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)
```

Class Room

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.

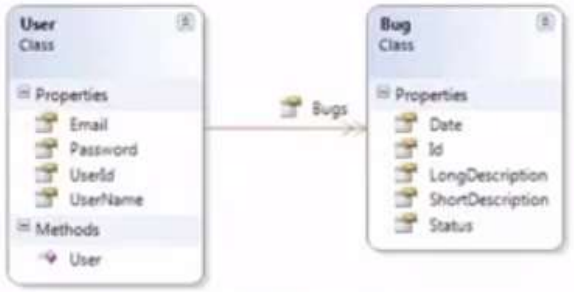
Remember:

- **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.

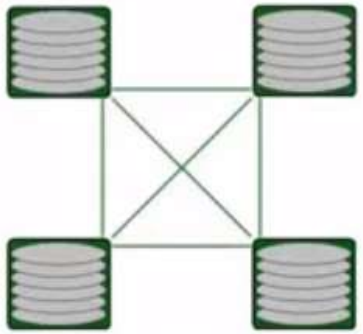
why NoSQL

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

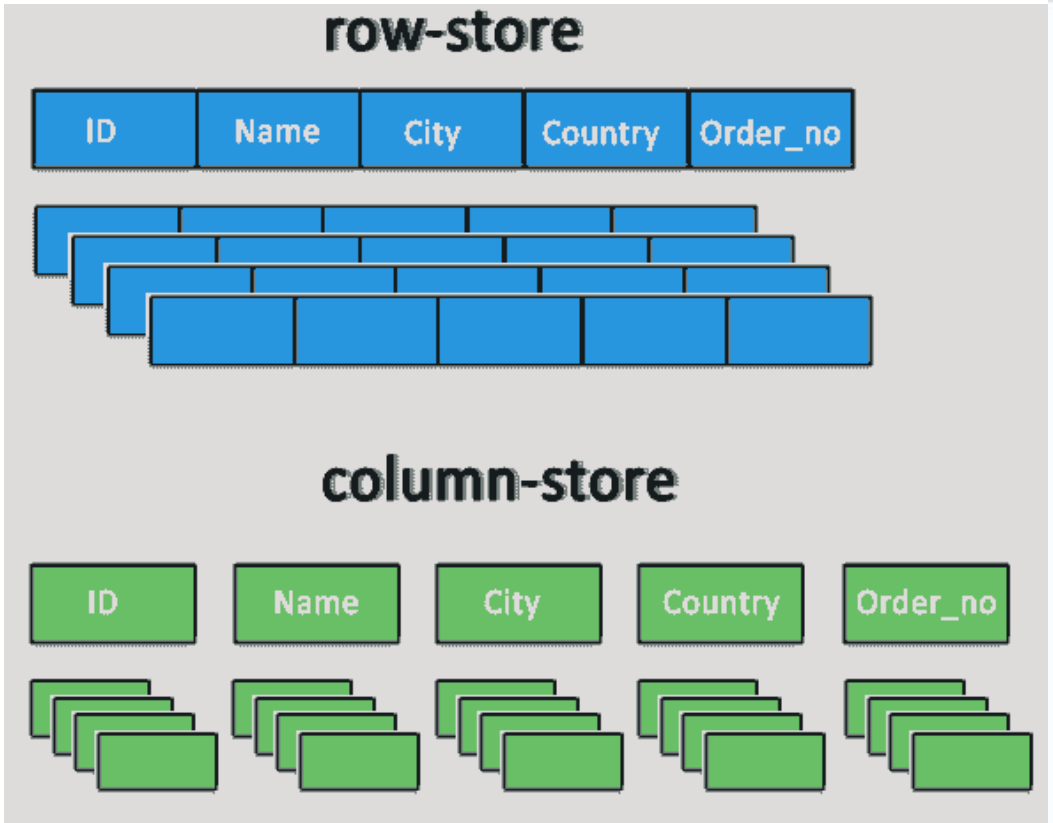
NoSQL Categories

There are 4 basic types of NoSQL databases.

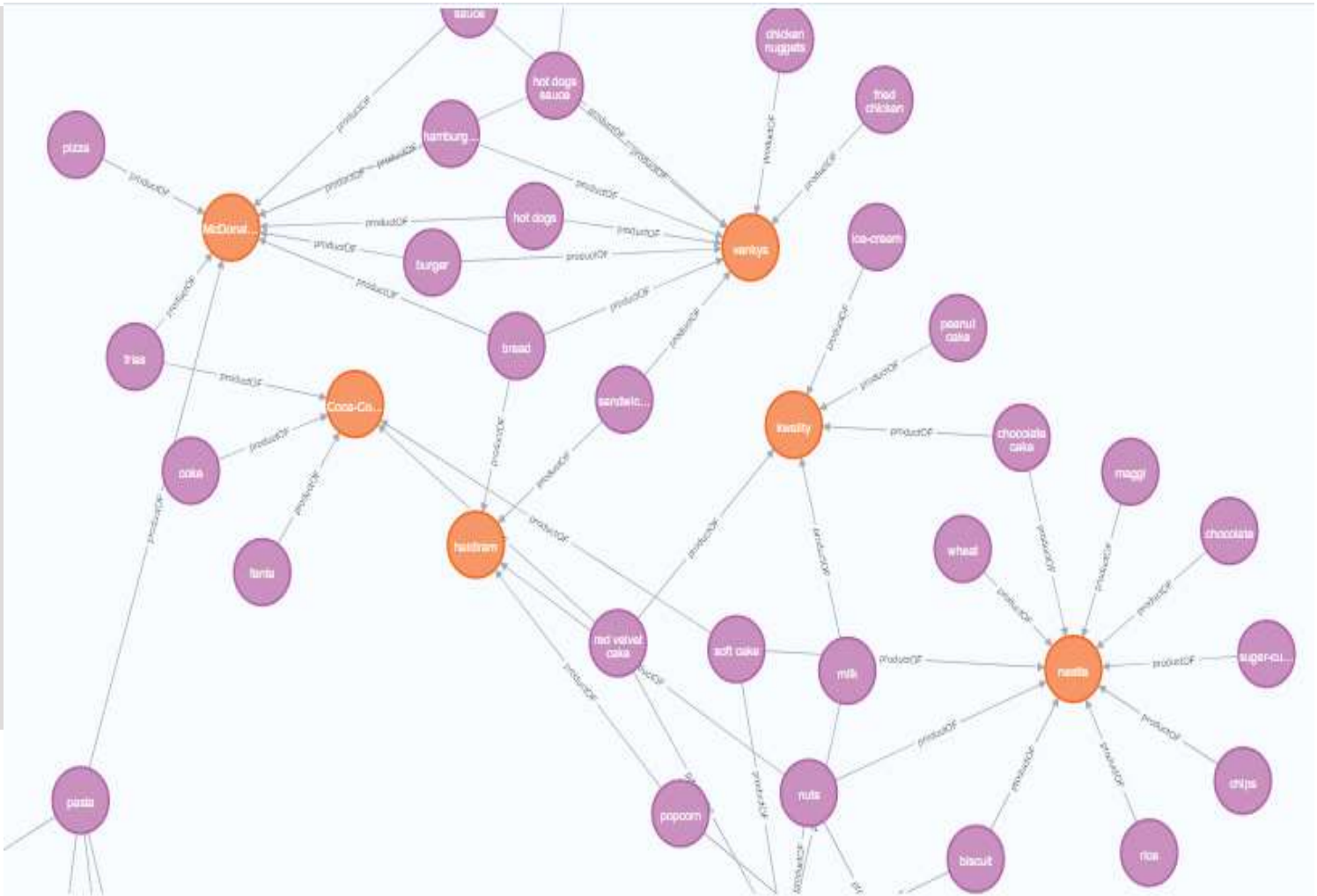
Key-value stores	Key-value stores, or key-value databases, implement a simple data model that pairs a unique key with an associated value. e.g. • **Redis**
Column-oriented	Wide-column stores organize data tables as columns instead of as rows. e.g. • **hBase, Cassandra**
Document oriented	Document databases, 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

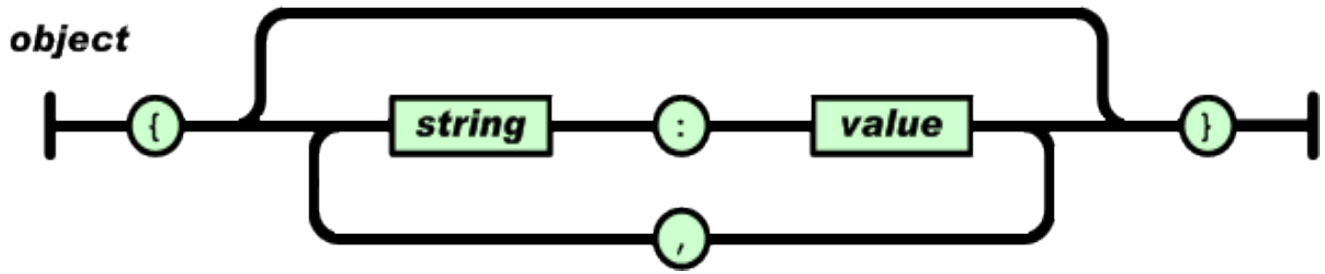
Column-oriented



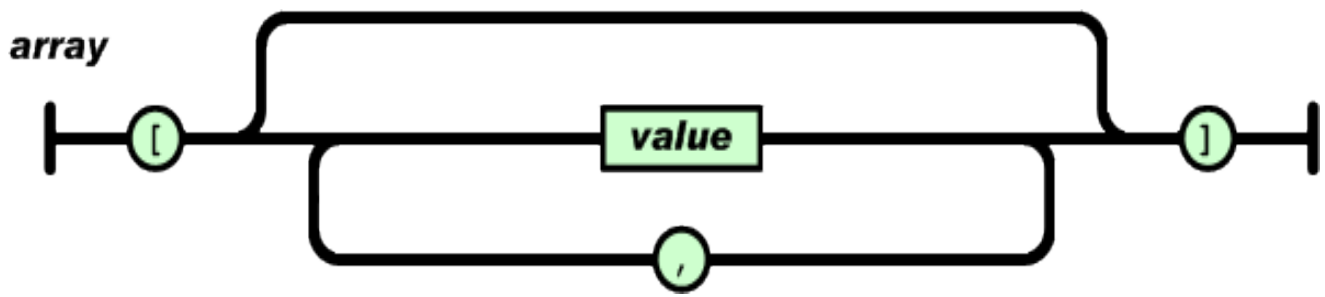
Graph



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

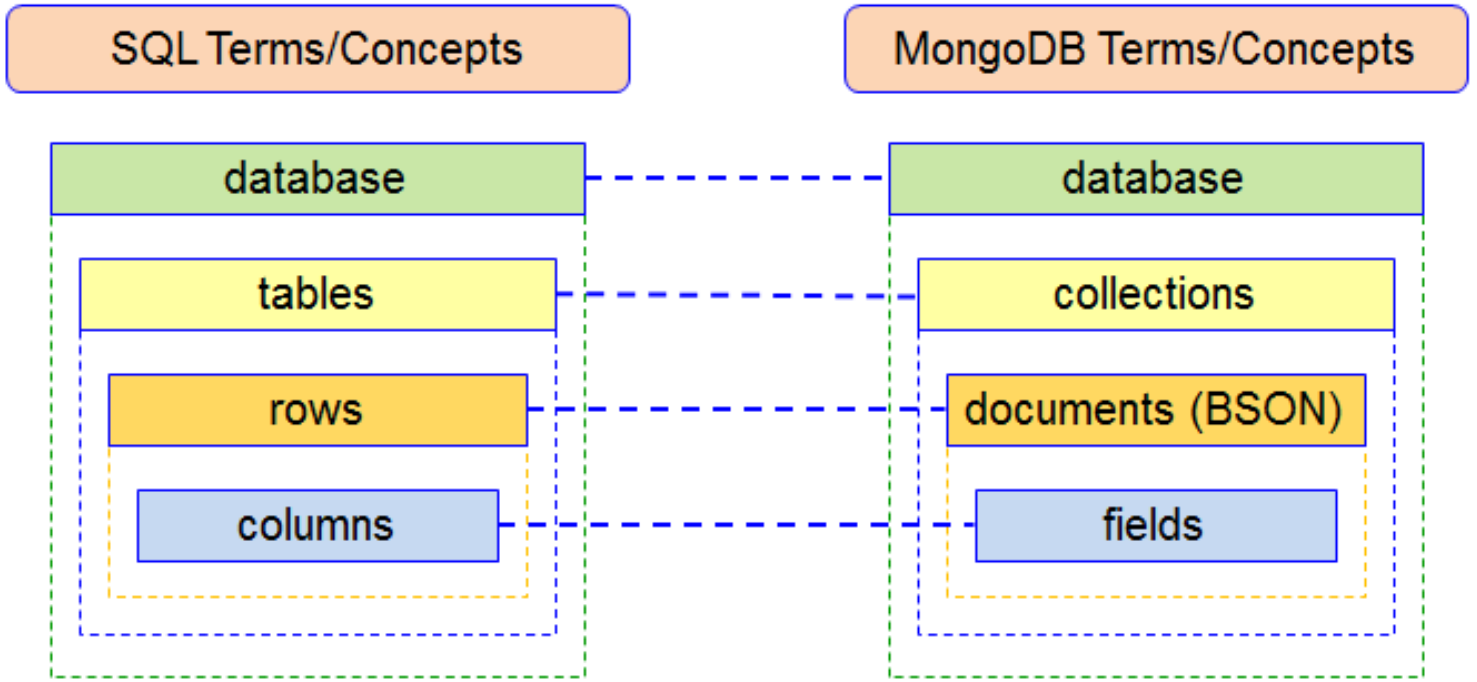
MongoDB is a cross-platform document-oriented database program. Classified as a NoSQL database.

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.

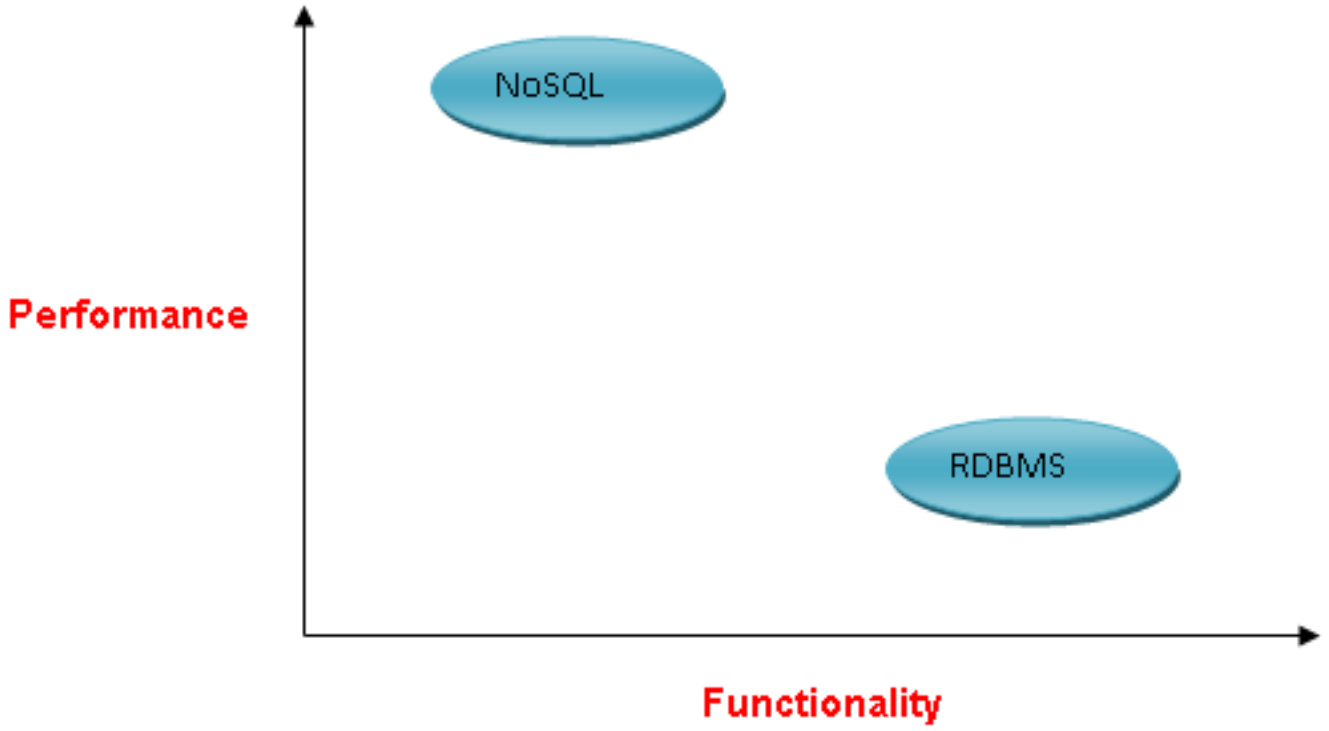
Core MongoDB Operations (CRUD), stands for create, read, update, and delete.

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.

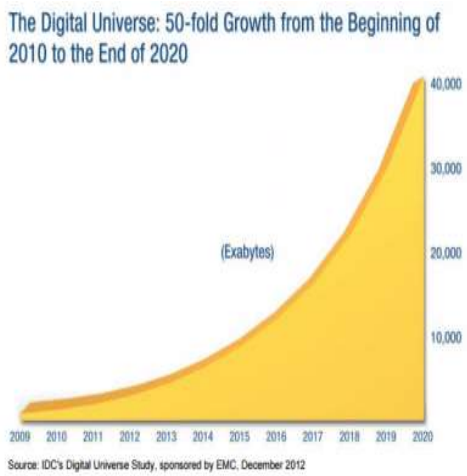


3Vs

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

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

Note: Always give --dbpath in ""

```
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"
    }
}])
```