

- The $push operator appends a specified value to an array <field>.
- The $each with $push operator to append multiple values to an array <field>.
- The $pop operator removes the first or last element of an array. Pass value of `-1` to remove the first element of an array and `1` to remove the last element in an array.
- The $pull operator removes from an existing array all the value or values that match a specified condition.
- The $pullAll operator removes all instances of the specified values from an existing array. Unlike the $pull operator that removes elements by specifying a query, $pullAll removes elements that match the listed values.
- The $addToSet operator adds a value to an array unless the value is already present, in which case $addToSet does nothing to that array.

array update

- { $push: { <field1>: { field: value, field: value, ... }, ... } }
- { $push: { <field1>: { $each: [value1, value2, ...], $position: <num> } } }
- { $pop: { <field>: <-1 | 1>, ... } }
- { $pull: { <field1>: <value | condition>, <field2>: <value | condition>, ... } }
- { $pullAll: { <field1>: [<value1>, <value2> ...], ... } }
- { $addToSet: { <field1>: <value1>, ... } }
- { $slice: <num> }
- { query }, { $set: { "<array>.$": value } }
- { query }, { "<array>.$[]": value }
- { query }, { "<array>.$[elem]": value }, { arrayFilters: [{ condition }] }

- db.books.updateOne({ _id: 1 }, { $push: { languages: "french" } });
- db.books.updateOne({ _id: 1 }, { $push: { website: { $each: ["redis.com", "redis.io"] } } });
- db.books.updateOne({ _id: 1 }, { $push: { email: { $each: ["redis.com", "redis.io"], $position : 0 } } });
- db.person.updateOne({}, { $pull: { phone: -333 } });
- db.person.updateOne({}, { $pull: { website: { $in: ["redis.com", "redis.io"] } } });

array update

- { $push: { <field1>: { field: value, field: value, ... }, ... } }
- { $push: { <field1>: { $each: [value1, value2, ...], $position: <num> } } }
- { $pop: { <field>: <-1 | 1>, ... } }
- { $pull: { <field1>: <value | condition>, <field2>: <value | condition>, ... } }
- { $pullAll: { <field1>: [<value1>, <value2> ...], ... } }
- { $addToSet: { <field1>: <value1>, ... } }
- { $slice: <num> }
- { query }, { $set: { "<array>.$": value } }
- { query }, { "<array>.$[]": value }
- { query }, { "<array>.$[elem]": value }, { arrayFilters: [{ condition }] }

- db.person.updateOne({}, { $pull: { Rating: { $gt: 6 }}});
- db.books.updateOne({ _id: 1 }, { $addToSet: { email: "redis.com" } });
- db.x.updateOne({colors: "lemon"}, { $set:{ "colors.$": "black" } });
- db.x.updateMany({}, { $set: { "color.$[elem]" : "White" } }, { arrayFilters: [{ elem : "red" }] });
- db.x.updateOne({ _id: 1, number: { $gt: 60 }}, {$inc:{"number.$[]": 5}});
- db.x.updateMany({ numbers: { $in: [44, 68] } }, {$inc: { "numbers.$[]": 1 }});
- db.x.updateMany({}, { $inc: { "numbers.$[elem]": 1 }}, { arrayFilters: [{ elem: 45 }] });

```
{ $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }
```

```
{ $unset: { <field1>: "", ... } }
```

```
{ $rename: { <oldfield1>: <newName1>, <oldfield2>: <newName2>, ... } }
```

db.collection.findOneAndUpdate()

Updates a single document based on the filter and sort criteria.

db.collection.findOneAndUpdate()

findOneAndUpdate() updates the first matching document in the collection that matches the filter. The sort parameter can be used to influence which document is updated.

```
db.collection.findOneAndUpdate({ filter }, { update }, { options })
```

Options

returnDocument : string - [Optional. Starting, returnDocument is an alternative for

- returnDocument: *"before"* returns the original document.
- returnDocument: *"after"* returns the updated document.]

returnNewDocument : boolean - [Optional. When true, returns the updated document instead of the original document. Defaults to false.]

Note:

- returnNewDocument. If both options are set, returnDocument takes precedence.

db.collection.replaceOne()

Replaces a single document within the collection based on the filter.

db.collection.replaceOne()

replaceOne() replaces a single document within the collection based on the filter.

```
db.collection.replaceOne(filter, replacement, options)
```

- `db.emp.replaceOne({ ename: "saleel" }, { x: 500, y: 500 })`

db.collection.deleteOne() & db.collection.deleteMany()

Removes a single/multiple document(s) from a collection.

db.collection.deleteOne()
db.collection.deleteMany()

deleteOne() removes a **single** document from a collection. Specify an empty document { } to delete the first document returned in the collection.

deleteMany() removes **all** documents that match the filter from a collection.

```
db.collection.deleteOne({ filter })
db.collection.deleteMany({ filter })
```

- `db.emp.deleteOne({});`
- `db.emp.deleteOne({ job: "manager" });`

- `db.emp.deleteMany({});`
- `db.emp.deleteMany({ job: "manager" });`

db.collection.findOneAndDelete()

Deletes a single document based on the filter and sort criteria, returning the deleted document.

db.collection.findOneAndDelete()

findOneAndDelete() deletes the first matching document in the collection that matches the filter. The sort parameter can be used to influence which document is updated.

```
db.collection.findOneAndDelete({ filter }, [ { sort },{ projection }])
```

- `db.emp.findOneAndDelete({ job: "manager" });`
- `db.emp.findOneAndDelete({ job:  "manager" }, { sort: { sal: 1 } });`

stages

All stages are independent.

$documents dual table	$match WHERE Clause	$project SELECT clause	$addFields ADD New fields	$sample RANDOM document	$unwind PIVOT an array	$group GROUP BY clause
$match HAVING clause	$sort ORDER BY clause	$limit TOP clause	$skip	$unset REMOVE fields from output	$sortByCount	$out NEW Collection

aggregate()

In aggregation, the result of one stage is simply passed to another stage.

```
db.collection.aggregate( [ { <stage1> }, { <stage2> }, ..., { <stageN> } ] )
```

- `db.emp.aggregate([]);`

aggregation <stageOperators> and aggregation <expression>

Each sage starts with stage operator.

```
{ $<stageOperator> : { } }
```

```
{ $match : { job: 'manager' } }
{ $group : { _id : '$job' } }
```

Each aggregation expression starts with $ sign.

```
"$<field>"
```

$documents dual table	$match WHERE clause	$project SELECT clause	$addFields ADD New fields	$sample RANDOM document	$unwind PIVOT an array	$group GROUP BY clause
$match HAVING clause	$sort ORDER BY clause	$limit TOP clause	$skip	$unset REMOVE fields from output	$sortByCount	$out NEW Collection

$documents

Returns literal documents from input values.

$documents

Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

```
{ $documents: [ { doc1 }, { doc2 }, { doc3 }, ... ] }
```

```
db.aggregate([ { $documents: [ { _id: 1001, ename: 'saleel', job: 'salesman' },
                               { _id: 1002, ename: 'sharmin', job: "manager" },
                               { _id: 1003, ename: 'vrushali', job: "manager" }
                             ]
               }
           ]);
```

$match

Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

$match

Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.

```
{ $match: { <query> } }
```

- `db.emp.aggregate([ { $match: { job: "manager" } } ]);`
- `db.emp.aggregate([ { $match: { comm: { $eq: null } } } ]);`
- `db.items.aggregate([ {$match:{ tags:{$in:["A", "K"] } } } ]);`
- `db.items.aggregate([ {$match:{ tags:{$nin:["A", "K"] } } } ]);`
- `db.emp.aggregate([ { $match: { sal: { $gt: 4000 } } }, { $group: { _id: "$job", count: { $sum: "$sal" } } } ]);`
- `db.emp.aggregate([ { $match: { favouriteFruit: { $size: 1 } } } ]);`
- `db.emp.aggregate([ { $match: { 'favouriteFruit.0': "Orange"} }, { $project: { favouriteFruit: true } } ]);`

$project

Passes along the documents with the requested fields to the next stage in the pipeline. The specified fields can be existing fields from the input documents or newly computed fields.

- <field>: <1 or true> - Specifies the inclusion of a field. Non-zero integers are also treated as true.
- _id: <0 or false> - Specifies the suppression of the _id field.
- <field>: <expression> - Adds a new field or resets the value of an existing field.
- <field>: <0 or false> - Specifies the exclusion of a field.

$project

Passes along the documents with the requested fields to the next stage in the pipeline. The specified fields can be existing fields from the input documents or newly computed fields.

```
{ $project: { <specification(s)> } }
```

- `db.emp.aggregate([ { $project: { ename: true } } ])`
- `db.emp.aggregate([ { $project: { "Employee Name" : "$ename" } } ]) // alias name`
- `db.emp.aggregate([ { $project: { _id: false, sal: true, comm: true } } ])`
- `db.emp.aggregate([ { $project: { sal: true, sm: { $sum: "$sal" } } } ])`
- `db.emp.aggregate([ { $project: { xx: { $max: [ "$sal", "$comm" ] } } } ])`
- `db.emp.aggregate([ { $project: { favouriteFruit: { $size: "$favouriteFruit" }}}])`

$unset

Removes/excludes fields from documents in output.

$unset

Removes field(s) from the output. **Will not delete the field(s) from the saved document.**

```
{ $unset: "<field>" }
{ $unset: [ "<field1>", "<field2>", ... ] }
{ $unset: "<field.nestedfield>" }
{ $unset: [ "<field1.nestedfield>", ...] }
```

- `db.emp.aggregate([ { $unset: "ename" } ])`
- `db.emp.aggregate([ { $unset: "address.building" } ])`

$literal

Returns a value without parsing. Use for values that the aggregation pipeline may interpret as an expression.

$literal

To avoid treating numeric or boolean literals as projection flags, use the $literal expression to wrap the numeric or boolean literals.

```
{ $literal: <value> }
```

- ```
  db.emp.aggregate([{ $project: { _id: 0, sal: 1, staticValue: {
  $literal: 1001 }, staticString: { $literal: "Saleel Bagde" }}}])
  ```
  ```

Remember:

If the name of the ***new field is the same as an existing field name*** (including _id), $addFields overwrites the existing value of that field with the value of the specified expression.

$addFields or $set

```
Adds new fields to documents. $addFields or $set outputs documents that
contain all existing fields from the input documents and newly added fields.
```

Note:

- $addFields appends new fields to existing documents. You can include one or more $addFields stages in an aggregation operation.
- The $addFields stage is equivalent to a $project stage that explicitly specifies all existing fields in the input documents and adds the new fields.
- If the $addFields stage is specified after $project stage, then all new fields in $addFields stage will be automatically get added in output document, but if it given before $project stage then all the new fields must be exclusively given in the $project stage.

$addFields or $set

TODO

```
{ $addFields: { <newField>: <expression>, ... } }
{ $set: { <newField>: <expression>, ... } }
```

- `db.emp.aggregate([ { $addFields: { NewSalary: 1450 } } ])`
- `db.emp.aggregate([ { $set: { NewSalary: 1450 } } ])`

- `db.emp.aggregate([{ $project: { _id: 0,  ename: 1,  salary: "$sal",  commission: "$comm" } }, { $addFields: { "Gross Salary": { $add: [ "$salary", "$commission" ] } } }])`
- `db.emp.aggregate([{ $project: { _id: false, ename: true, sal: true, x: { $literal: 100 }, y: { $literal: 200 } } }, { $addFields: { z: { $add: [ '$x', '$y' ] } } }])`

$expr

$expr can build query expressions that compare fields from the same document in a $match stage.

$expr

TODO

```
{ $expr: { <operator>: ['$<argument1>' | value1, '$<argument2>' | value2 ] } }

$expr: { $eq: ['$sal', 3000 ] } }
$expr: { $gt: ['$<argument1>' | value1, '$<argument2>' | value2 ] } }
$expr: { $gte: ['$<argument1>' | value1, '$<argument2>' | value2 ] } }
$expr: { $lt: ['$<argument1>' | value1, '$<argument2>' | value2 ] } }
$expr: { $lte: ['$<argument1>' | value1, '$<argument2>' | value2 ] } }
```

- `db.emp.aggregate([ { $project: { _id: false, ename: true, sal: true }}, { $match: { $expr: { $gt: [ '$sal', 5000 ] } } } ]);`
- `db.emp.aggregate([ { $project: { _id: false, ename: true, sal: true, x: { $literal: 10 }, y: { $literal: 10 } } }, { $match: { $expr: { $eq: ['$x', '$y'] } } } ]);`

$sample

Randomly selects the specified number of documents from its input.

$sample

Randomly selects the specified number of documents from its input.

```
{ $sample: { size: <positive integer N> } }
```

- `db.emp.aggregate([ { $sample: { size: 2 } } ])`

$unwind

Deconstructs an array field from the input documents to output a document for each element. Each output document is the input document with the value of the array field replaced by the element.

$unwind

Deconstructs an array field from the input documents to output a document for each element.

```
{ $unwind: { path: '$<field path>', includeArrayIndex: <string> }
```

- ```
  db.emp.aggregate([ { $project: { favouriteColor: true } }, { $unwind: "$favouriteColor" } ])
  ```
  ```

arithmetic expression operators

- `db.movies.aggregate([ { $project: { _id: false, "Title": true, R: { $round: { $multiply: [ {$rand: {} }, 800 ] } } } } ])`

arithmetic expression operators

Arithmetic expressions perform mathematic operations on numbers. Some arithmetic expressions can also support date arithmetic.

Arithmetic expressions
x: { $abs: '$<number>' }
x: { $add: ['$<expression1>', '$<expression2>', ...] }
x: { $subtract: ['$<expression1>', '$<expression2>'] }
x: { $multiply: ['$<expression1>', '$<expression2>', ...] }
x: { $divide: ['$<expression1>', '$<expression2>'] }
x: { $mod: ['$<expression1>', '$<expression2>'] }
x: { $round : [<number>, <place>] }
x: { $trunc: '$<number>' } x: {{ $round : [<number>, <place>] }
x: { $rand:{} }

- db.emp.aggregate([{ $project : { op: { $trunc: "$sal" } } }])
- db.emp.aggregate([{ $project: { sal: true, op : { $add: ["$sal", 1000] } } }])
- db.emp.aggregate([{ $project: { x: { $rand: {} } } }])

$ifNull(), $toUpper, $toLower, $concat, . . .

$ifNull(), $toUpper(), $toLower(), $concat

Evaluates an expression and returns the value of the expression if the expression evaluates to a non-null value. If the expression evaluates to a null value, including instances of undefined values or missing fields, returns the value of the replacement expression.

```
x: { $ifNull:[ '$<expression>', <replacement-expression-if-null> ] }
x: { $toUpper: '$<expression>' }
x: { $toLower: '$<expression>' }
x: { $strLenCP: '$<expression>' } // string expression
x: { $concat:[ '$<expression1>', '$<expression2>', ... ] }
x: { $substr: [ <string>, <start>, <length> ] }
x: { $size: '$<expression>' }
x: { $arrayElemAt: [ '$<array>', <idx> ] }  // -1 will get last element from array
x: { $split: [ '$<expression>', '<delimiter>' ] }
x: { $slice: [ '$<array>' , <position>,  <n> ] } ] }
```

$ifNull(), $toUpper(), $toLower(), $concat

- db.emp.aggregate([{ $project: { comm : { $ifNull: ["$comm", "NA"] } } }])
- db.emp.aggregate([{ $project: { "Gross Salary": { $add: ["$sal", { $ifNull: ["$comm", 0] }] } } }])
- db.emp.aggregate([{ $project: { ename : { $toUpper: "$ename" } } }])
- db.emp.aggregate([{ $project: { ename : { $toLower: "$ename" } } }])
- db.movies.aggregate([{ $project: { movie_title: true, movie_length: { $strLenCP: { $toString: "$movie_title" } } } }])
- db.emp.aggregate([{ $project: { ename : { $concat: ["$ename", "$job"] } } }])
- db.emp.aggregate([{ $project: { favouriteFruit: { $size: "$favouriteFruit" } } }])
- db.emp.aggregate([{ $project: { op: { $arrayElemAt: ["$favouriteFruit", 1] } } }])
- db.movies.aggregate([{ $project: { _id: false, title: "$movie_title", genres: true, x: { $split: ["$genres", "|"] } } }, { $limit: 4 }])

- db.emp.aggregate([{ $project: { x: { $arrayElemAt: ["$favouriteFruit", 1] } } }, { $match: { x: 'Orange' } }])

$toString(), $toInt(), $toDoube(), . . .

TODO

```
x: { $toString: '$<expression>' }
```

```
x: { $toInt: '$<expression>' }
```

```
x: { $toDouble: '$<expression>' }
```

```
x: { $toLong: '$<expression>' }
```

```
x: { $toBool: '$<expression>' }
```

$type(), $isNumber(), . . .

TODO

`x: { $type: '<$expression>' }`	
`x: { $isNumber: '<expression>' }`	`// true if the expression is number.` `// false if the expression is any other BSON type, null, or a missing field.`

- `db.movies.aggregate([{ $addFields:{ x: { $type: "$movie_title" }}}]);`
- `db.movies.aggregate([{ $addFields:{ x: { $isNumber: "$movie_title" }}}]);`
- `db.movies.aggregate([{ $addFields: { x: { $type: "$movie_title" } } }, { $match: { x: "int" } }, { $project: { _id: false, movie_title: true } }]);`
- `db.movies.aggregate([{ $project: { _id: false, x: { $isNumber: '$movie_title' }, movie_title: true, }}, { $match: { x: true } }]);`

$first(), $last(), $range(), $allElementsTrue(), $anyElementTrue(), ...

```
x: { $replaceOne: { input: <expression>, find: <expression>,
                    replacement: <expression> } }
x: { $replaceAll: { input: <expression>, find: <expression>,
                    replacement: <expression> } }
```

```
x: { $first: '$<expression>' }
```

```
x: { $last: '$<expression>' }
```

```
x: { $range: [ <start>, '$<expression>', <non-zero step> ] }
```

```
x: { $allElementsTrue: '$<expression>' }  //in list → [ ]
```

```
x: { $anyElementTrue: '$<expression>' }   //in list → [ ]
```

```
x: { $cond: { if: <boolean-expression>, then: <true-case>, else: <false-case> } }
```

- `db.movies.aggregate([{$project:{_id: false, "Title": true, x:{ $replaceAll: { input: { $toString: "$Title" }, find: "T", replacement: "@" }}}}])`

$first(), $last(), $range(), $allElementsTrue(), $anyElementTrue(), …

- `db.emp.aggregate([ { $project: { _id: false, x: { $first: "$cards" } } } ])`
- `db.emp.aggregate([ { $project: { _id: false, x: { $last: "$cards" } } } ])`
- `db.emp.aggregate([ { $project: { "address.coord": true, x: { $last: "$address.coord" } } } ])`
- `db.movies.aggregate([ { $project: { x: { $range: [ 0, { $ifNull: [ "$duration", 0 ] }, 30 ] }} } ])`
- `db.survey.aggregate([ { $project:{ responses: true, x: { $allElementsTrue: "$responses" } } } ])`
- `db.survey.aggregate([ { $project:{ responses: true, x: { $anyElementTrue: "$responses" } } } ])`
- `db.movies.aggregate([ { $match: { movie_title: /Horse/ }}, { $project: { _id: true, movie_title: true, duration: true, x: { $range: [ 0, "$duration", 60 ] } } }])`
- `db.movies.aggregate([ { $project:{ duration: true, x: { $cond: { if: { $eq: [ "$duration", 100 ]}, then: "$duration", else: "More" } } } } ])`
- `db.emp.aggregate([{ $project: { _id: false, ename: true, sal: true, output: { $cond: { if: { $gte: ['$sal', 3000] }, then: { $add: ['$sal', 1000] }, else: '$sal' } } } }])`

date operators

date operators

TODO

Date expressions
`x: { $dayOfMonth: '$<dateExpression>' }`
`x: { $dayOfWeek: '$<dateExpression>' }`
`x: { $dayOfYear: '$<dateExpression>' }`
`x: { $month: '$<dateExpression>' }`
`x: { $week: '$<dateExpression>' }`
`x: { $year: '$<dateExpression>' }`

- `db.emp.aggregate([ { $project: { Day: { $dayOfMonth: "$hiredate" } } } ])`
- `db.emp.aggregate([ { $project: { Month: { $month: "$hiredate" } } } ])`

```
const x = [ 'January', 'February', 'March', 'April', 'May', 'June', 'July', ... ]
```

$group

Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping. The output documents contain an _id field which contains the distinct group by key. The output documents can also contain computed fields that hold the values of some accumulator expression grouped by the $group’s _id field. $group does not order its output documents.

$group

The _id field is mandatory; however, you can specify an _id value of null to calculate accumulated values for all the input documents as a whole.

```
{ $group: { _id: '$<expression>', <field1>: { <accumulator1> :
<expression1> }, ... } }
```

Accumulator Operator - [$group and $project stage]
x: { $avg: '$<expression>' }
x: { $sum: '$<expression>' }
x: { $min: '$<expression>' } x: { $min: ['$<expression>', '$<expression>' ...] }
x: { $max: '$<expression>' } x: { $max: ['$<expression>', '$<expression>' ...] }
x: { $count: {} }

- db.emp.aggregate([{ $group: { _id: null, count: { $sum: 1} } }])
- db.emp.aggregate([{ $group: { _id: null, total: { $sum: "$sal" } } }])
- db.emp.aggregate([{ $group: { _id: "$job", count: { $sum: 1 } } }])

$group on multiple fields

$ group

The _id field is mandatory; however, you can specify an _id value of null to calculate accumulated values for all the input documents as a whole.

```
{ $group: { _id: { <field1>: '$<expression>', ... }, <field1>: {
<accumulator1> : '$<expression1'> }, ... } }
```

- `db.emp.aggregate([ { $group: { _id: { job: "$job", deptno: "$deptno" }, count : { $sum: 1 } } } ])`

$sort

Sorts all input documents and returns them to the pipeline in sorted order.

$sort

TODO

```
{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
```

- `db.emp.aggregate([ { $sort: { ename: 1 } } ])`

Value	Description
1	Sort ascending.
-1	Sort descending.

$sortByCount

The documents are sorted by count in descending order.

$sortByCount

TODO

```
{ $sortByCount: $<expression> }
```

- `db.emp.aggregate([ { $sortByCount : "$job" } ])`

$limit

Limits the number of documents passed to the next stage in the pipeline.

$limit

TODO

```
{ $limit: <positive integer> }
```

- `db.emp.aggregate([ { $limit: 2 } ])`
- `db.emp.aggregate([ { $project: { total: { $add: [ "$sal", "$comm" ] } } }, { $limit: 2 } ])`

$skip

Skips over the specified number of documents that pass into the stage and passes the remaining documents to the next stage in the pipeline.

$skip

TODO

```
{ $skip: <positive integer> }
```

- `db.emp.aggregate([ { $skip: 2 } ])`

$count

Counts the number of documents in a collection or a view.

$count

TODO

```
{ $count: "Field-name" }
```

- `db.emp.aggregate([ { $count: "ename" } ])`

$out

Takes the documents returned by the aggregation pipeline and writes them to a specified collection.

$out

Takes the documents returned by the aggregation pipeline and writes them to a specified collection.

```
{ $out: { db: "<output-db>", coll: "<output-collection>" } }
```

- `db.emp.aggregate([ { $project: { movie_title: true, director: true, duration: true } }, { $out: "movieList" } ])`
- `db.emp.aggregate([ { $project: { movie_title: true, director: true, duration: true } }, { $out: { db: "new-db-name"}, coll: "movieList" } ])`

Note:

- The $out stage must be the last stage in the pipeline.

db.createView()

```
Views are read-only; write operations on views will error.
```

Note:

- You must create views in the same database as the source collection.
- A view definition pipeline cannot include the $out or the $merge stage. This restriction also applies to embedded pipelines, such as pipelines used in $lookup or $facet stages.
- You cannot rename a view once it is created.
- Views are read-only; write operations on views will error.

db.createView()

Views are read-only; write operations on views will error. You cannot rename views.

```
db.createView("<viewName>", "<source>", [<pipeline>],
  {
    "collation" : { <collation> }
  }
)
```

- `db.createView("empView", "emp", [ { $match: { job: "manager" } } ])`
- `db.createView("employeeView", "emp", [ { $project: { _id: false, ename: true, address: true, job: true, salary: "$sal" } } ])`

- `db.empView.aggregate([ { $match: { empid: 1029 } } ])`
- `db.empView.drop()`

db.createIndex()

```
todo
```

Note:

- todo

db.createIndex()

todo

```
db.collection.createIndex( <field>, <options> )
```

Value	Description
1	specifies an index that orders items in ascending order.
-1	specifies an index that orders items in descending order..

- `db.createView("empView", "emp",  [ { $match: { job: "manager" } } ])`
- `db.empView.aggregate([ { $match: { empid: 1029 } } ])`

$setUnion / $setIntersection / $setDifference

- Takes two or more arrays and returns an array containing the elements that appear in any input array.
- Takes two or more arrays and returns an array that contains the elements that appear in every input array.
- Takes two sets and returns an array containing the elements that only exist in the first set.

$setUnion

TODO

```
{ $setUnion: [ <expression1>, <expression2> ] }
```

$setIntersection

TODO

```
{ $setIntersection: [ <expression1>, <expression2> ] }
```

$setDifference

TODO

```
{ $setDifference: [ <expression1>, <expression2> ] }
```

$rank / $denseRank / $documentNumber

todo

$rank / $denseRank / $documentNumber

Returns the document position.

```
{ $setWindowFields: {
      partitionBy: "$Field", </optional>
      sortBy: { field: -1/1},
      output: { x: { $rank: {} } } OR
      output: { x: { $denseRank: {} } } OR
      output: { x: { $documentNumber: {} } }
  }
}
```

```
db.orders.aggregate([{ $setWindowFields: {
   sortBy: { _id: 1 },
   output: { x: { $documentNumber: {} } }
}}])
```

Note:

- $rank/$denseRank/$documentNumber does not accept any parameters.
- $rank/$denseRank/$documentNumber is only available in the $setWindowFields stage.
- partitionBy: "$Field" is optional property for $setWindowFields stage.

$rank / $denseRank / $documentNumber

TODO

- ```
  db.orders.aggregate([ { $setWindowFields:
  {
     sortBy: { _id: 1 },
     output: { x: { $documentNumber: {} } }
  }
  } }]);
  ```
- ```
  db.movies.aggregate([ { $setWindowFields:
  {
     sortBy: { _id: 1 },
     output: { x: { $documentNumber: {} } }
  },
  { $project: { _id: false, title: '$movie_title', x: true } }, { $match: { x: 3 }
  }]);
  ```

```
localField: <datatype of field from parent collection must be same>,
foreignField: <datatype of field from child collection must be same>,
```

$lookup

To perform an equality match between a field from the input documents with a field from the documents of the “joined” collection

$lookup

```
{
   $lookup:
     {
       from: <foreign collection>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: < text >,
      pipeline: [ { $project }, { $match } ]
     }
},
{
   $lookup:
     {
       from: <foreign collection>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: < text >,
      pipeline: [ { $project }, { $match } ]
     }
}
```

$lookup

```
db.order.aggregate([{
    $lookup: {
        from: 'orderdetails',
        localField: '_id',
        foreignField: 'orderID',
        as: 'OrderDetails',
        pipeline: [{
            $project: {
                _id: false,
                product: true,
                qty: true,
                rate: true,
                Total: {
                    $multiply: ['$qty', '$rate']
                }
            }
        }]
    }
}])
```

$lookup

e.g. TODO

- ```
  db.orders.insertMany([
  { "orderNo" : 1, "orderDay" : "Mon" },
  { "orderNo" : 2, "orderDay" : "Sat" },
  { "orderNo" : 3 } ])
  ```
- ```
  db.orderdetails.insertMany([
  {"orderNo" : 1, "item" : "maggi", "price" : 40, "quantity" : 7 },
  {"orderNo" : 1, "item" : "butter", "price" : 125, "quantity" : 12 },
  {"orderNo" : 1, "item" : "cheese", "price" : 225, "quantity" : 12 },
  {"orderNo" : 2, "item" : "coffee", "price" : 75, "quantity" : 1 },
  {"orderNo" : 2, "item" : "tea", "price" : 175, "quantity" : 3 },
  {"orderNo" : 2, "item" : "nuts", "price" : 375, "quantity" : 2 } ])
  ```

```
> db.orders.drop()
> db.orderdetails.drop()
> db.orders.find()
> db.orderdetails.find()
```

$lookup

e.g. TODO

```
db.orders.aggregate ([
  { $lookup: {
      from : "orderdetails",
      localField : "orderNo",
      foreignField : "orderNo",
      as : "Order Details" }
  }]).forEach(printjson);
```

```
db.orders.aggregate([
  { $lookup: {
      from: "orderdetails",
      localField: "orderNo",
      foreignField: "orderNo",
      as: "Order Details" } },
  { $project:{ _id: false, "Order Details._id": false}}]).forEach(printjson);
```

Database Security and Authentication

Authentication is the process of verifying the identity of a client. When access control, i.e. authorization, is enabled, MongoDB requires all clients to authenticate themselves in order to determine their access. Although authentication and authorization are closely connected, authentication is distinct from authorization. Authentication verifies the identity of a user; authorization determines the verified user’s access to resources and operations.

db.getUser() / db.getUsers()

db.getUser() / db.getUsers()

Returns user information for a specified user.

```
db.getUser(username, args)
```

- `db.getUser("user01");`

Returns information for all the users in the database.

```
db.getUsers()
```

- `db.getUsers();`

db.createUser

db.createUser

Creates a new user for the database on which the method is run. db.createUser() returns a duplicate user error if the user already exists on the database.

```
db.createUser(user, [writeConcern])
```

-
```
db.createUser (
{
    user: "user01",
    pwd: "user01",
    roles:[{role: "userAdmin" , db: "db1"},
           {role: "readWrite", db: "db1"}],
    authenticationRestrictions: [ {
          clientSource: [ "192.168.100.26", "192.168.100.20", "192.168.100.120",
                          "192.168.100.83"],
    serverAddress: ["192.168.100.20"]
     } ]
});
```

db.grantRolesToUser /
db.revokeRolesFromUser

db.grantRolesToUser
db.revokeRolesFromUser

TODO

```
db.grantRolesToUser( "<username>", [ <roles> ], { <writeConcern> }
```

- ```
  db.grantRolesToUser( "user01",
      [
        { role: "read", db: "db1" }
      ]
  );
  ```

```
db.revokeRolesFromUser("<username>", [<roles>], {<writeConcern>} )
```

- ```
  db.revokeRolesFromUser( "user01",
      [
        { role: "read", db: "db1" }
      ]
  );
  ```

roles

The role provides the following actions on those collections

```
1. Read :- [dbStats, find, listIndexes, listCollections, etc...]

2. readWrite :- [collStats, convertToCapped, createCollection,
   dbHash, dbStats, dropCollection, createIndex, dropIndex, find,
   insert, killCursors, listIndexes, listCollections, remove,
   renameCollectionSameDB, update]

3. userAdmin :- [TODO]

4. readAnyDatabase :- [TODO]

5. readWriteAnyDatabase :- [TODO]
```

db.dropAllUser() / db.dropUser()

db.dropUser() / db.dropAllUsers()

Removes the user from the current database.

```
db.dropUser(username, writeConcern)
```

- `db.dropUser("user01");`

Removes all users from the current database.

```
db.dropAllUsers([writeConcern])
```

- `db.dropAllUser();`

IMP

1. Think about how multiplication can be done without actually multiplying

$7 * 4 = 28$

$7 + 7 + 7 + 7 = 28$

$5 * 6 = 30$

$5 + 5 + 5 + 5 + 5 + 5 = 30$

2. Square

$1^2 = (1) = 1$

$2^2 = (1 + 3) = 4$

$3^2 = (1 + 3 + 5) = 9$

$4^2 = (1 + 3 + 5 + 7) = 16$

IMP

Camel Case: Second and subsequent words are capitalized, to make word boundaries easier to see.
Example: numberOfCollegeGraduates

Pascal Case: Identical to Camel Case, except the first word is also capitalized.
Example: NumberOfCollegeGraduates

Snake Case: Words are separated by underscores.
Example: number_of_college_graduates

IMP

Types of Machine Learning
(Supervised, Un-Supervised, Reinforcement)

Supervised Learning

Supervised machine learning has two key components: first is **input data** and second corresponding **output labels**. The goal is to build a model that can learn from this labeled data to make predictions or classifications on new, unseen data.

The labeled data consists of input features and the corresponding output labels.

Supervised Learning

Medical Diagnosis: Supervised algorithms are also used in the medical field for diagnosis purposes. It is done by using medical images and past labelled data with labels for disease conditions. With such a process, the machine can identify a disease for the new patients.

Fraud Detection - Supervised Learning algorithms are used for identifying fraud transactions, fraud customers, etc. It is done by using historic data to identify the patterns that can lead to possible fraud.

Spam detection - In spam detection & filtering algorithms are used. These algorithms classify an email as spam or not spam. The spam emails are sent to the spam folder.

Unsupervised Learning

Unsupervised learning is a type of machine learning where the algorithm learns from unlabeled data without any predefined outputs or target variables. The unsupervised learning finds patterns, similarities, or groupings within the data to get insights and make data-driven decisions.

“Accept your past without regret, handle our present with confidence and face your future without fear.“

A.P.J. Abdul Kalam

array operations in mongodb examples

array operation on student collection

- `db.student.drop();`

```
db.student.insertMany([{
    _id: 1,
    name: "saleel",
    hobbies: [],
    qualifications: [{
        name: "10th", grade: "A", year: 1980
    }]
}, {
    _id: 2,
    name: "sharmin",
    hobbies: [ "painting" ],
    qualifications: [{
        name: "10th", grade: "A+", year: 2017
    }, {
        name: "12th", grade: "A+", year: 2019
    }]
}]);
```

operation on student collection

- Add new student with following fields (_id, name, hobbies, city, and state) in the student collection.

```
db.student.insertOne( { _id: 3, name: "ruhan", hobbies: [ "painting" ], city: "pune", state: "MH" })
```

- Add a grade, year, school, and fees of 10th standard in the qualification field for student _id:3.

```
db.student.updateOne( { _id: 3 }, { $set: { qualification: [ { name: "10th", grade: "B+", year: 2017, school: "mit", fees: 19000 } ] } })
```

- Add a grade, year, school, and fees of 12th standard in the qualification field for student _id:3.

```
db.student.updateOne( { _id: 3 }, { $push: { qualification: [ { name: "12th", grade: "A", year: 2019, school: "mit", fees: 22000 } ] } })
```

operation on student collection

- This query is related to the previous example. In the previous example the 12th standard details is added as an array in the qualification field.

See the output of previous example:- db.student.aggregate()

- Delete the 12th standard qualification from student collection whose _id:3

```
db.student.updateOne( { _id: 3 }, { $pop: { "qualification": 1 } })
```

- Add a grade, year, school, and fees of 12th standard in the qualification field whose student _id:3.

```
db.student.updateOne( { _id: 3 }, { $push: { qualification: { name: "12th", grade: "A", year: 2019, school: "mit", fees: 22000 } } })
```

- Add emailID field to 10th std. and 12th std. school in the student collection whose student _id:3.

```
db.student.updateOne( { _id: 3 }, { $set: { "qualification.0.emailID": "mit@gmail.com", "qualification.1.emailID": "meer@gmail.com" } })
```

array operation on student collection

- Display the following fields from student collection { _id, student name, hobbies, and count the number of hobbies for every student and show the count in the field hobbyCount }.

```
db.student.aggregate([ { $project: { _id: true, name: true, hobbies: true,
hobbyCount: { $size:"$hobbies" } } } ])
```

- Display the following fields from student collection { _id, student name, city, state, and qualification details of only 10th std.

```
db.student.aggregate([ { $project: { _id: true, name: true, city: true, state: true,
"10th details": { $arrayElemAt: [ '$qualification', 0 ] } } } ])
```

array operation on student collection

array operation on student collection

array operation on student collection

- Add city field for student _id:1

```
db.student.updateOne( {_id: 1}, { $set: { city: "pune" } } )
```

- Rename qualifications field to qualification for all the documents.

```
db.student.updateMany( {}, { $rename: { "qualifications": "qualification" } } )
```

- Add new school field to zeroth element of qualification field for student _id:2.

```
db.student.updateOne( {_id: 2}, { $set: { "qualification.0.school": "rosary" } } )
```

- Add new school field and empty array for emailID field to zeroth element of qualification field for student _id:1.

```
db.student.updateOne( {_id: 1}, { $set: { "qualification.0.school": "MIT",
"qualification.0.emailID": [ ] } } )
```

- Add new value to an empty array for emailID field

```
db.student.updateOne( {_id: 1}, { $set: { "qualification.0.emailID":
"ruhan@gmail.com" } } )
```

array operation on student collection

- Add new value to an empty array for emailID field

```
db.student.updateOne( {_id: 1}, { $set: { "qualification.0.emailID": [
"sharmin@gmail.com" ] } } )
```

- Add new value to an empty array for emailID field to zeroth element of qualification field for student _id:1.

```
db.student.updateOne( {_id: 1}, { $push: { "qualification.0.emailID":
"saleel@gmail.com" } } )
```

- Add new value to an empty array for emailID field to zeroth element of qualification field for student _id:1.

```
db.student.updateOne( {_id: 1}, { $push: { "qualification.0.emailID":
"sharmin@gmail.com" } } )
```

- Display student name and his 12th qualification details for all students.

```
db.student.aggregate([ { $project: { _id: 0, name: true, 'qualification details': {
$arrayElemAt: [ '$qualification', 1 ] } } } ])
```

array operation on student collection

- Add 12th qualification details for student whose _id:1.

```
db.student.updateOne( {_id: 1}, { $push: { qualification: { name: "12th", grade: 'B+', year: 1982 } } } )
```

- Add school field in qualification field for both elements.

```
db.student.updateOne( {_id: 1}, { $set: { 'qualification.0.school': "navrachana", 'qualification.1.school': "bhavance" } } )
```

- Add emailID for each school for student whose _id:1.

```
db.student.updateOne( {_id: 1}, { $set: { 'qualification.0.emailID': [ "nvrachana@gmail.com" ], 'qualification.1.emailID': [ "bhavance@gmail.com" ] } } )
```

- Add fees field with value 7000 for all student who are in 10th std.

```
db.student.updateMany( { 'qualification.name': "10th" }, { $set: { 'qualification.0.fees': 7000 } } )
```

array operation on student collection

- Add football and cricket hobbies whose student _id:1.

```
db.student.updateOne( {_id: 1}, { $push: { hobbies: { $each: [ "football", "cricket" ] } } } )
```

- Increase the fees by Rs. 2000 of all student who are in 10th std.

```
db.student.updateMany( { 'qualification.name': "10th" }, { $inc: { 'qualification.0.fees': 2000 } } )
```

- Add new student.

```
db.student.insertOne( {_id: 3, name: "sangita", hobbies: [], city: "baroda", state: "GJ" } )
```

- Change the name to 'ruhan' whose _id:3.

```
db.student.updateOne( {_id: 3}, { $set: { name: "ruhan" } } )
```

- Display all students where name starts with the letter 'r'.

```
db.student.aggregate( [ { $match: { name: /^r/ } } ] )
```

array operation on student collection

- Print student name, hobbies and the first hobby for all the student (using $first).

```
db.student.aggregate([ { $project: { _id: false, name: true, hobbies: true,
firstHobby: { $first: "$hobbies" } } } ])
```

- Print student name, hobbies and the first hobby for all the student (using arrayElemAt).

```
db.student.aggregate([ { $project: { _id: false, name: true, hobbies: true,
firstHobby: { $arrayElemAt: [ "$hobbies", 0 ] } } } ])
```

- Print student name, hobbies and the last hobby for all the student (using $last).

```
db.student.aggregate([ { $project: { _id: false, name: true, hobbies: true,
lasttHobby: { $last: "$hobbies" } } } ])
```

- Print student name, hobbies and the last hobby for all the student (using arrayElemAt).

```
db.student.aggregate([ { $project: { _id: false, name: true, hobbies: true,
lastHobby: { $arrayElemAt: [ "$hobbies", -1 ] } } } ])
```

array operation on student collection

- todo

- todo

- todo

- todo

- todo

some operations on emp collection

operation on emp collection

- Print all documents from emp collection.

```
db.emp.aggregate()
```

- Print employee name and the zeroth element of his favourite color.

```
db.emp.aggregate([ { $project: { _id: false, ename: true, color: { $arrayElemAt: [ '$color', 0 ] } } } ])
```

- Print all fields {_id, ename, gender, address, and isDocActive } from emp collection whose gender is ‘male’ and isDocActive is true for all the employees.

```
db.emp.aggregate([ { $match: { gender: "male", isDocActive: true } }, { $project: { _id: false, ename: true, gender: true, address: true, isDocActive: true } } ])
```

- Print entire document of the 7th employee.)

```
db.emp.find()[6]; (Note:- This will not work in mongosh shell)
```

- Count total documents of emp collection.

```
db.emp.countDocuments({})
```

operation on emp collection

- Count all isDocActive document.

```
db.emp.countDocuments({isDocActive: true})
```

- Print the last document from emp collection.

```
db.emp.find()[ db.emp.countDocuments({}) -1 ]
```

- Print the sum of salary for all employees.

```
db.emp.aggregate([ { $group: { _id: null, totalSalary: { $sum: "$sal" } } } ])
```

- Print random 3 {employee name, address and salary} from emp collection.

```
db.emp.aggregate([ { $sample: {size: 3 } } ,{ $project: { _id: false, ename: true,
address: true, sal: true } } ])
```

- Print first 7 employee name, sal, and comm by changing the heading as Employee Name, Salary and Commission

```
db.emp.aggregate([ { $project: { _id: false, "Employee Name": "$ename", Salary:
"sal", Commission: "$comm" } }, { $limit: 7 } ])
```

operation on emp collection

- Print highest paid salary.

```
db.emp.aggregate([ { $group: { _id: null, "Maximum Salary" : { $max: '$sal' } } }, { $project: { _id: false } } ])
```

- Print employee name, his salary and also give documentNumber to every document.

```
db.emp.aggregate([ { $setWindowFields: { sortBy: { sal: -1 }, output: { documentNumber: { $documentNumber:  {} } } } } }, { $project: { _id: false, ename: true, sal: true, documentNumber: true } }])
```

- Give the ranking to all document in descending order on salary field

```
db.emp.aggregate([ { $setWindowFields: { sortBy: { sal: -1 }, output: { documentNumber: { $documentNumber:  {} } } } } }, { $project: { _id: false, ename: true, sal: true, documentNumber: true } }])
```

- Print the document who is getting 3rd highest salary.

```
db.emp.aggregate([ { $setWindowFields: { sortBy: { sal: -1 }, output: { denseRank: { $denseRank: {} } } } } }, { $match: { denseRank: 3 } }, { $project: { _id: false, ename: true, sal: true, denseRank: true } } ])
```

operation on emp collection

- Print the first element from cards array field from emp collection.

```
db.emp.aggregate([ { $match: { _id: ObjectId("62bfd2ff6a923392ce172cb8") } },  {
$project: { _id: true, ename: true,  x: { $first: "$cards" } } } ]);
```

- Print the last element from cards array field from emp collection.

```
db.emp.aggregate([ { $match: { _id: ObjectId("62bfd2ff6a923392ce172cb8") } },  {
$project: { _id: true, ename: true,  x: { $last: "$cards" } } } ]);
```

- todo

- todo

- todo

operation on emp collection

- todo

- todo

- todo

- todo

- todo

some operations on movie collection

operation on movies collection

- Import movies.csv file in

```
mongoimport --host=192.168.1.21 --port=27017 --db=assignment --collection=movies --type=csv --file=d:\movie.csv --headerline --useArrayIndexFields
```

- Print movie_title, director, relese date, and genres whose director name starts with the letter ‘D’.

```
db.movies.aggregate([ { $match: { director: /^D/ }  }, { $project: { movie_title: true, director: true, relese: true, genres: true } } ])
```

- Print movie_title, director, genres, color, week1, week2, week3, week4, and create Total virtual field that print the addition of week1 + week2 + week3 + week4, round the Total to 3 decimal places.

```
db.movies.aggregate([ { $project: { movie_title: true, director: true, genres: true, color: true, week1: true, week2: true, week3: true, week4: true } }, { $addFields: { Total: { $round: [ { $add: [ '$week1', '$week2', '$week3', '$week4' ] }, 3 ] } } } ])
```

operation on movies collection

- Print movie_title, director, language, genres, and color of all ‘English’ language movies.

```
db.movies.aggregate([ { $match: { language: "English" } }, { $project: {
movie_title: true, director: true, language: true, genres: true, color: true } } ])
```

- Count ‘Hindi’ movies.

```
db.movies.aggregate([ { $match: { language: "Hindi" } }, { $count: "Total Hindi
Movies" } ])
```

- Print color, director, duration, genres, movie_title, title_year, productionhouses where genres is ‘Horror’.

```
db.movies.aggregate([ { $match: { genres: /Horror/ } }, { $project: { color: true,
director: true, duration: true, genres: true, movie_title: true, title_year: true,
productionhouses: true } } ])
```

- Create a copy of emp collection from primaryDB collection to assignment collection

```
db.getSiblingDB('primaryDB').getCollection('emp').aggregate([{ $out: { db:
"assignment", coll: "emp" } }])
```

operation on movies collection

- Count languages wise movies.

```
db.movies.aggregate([ { $group: { _id: '$language' , count :  { $sum: 1 } } } ])
```

- Print movie_title, director, genres, actor_1_name, actor_2_name, actor_3_name, budget, gross, stars and add new virtual field Rating and compute total sum of stars.

```
db.movies.aggregate([ { $project: { movie_title: true, director: true, genres: true, actor_1_name: true, actor_2_name: true, actor_3_name: true, budget: true, gross: true, stars: true } }, { $addFields: { Rating: { $sum: ['$stars'] } } } ])
```

- Split genres in array and print the first element from the array.

```
db.movies.aggregate([ { $project: { _id: false, title: "$movie_title", genres: true, x: { $split: [ "$genres", "|" ] } } }, { $addFields: { y: { $arrayElemAt: [ "$x", 0 ] } } } ])
```

- Print actor one and count how many characters are there in their name.

```
db.movies.aggregate([ { $project: { "actor name and length": { $concat: [ "$actor_1_name", " ---> ", { $toString: { $strLenCP: "$actor_1_name" } } ] } } } ])
```

operation on movies collection

- Count movies which is directed by director whose name starts with a letter ‘B’.

```
db.movies.aggregate([ { $match: { director: /^B/ } }, { $group: { _id: null, count: { $sum: 1 } } } ]);
```

- Print movie title and rating count.

```
db.movies.aggregate([ { $project: { _id: false, title: "$movie_title", rating : { $concat: [ { $toString: { $sum: '$stars' } } , ' star' ] } } } ])
```

- Print movie list whose rating is = ‘5 star’

```
db.movies.aggregate([ { $project: { _id: false, title: "$movie_title", stars: true, rating : { $concat: [ { $toString: { $sum: '$stars' } }, ' star' ] } } }, { $match: { rating: '5 star' } } ])
```

- todo

operation on movies collection

- todo

- todo

- todo

- todo

- todo

operation on movies collection

- todo

- todo

- todo

- todo

- todo

one-to-one and one-to-many relationship

operation on one-to-one and one-to-many collection

Create one-to-many relation between order and orderitems collection.
• Create order collection with following fields [_id, orderDate, customer, city, latitude, and longitude]. Insert minimum 7 customer details in the order collection.
• Create orderitems collection with following fields [_id, orderid, cart [{ item, price, quantity, and unit },{item, price, quantity, and unit }, ...], Insert minimum 3-4 items in cart for every customer.
Create one-to-one relation between driver and licence collection.
• Create driver collection with the following fields [_id, name, age, city, phone]. Insert 4-5 driver details in the collection.
• Create licence collection with the following fields [_id, driverId, licenceNumber, issuedOn, expireOn]. Insert licence details for all the drivers.

operation on one-to-one and one-to-many collection

- Display all order details with their orderItems details.

```
db.orders.aggregate([ { $lookup: { from: "orderitems", localField: "_id",
foreignField: "orderid", as: "Cart Details" } } ])
```

- Display all order details with their orderItems details whose customer name is ‘ruhan’.

```
db.orders.aggregate([ { $match: { customer: "ruhan" } }, { $lookup: { from:
"orderitems", localField: "_id", foreignField: "orderid", as: "Cart Details" } } ])
```

- Display all drivers with their licence details.

```
db.driver.aggregate([ { $lookup: { from: "licence", localField: "_id", foreignField:
"_id", as: "Licence Details" } } ])
```

- Display all drivers details and their licence number, issuedOn, expireOn only.

```
db.driver.aggregate([{ $lookup: { from: "licence", localField: "_id", foreignField:
"_id", as: "Licence Details" } }, { $project: { _id: true, name: true, 'Licence
Details.licenceNumber': true } }])
```

operation on one-to-one and one-to-many collection

- Rename field name to driverName in driver collection.

```
db.driver.updateOne( { }, { $rename: { name: "driverName" } } )
```

```javascript
var x = "Saleel bagde";
console.log(`The value of x is ${x}`);
```

javascript examples

- Write a javascript program to print documents from emp collection between the range of numbers.

```
const fn = (a = 0, b = 1) => {
  return db.emp.aggregate([ { $skip: a }, { $limit: b } ]);
};
```

- Print all employee names in title case.

```
const fn = () => {
  return db.emp.aggregate([ { $project: { _id: false, ename: true, TitleCaseName: {
$concat: [ { $toUpper: { $substr: [ '$ename', 0, 1 ] } }, { $substr: [ '$ename', 1,
-1 ] } ] } } } ]);
  };
```

- Print the document who is getting the highest salary.

```javascript
const fn = () => {
  db.emp.aggregate([{
    $group: {
      _id: null,
      Salary: {
        $max: '$sal'
      }
    }
  }]).forEach((doc) => {
      console.log(db.emp.aggregate([ { $match: { sal: doc.Salary } } ]));
  });
};
```

- Write a javascript program to print documents from emp collection who is getting the 3rd highest salary.

```javascript
const fn = () => {
  return db.emp.aggregate([{
    $setWindowFields: {
      sortBy: {
        sal: -1
      },
      output: {
        DenseRank: {
          $denseRank: {}
        }
      }
    }
  }]).forEach(doc => {
    if (doc.DenseRank == 3) {
      console.log(doc.ename, " ", doc.sal, " ", doc.DenseRank);
    }
  });
}
```

- TODO

```javascript
const fn = () => {
  db.emp.aggregate([{
    $project: { _id: false, ename: true }
  }]).forEach((doc) => {
    var x = "";
    for (let i = 0; i < doc.ename.length; i++) {
      x = x + doc.ename.substr(i, 1) + "-";
    }
    console.log(x.substr(0, x.length - 1))
  });
}
```

```javascript
var x = "Saleel bagde";
console.log(`The value of x is ${x}`);
```

javascript-mongodb examples

loop in javascript

for (..in) loop

The JavaScript for (..in) statement loops through the enumerable properties of an object.

for (..of) loop

This for (..of) statement lets you loop over the data structures that are inerrable such as Arrays, Strings, Maps, Node Lists,

for (..in) loop

```
let person = {
    firstName: "Saleel",
    lastName: "Bagde",
    rank: 43
};
for (const i in person) {
    console.log(person[i]);
}
```

for (..of) loop

```
let color = ["orange", "blue ", "yellow"];
for (const i of color) {
    console.log(i);
}
```

javascript operation on collection

- Write a javascript program to insert new driver in drive collection.

```
let addDriver = (_id, driverName, age, city, phone) => {
        db.driver.insertOne({
            _id: _id,
            driverName: driverName,
            age: age,
            city: city,
            phone: phone
        });
};
```

javascript operation on collection

- Write a javascript program to insert new driver in drive collection with multiple phone numbers.

```
let addDriver = (_id, driverName, age, city, ...phone) => {
      db.driver.insertOne({
         _id: _id,
         driverName: driverName,
         age: age,
         city: city,
         phone: phone
      });
};
```

```
Enterprise assignment> addDriver( 'driver001', 'Sanjay', 21, 'pune', 9850, 9922,
8080 );
```

javascript operation on collection

- Write a javascript program to insert new driver who must be above or equals to 18 yrs. in drive collection.

```
let addDriver_above18 = (_id, driverName, age, city, phone) => {
        if ( age >= 18 ) {
        db.driver.insertOne({
            _id: _id,
            driverName: driverName,
            age: age,
            city: city,
            phone: phone
        });
        }
        else {
                print("Age of driver must be more or equals to 18 yrs.");
        }
};
```

javascript operation on collection

- Write a javascript program to insert new driver in drive collection, the driverId must be auto_increment number. [e.g. _id: ‘driver1’, ‘driver2’, ‘driver3’, ...]

```
let generateDriverID = (driverName, age, city, phone) => {
        let cnt = db.driver.find().count() + 1;

        db.driver.insertOne({
            _id: "driver" + cnt,
            driverName: driverName,
            age: age,
            city: city,
            phone: phone
        });
};
```

javascript operation on collection

- Write a javascript program to accept the number from user and print only those number of documents from movie collection.

```
let displayFirst_Movies = (x) => {
       return (db.movies.aggregate([
       {
          $project: {
              _id: false,
              director: true,
             movie_title: true,
             gross: true,
             music: true,
             title_year: true,
             genres: true
          }
       }, {
             $limit: x
       }
    ])
   );
};
```

javascript operation on collection

- TODO

javascript operation on collection

- TODO

javascript operation on collection

- TODO

Mongodb - Replication

replication

Replication is a core feature of MongoDB that provides fault tolerance by creating multiple copies of the same data across different servers, known as a replica set.

1. **Primary Node:** The main server that handles all write operations and, by default, read operations.

2. **Secondary Nodes**: Servers that maintain copies of the primary's data. These nodes can take over as the primary in case of failure.

3. **Automatic Failover**: If the primary node fails, one of the secondary nodes is automatically elected as the new primary.

4. **Data Redundancy**: Even if one node fails, the other nodes still have the data, ensuring no data loss.

5. **Voting**: A replica set can have up to 50 members but only 7 voting members.

Step: 1 Create 3 blank folders in Ubuntu

```
shell~$ mkdir a b c d
```

Step: 2 Start Mongodb Server

```
./mongosh --dbpath=/home/saleel/a --bind_ip=192.168.150.74 --port=17017 --replSet=rs1
./mongosh --dbpath=/home/saleel/b --bind_ip=192.168.150.74 --port=17018 --replSet=rs1
./mongosh --dbpath=/home/saleel/c --bind_ip=192.168.150.74 --port=17019 --replSet=rs1
```

Step: 3 Start Mongodb Client

```
mongosh --host=192.168.150.74 --port=27017 db1
```

Step: 4 Check replSet is created of not

- `rs.status();`

Step: 5 Create replication initiate document

```
rs.initiate ({
      _id: "rs1",
      members:[
                { _id: 0, host: "192.168.150.74:27017" },
                { _id: 1, host: "192.168.150.74:27018" },
                { _id: 2, host: "192.168.150.74:27019" }
              ]
  });
```

Step: 6 Add replSet afterwards

- `rs.add({ _id: 3, host: "192.168.150.74:27020" });`

replication

replication

```js
var x = "Saleel bagde";
console.log(`The value of x is ${x}`);
```

```
C:\Users\Admin\Desktop\JS> cls;  node .\app.js OR node .
```

```
package.json
"main": "app.js"
```

Node.js-mongodb examples

mongoimport movie.csv collection

- Write a program to import movies.csv file using Node.js

```
import { exec } from 'child_process';
exec('mongoimport --host=192.168.100.91 --port=27017 --db="db1" --collection="movies" --type="csv"
      --file="C:/data/movie.csv" --headerline', (err, res) => {
    if (err) {
       console.log("Some error occurred");
    }
    else {
         console.log("movie documents imported!");
    }
});
```

list current database

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

const run = async () => {
    try {
        await client.connect();
        const database = client.db('db1');
        console.log(database.databaseName);
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log('connection closed...');
    }
}
run();
```

list all databases

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

const run = async () => {
    try {
        await client.connect();
       const dbList = client.db().admin().listDatabases();

       for (const x of (await dbList).databases) {
            console.log(`${x.name}`);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        client.close();
        console.log(`connection closed...`);
    }
}
run();
```

list collections from all databases

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

const run = async () => {
    try {
        await client.connect();
        const dbList = client.db().admin().listDatabases();

        for (const x of (await dbList).databases) {
            const database = client.db(x.name);
            const y = database.listCollections();

            for await (const doc of y) {
                console.log( x.name + " --> " + doc.name);
            }
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        client.close();
        console.log('connection closed...');
    }
}
run();
```

createCollection()

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        await database.createCollection("employee");
        console.log("Collection created...");
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed");
    }
};
run();
```

capped collection

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        await database.createCollection("doctor", { capped: true, size: 100, max: 2 })
        console.log("Collection created...");
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...")
    }
};
run();
```

listCollections()

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const cursor = database.listCollections();

       for await (const doc of cursor) {
            console.log(doc.name);
        }
        console.log("Collection created...");
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed");
    }
};
run();
```

listCollections() using Array()

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        let arr = new Array();
        await client.connect();
        const database = client.db("db1");
        const cursor = database.listCollections();

       for await (const doc of cursor) {
            arr.push(doc.name);
        }
        console.log(arr);
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
};
run();
```

for (... of) loops

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        let arr = new Array();
        await client.connect();
        const database = client.db("db1");
        const cursor = database.listCollections();

        for await (const doc of cursor) {
            arr.push(doc.name);
        }
        console.log(arr);
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed");
    }
};
run();
```

array.forEach() loops

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        let arr = new Array();
        await client.connect();
        const database = client.db("db1");
        const cursor = database.listCollections();

        for await (const doc of cursor) {
            arr.push(doc.name);
        }
        arr.forEach((value, index) => {
            console.log(index + value);
        });
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
};
run();
```

for (... in) loops

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        let arr = new Array();
        await client.connect();
        const database = client.db("db1");
        const cursor = database.listCollections();

        for await (const doc of cursor) {
            arr.push(doc.name);
        }
        for (const key in arr) {
            console.log(arr[key]);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
};
run();
```

createCollection() and add new document

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        await database.createCollection("person");
        const p = database.collection("person");
        await p.insertOne({ _id: 1, ename: 'saleel' })
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
}
run();
```

renameCollection()

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const p = database.collection("person");
        await p.rename("newPerson");
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
}
run();
```

dropCollection()

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const p = database.collection("person");
        await p.drop();
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
}
run();
```

countDocuments({ })

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");
        console.log(await m.countDocuments());
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...");
    }
}
run();
```

countDocuments({query})

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");
        console.log(await m.countDocuments({ duration: 100 }));
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...");
    }
}
run();
```

insertOne({ })

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");
        await m.insertOne({ _id: 1, title: "DON" });
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...");
    }
}
run();
```

insertOne({ }) with arguments

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run(id, relese, color, director, title, gross) {
    try {
        await client.connect();
        const database = client.db("db1");
        const e = database.collection("movies");
        await e.insertOne({ _id: id, relese: relese, director: director, title: title });
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed");
    }
};
run(1, '11-05-1978', 'Chandra Barot', 'DON');
```

Note:- _id must be auto generated max() + 1

insertOne({ })

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run(relese, director, title) {
    try {
        const database = client.db("db1");
        const e = database.collection("movies");
        let cnt = 0;
        const cursor = e.aggregate([{ $group: { _id: null, x: { $max: "$_id" } } }]);
        for await (const doc of cursor) {
            cnt = doc.x + 1;
        }
        await e.insertOne({ _id: cnt, relese: relese, director: director, title: title });
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...")
    }
};
run('11-05-1978', 'Chandra Barot', 'DON');
```

insertOne({ }) using arrow function

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

const run = async (id, relese, color, director, title, gross) => {
    try {
        await client.connect();
        const database = client.db("db1");
        const e = database.collection("movies");
        await e.insertOne({ _id: id, relese: relese, director: director, title: title });
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed");
    }
};
run(30, '11-05-1978', 'Chandra Barot', 'DON');
```

insertMany([{ }, { }])

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("mv");
        await m.insertMany([{ _id: 31, title: "DON" }, { _id: 32, title: "Trishul" }]);
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...");
    }
}
run();
```

find({ }, { })

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");
        const cursor = m.find({}, {});

      for await (const doc of cursor) {
            console.log(doc);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
}
run();
```

find({ }, { projection })

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const movie = database.collection("movies");
        const cursor = movie.find({}, { projection: { _id: false, Title: '$movie_title' }
});

        for await (const doc of cursor) {
            console.log(doc);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...");
    }
}
run();
```

find({ }, { projection })

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const movie = database.collection("movies");
        const cursor = movie.find({ duration: { $gt: 250 } }, { projection: { _id: false,
movie_title: true, duration: true } });

        for await (const doc of cursor) {
            console.log(fn1(doc));
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...");
    }
};
function fn1(doc) {
    return doc.movie_title + " ---> " + doc.duration
};
```

runtime collectionName and fieldNames

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run(_collectionName, _query, _fieldList) {
    try {
        await client.connect();
        const database = client.db("db1");
        const i = database.collection(_collectionName);
        const cursor = i.find(_query, { projection: _fieldList });

       for await (const doc of cursor) {
            console.log(doc);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Connection Closed...");
    }
};

run("movies", { genres: /^Horror$/ }, { _id: false, relese: true, director: true, title:
"$movie_title", genres: true });
```

find( ).filter({ query }).project({ fieldList })

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.68:27017");

async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");
        const cursor = m.find().filter({ language: 'Hindi' }).project({ _id: false, color:
true, movie_title: true, director: true, language: true });

        for await (const doc of cursor) {
            console.log(doc);
        }
    } catch (error) {
        console.log(error.message);
    }
    finally {
        await client.close();
    }
}
run();
```

find({ query }, { projection }) - like

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const movie = database.collection("movies");
        const cursor = movie.find({ genres: /^H/ }, { projection: { _id: false, Title: '$movie_title',
        genres: true } });
        for await(const doc of cursor){
            console.log(doc.Title, doc.genres);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...");
    }
}
run();
```

find({ query }, { projection }) - $exists

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const e = database.collection("emp");
        const cursor = movie.find({ phone: { $exists: true } }, { projection: { _id: false, Title:
        '$movie_title', genres: true } });
        for await(const doc of cursor){
            console.log(doc.Title, doc.genres);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...");
    }
}
run();
```

find({ query }, { projection })

- Write a program to fetch all document from the ‘movies’ collection whose color = ‘Eastman’ and language = ‘Hindi’ and genres = ‘Comedy’ using Node.js

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");
        const cursor = m.find({color:'Eastman', language:"Hindi", genres:"Comedy"}, {
        projection:{_id:false, color:true, title:"$movie_title", language:true, genres:true}});
        for await (const doc of cursor) {
            console.log(doc);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally{
        await client.close();
        console.log("Client connection closed");
    }
};
run();
```

find({ query as variable }, { projection as variable})

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");

        const query = { country: 'USA' };
        const fieldList = { _id:false, color:true, director:true, title:'$movie_title', country:true }

        const cursor = m.find(query, { projection:fieldList });

        for await(const doc of cursor){
            console.log(doc);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally{
        await client.close()
    }
};
run();
```

find({ query as variable }, { projection as variable})

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");

        const query = { $and:[ {country: 'USA' }, { genres:{$ne:''} } ]};
        const fieldList = {_id:false, color:true, director:true, title:'$movie_title', country:true,
        genres:true}

        const cursor = m.find(query, {projection:fieldList});
        for await(const doc of cursor){
            console.log(doc);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally{
        await client.close()
    }
};
run();
```

find().skip{ m.countDocuments() – param }

- Write a program to fetch nth-document from the ‘movies’ collection using Node.js

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run(x) {
  try {
      await client.connect();
      const database = client.db("db1");
      const m = database.collection("movies");
      const cursor = m.find({}, {}).skip(await m.countDocuments() - x);
      for await(const doc of cursor){
        console.log(doc.ename, doc.job, doc.sal);
      }
  } catch (error) {
      console.log(error.code, error.name, error.message);
  }
  finally {
      await client.close();
      console.log("Client connection closed");
  }
};
run(4);
```

findOne({ }, { projection })

- Write a program to fetch first document from the ‘movies’ collection, give alias name to movie_title field using Node.js

```
import { MongoClient } from "mongodb";
const client =  new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const movie = database.collection("movies");
        const cursor = await movie.findOne({}, { projection:{ _id:false, Title: '$movie_title' }});
        console.log(cursor);
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally{
        await client.close();
        console.log("Client connection closed...");
    }
}
run();
```

aggregate([{ }])

- Write a program to display all documents from ‘movies’ database.

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const e = database.collection("movies");
        const cursor = e.aggregate();
        for await (const doc of cursor){
            console.log(doc.movie_title);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed");
    }
};
run();
```

aggregate([{ }])

- Display all documents from every collection in the database.

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const cursor1 = database.listCollections();
        for await(const doc of cursor1) {
            const e =  database.collection(doc.name);
            const cursor2 = e.aggregate();
            for await(const c of cursor2) {
                console.log(c);
            }
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
}
run();
```

aggregate([{ $project }])

- Write a program to fetch all document using aggregate method from ‘movies’ collection using Node.js

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");

async function run() {
    try {
        const database = client.db("db1");
        const e = database.collection("movies");
        const cursor = e.aggregate([{ $project:{_id:true, relese:true, color:true, director:true,
movie_title:true, genres:true, gross:true }}]);

        for await (const doc of cursor) {
            console.log(doc._id, doc.relese, doc.color, doc.director, doc.movie_title, doc.genres,
doc.gross);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Client connection closed...")
    }
};
run();
```

aggregate([{$match}]) with await cursor.hasNext() == true

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run(movieDurationion) {
    try {
        const database = client.db("db1");
        const m = database.collection("movies");
        const cursor = m.aggregate([{ $match: { $expr: { $gte: ['$duration', movieDurationion] },
        duration: { $not: { $eq: '' } } } }, { $project: { _id: false, duration: true } }]);

        if(await cursor.hasNext() == true) {
            for await(const doc of cursor) {
                console.log(doc);
            }
        }else{
            console.log("No documents found...");
        };
     } catch (error) {
         console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
};
run(400);
```

aggregate([{ $match }, { $project }])

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        const database = client.db("db1");
        const e = database.collection("movies");
        let cnt = 1;
        const cursor = e.aggregate([{ $match:{genres:/^Horror$/ }}, { $project:{_id:true, relese:true,
        color:true, director:true, movie_title:true, genres:true, gross:true }}]);
        for await (const doc of cursor) {
            console.log(cnt, doc._id, doc.relese, doc.color, doc.director, doc.movie_title, doc.genres,
                doc.gross);
            cnt++;
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
};
run();
```

aggregate([{ $match }, { $project }]) using variables

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.150.24:27017");
async function run() {
    try {
        const database = client.db("db1");
        const e = database.collection("movies");
        const query = { $match: { genres: /^Horror$/ } };
        const projection = { $project: {director: true, movie_title: true, genres: true } };
        const cursor = e.aggregate([query, projection]);
        for await (const doc of cursor) {
            console.log('Director [' + doc.director + '] Movie Title [' + doc.movie_title + '] Genres
[' + doc.genres + ']');
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
};
run();
```

aggregate([{ $match }]) with $regex

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const e = database.collection("movies");
        const cursor = e.aggregate([{ $match:{ genres: { $regex:/^Horror/ }}}, { $project:{
        movie_title:true,  genres:true }}]);
        for await(const doc of cursor) {
            console.log(doc);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally
    {
        await client.close();
        console.log("Connection closed...");
    }
}
run();
```

aggregate([{ $match }]) with $expr

- Write a program to fetch all document from ‘movies’ collection where movie duration is more than 300 minutes using Node.js

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");
        const cursor = m.aggregate([{ $project: { title: "$movie_title", duration: true, x: { $literal:
        300 } } }, { $match: { $expr: { $gte: ['$duration', '$x'] } } }]);
        for await(const doc of cursor) {
            console.log(doc);
        };
     } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Connection closed...");
    }
};
run();
```

aggregate([{ $setWindowFields }]) – documentNumber()

```
import { MongoClient } from "mongodb";
const client =  new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");
        const cursor = m.aggregate([{$match:{genres:{$ne:'' } } }, {$project:{_id:false, color:true,
        title:'$movie_title', director:true, genres:true } }, {$setWindowFields:{sortBy:{director:1},
        output:{x:{$documentNumber:{}}}}} ]);

        for await(const doc of cursor) {
            console.log(doc);
        };
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally{
        await client.close();
        console.log("Connection closed");
    }
};
run();
```

aggregate([{ $setWindowFields }]) – denseRank()

```
import { MongoClient } from "mongodb";
const client =  new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");
        const cursor = m.aggregate([{ $match:{genres:{$eq:'' } } }, { $project:{_id:false, color:true,
        title:'$movie_title', director:true, genres:true } }, {$setWindowFields:{sortBy:{director:1},
        output:{x:{$denseRank:{}}}}} ]);

        for await(const doc of cursor) {
            console.log(doc);
        };
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally{
        await client.close();
        console.log("Connection closed");
    }
};
run();
```

aggregate([{ $setWindowFields }]) – denseRank()

```
import { MongoClient } from "mongodb";
const client =  new MongoClient("mongodb://192.168.100.91:27017");
async function run(y) {
    try {
        await client.connect();
        const database = client.db("db1");
        const m = database.collection("movies");
        const cursor = m.aggregate([{$match:{genres:{$eq:'' } } }, {$project:{_id:true, color:true,
        title:'$movie_title', director:true, genres:true } }, { $setWindowFields: {
        partitionBy:"$color", sortBy: { _id:1 }, output:{ x:{ $denseRank:{}}}}}, { $match: { x: y } }
        ]);
        for await(const doc of cursor) {
            console.log(doc);
        };
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally{
        await client.close();
        console.log("Connection closed");
    }
};
run(5);
```

Node.JS operation – aggregate([{ $match }, { $sort }])

- Write a program to get the movie duration from the user and fetch all document from ‘movies’ collection for the give duration using Node.js

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run(movieDurationion) {
    try {
        const database = client.db("db1");
        const m = database.collection("movies");
        const cursor = m.aggregate([{ $project: { title: "$movie_title", duration: true } },
              { $match: { $expr: { $gte: ['$duration', movieDurationion] }, duration: { $not: { $eq: ''
                        } } } }, { $sort:{ duration: 1 }}
        ]);
        for await(const doc of cursor){
            console.log(doc);
        };
     } catch (error) {
         console.log(error.code, error.name, error.message);
     }
     finally {
         await client.close();
     }
};
run(250);
```

Node.JS operation – aggregate([{ $lookup }]) – one-to-one

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");

async function run() {
    try {
        const database = client.db("db1");
        const p = database.collection("person");
        const cursor = p.aggregate([{ $lookup: {
                from: "passport",
                localField: "_id",
                foreignField: "_id",
                as: "PassportDetails",
                pipeline: [{ $project: { _id: 0, name: 1, city: 1, "passport number": 1} }] } }]);
        for await(const doc of cursor) {
            console.log(doc);
        };
     } catch (error) {
         console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
};
run();
```

updateOne({ }, { $set })

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const p = database.collection("person");
        await p.updateOne({
            _id: 1
        }, {
            $set: { salary: 45000, gender: 'M'
            }
        })
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
        console.log("Connection closed");
    }
};
run();
```

updateOne({ }, { $push })

```
import { MongoClient } from "mongodb";
const client = new MongoClient("mongodb://192.168.100.91:27017");
async function run() {
    try {
        await client.connect();
        const database = client.db("db1");
        const p = database.collection("person");
        await p.updateOne({
            _id: 1
        }, {
            $push: {
                phone: {
                    $each: [-2, -1],
                    $position: 0
                }
            }
        })
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client.close();
    }
};
run();
```

MongoDB to Redis

```
import { MongoClient } from "mongodb";
import { createClient } from "redis";
const client1 = new MongoClient("mongodb://192.168.100.91:27017");
const client2 = createClient({ url: "redis://192.168.100.84:6379" });

async function run() {
    try {
        let cnt = 1;
        await client1.connect();
        await client2.connect();
        const database = client1.db("db1");
        const m = database.collection("movies");
        const cursor = m.find({}, {projection:{ _id: 0, title:'$movie_title' }});

        for await(const doc of cursor) {
            let x = await client2.SET("Title-" + cnt++, doc.title);
            console.log("KEY" + cnt + " created ... " + x);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client1.close();
        await client2.QUIT();
    }
};
run();
```

MongoDB to Redis

```
import { MongoClient } from "mongodb";
import { createClient } from "redis";
const client1 = new MongoClient("mongodb://192.168.100.91:27017");
const client2 = createClient({ url: "redis://192.168.100.84:6379" });

async function run() {
    try {
        let cnt = 0;
        await client1.connect();
        await client2.connect();
        const database = client1.db("db1");
        const m = database.collection("movies");
        const cursor = m.find({}, {});

        for await(const doc of cursor) {
            await client2.SET("title:" + ++cnt, doc.movie_title);
        }
    } catch (error) {
        console.log(error.code, error.name, error.message);
    }
    finally {
        await client1.close();
        await client2.QUIT();
    }
};
run();
```

aggregate([{ }, { }])

- TODO

aggregate([{ }, { }])

- TODO

aggregate([{ }, { }])

- TODO

aggregate([{ }, { }])

- TODO

```
create table book (id raw(16) primary key, data clob check(data is json));

select book.*
        from books,
        json_table(data,'$'
        columns(isbn   varchar2(20) path '$.isbn',
                title  varchar2(20) path '$.title',
                price  varchar2(10) path '$.price',
                author varchar2(20) path '$.author',
                phone  varchar2(10) path '$.phone')) book
```

"If you cry because the sun has gone out of your life, your tears will prevent you from seeing the stars."

Rabindranath Tagore

JavaScript is a cross-platform, object-oriented scripting language used to make webpages interactive (e.g., having complex animations, clickable buttons, popup menus, etc.). There are also more advanced server side versions of JavaScript such as Node.js, which allow you to add more functionality to a website.

JavaScript

Session 1

module export and import

module.js

- `export { fn1, fn2, personObj as person, colorArray as colors };`
- `export { fn1 as a, fn2 as b, personObj as person, colorArray as colors };`

app.js

- `import { fn1, fn2 } from "./module.js";`
- `import { a, b, person, colors } from "./module.js";`
- `import * as all from "./module.js";`

todo