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

Database Technologies – MongoDB

iet

```
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.

APACHE HBASE Cassandra riak CouchDB relax mongoDB HYPERTABLE INC Neo4j redis

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:

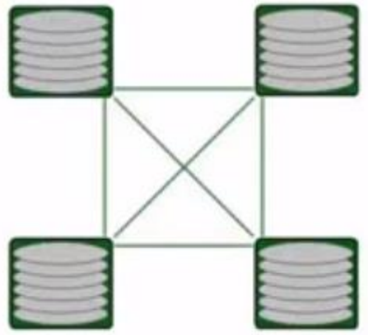
- **Horizontal scaling** means that you **scale** by adding more machines into your pool of resources.
- **Vertical scaling** means that you **scale** by adding more power (**CPU**, **RAM**) to an existing machine.

why NoSQL

Next Generation Databases

Not Only SQL

User
Class
Properties
Email
Password
UserId
UserName
Methods
User
Bugs
Bug
Class
Properties
Date
Id
LongDescription
ShortDescription
Status
Non – Relational


Distributed Architecture

Open Source

Horizontal Scaling
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

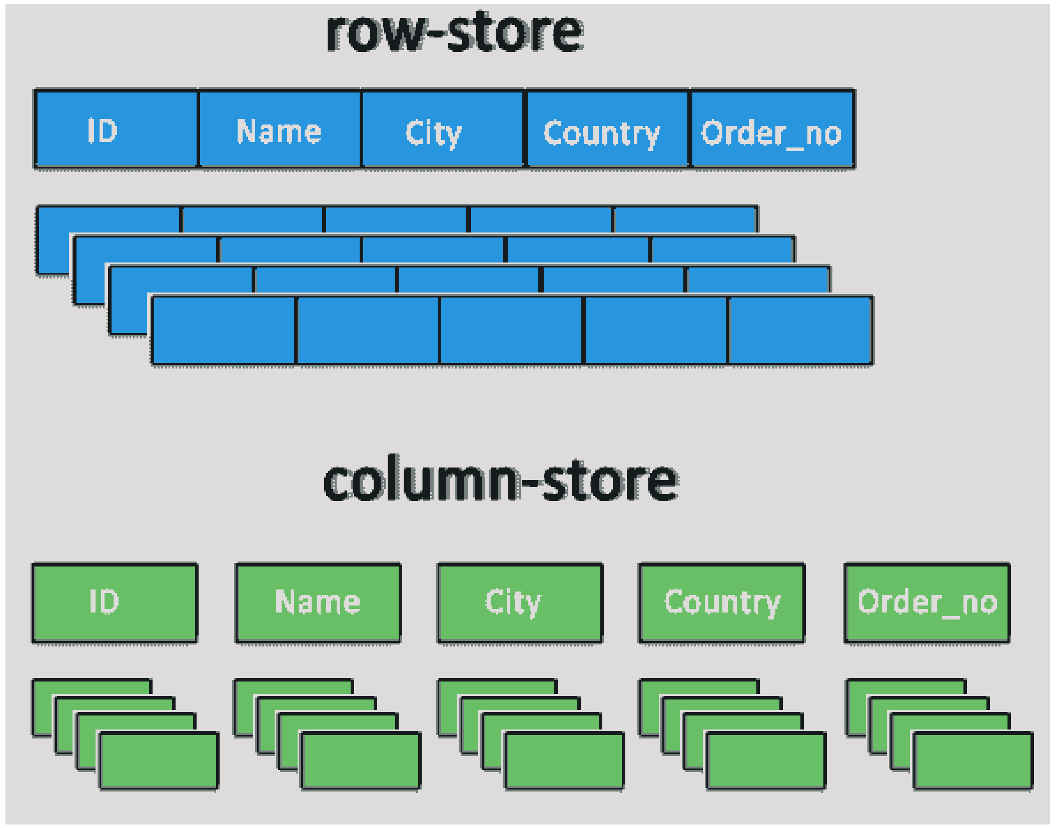
NoSQL Categories

There are 4 basic types of NoSQL databases.

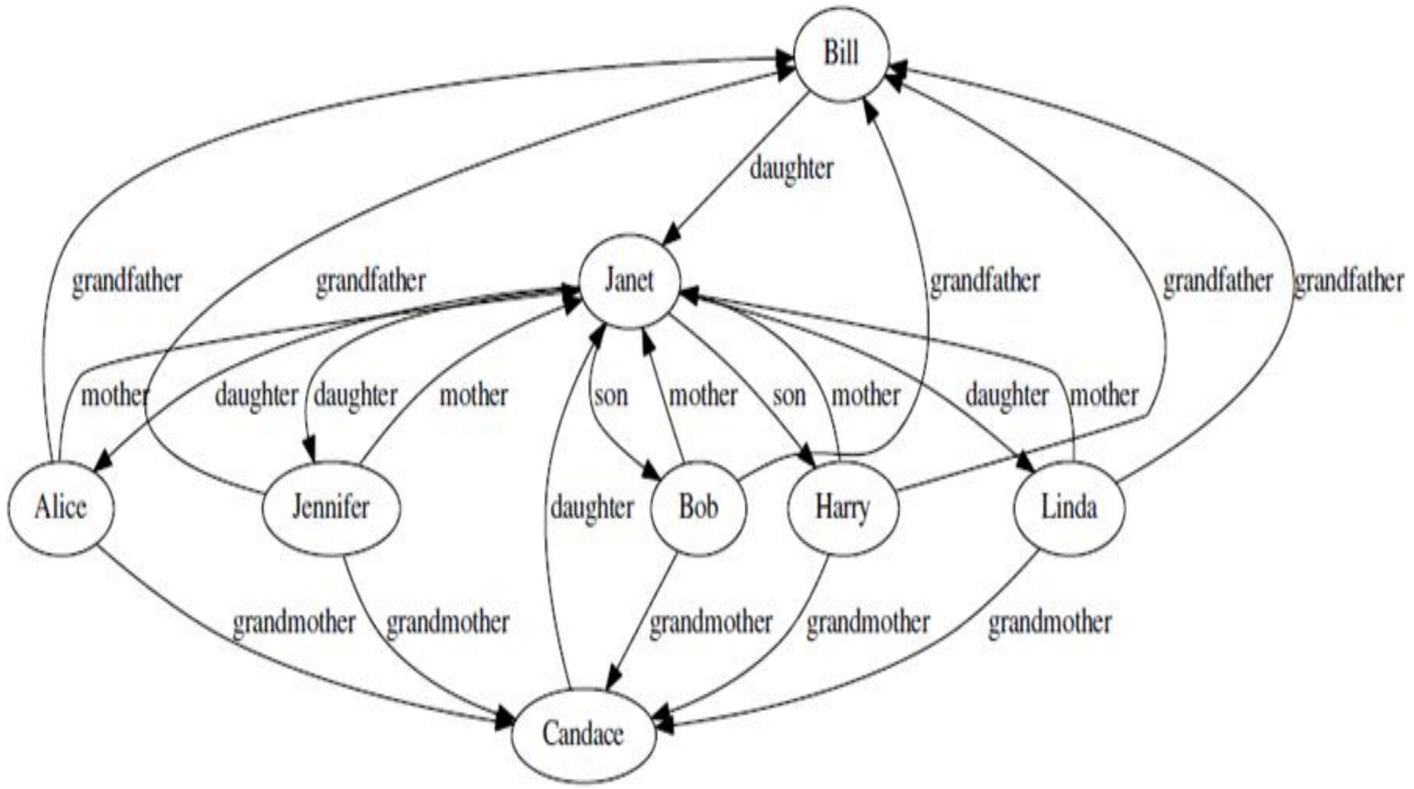
Key-value stores	Key-value stores, or key-value databases, implement a simple data model that pairs a unique key with an associated value. e.g. • **Redis**
Column-oriented	Wide-column stores organize data tables as columns instead of as rows. e.g. • **hBase, Cassandra**
Document oriented	Document databases,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

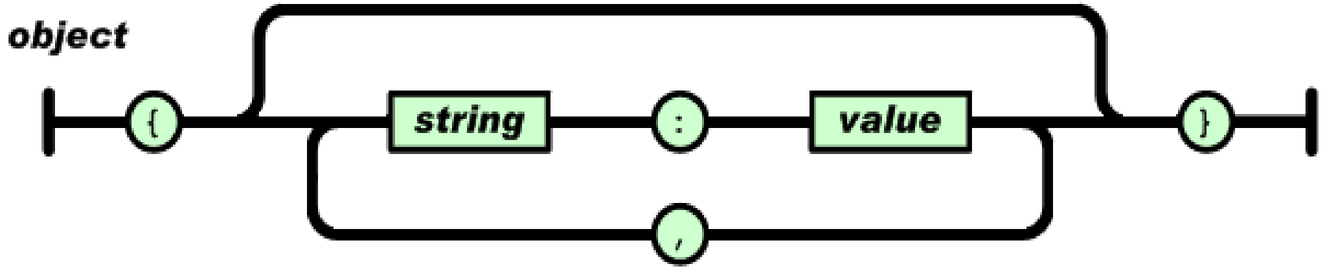
Column-oriented



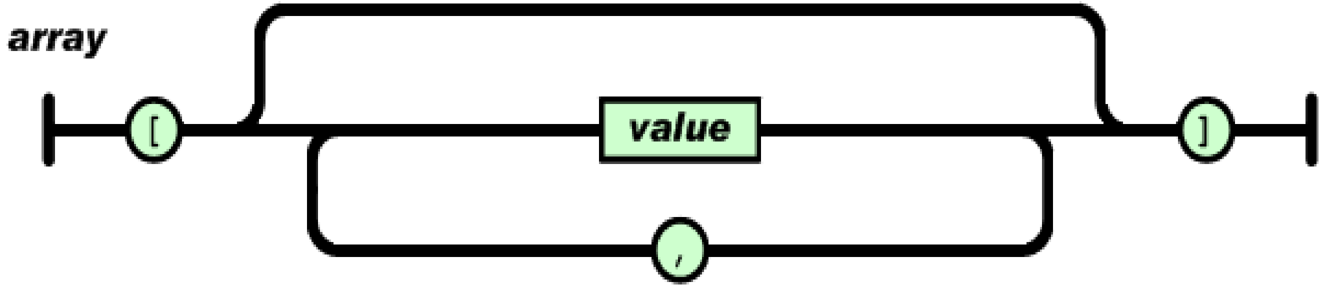
Graph



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



An array is an ordered collection of values.



SQL vs NoSQL Database

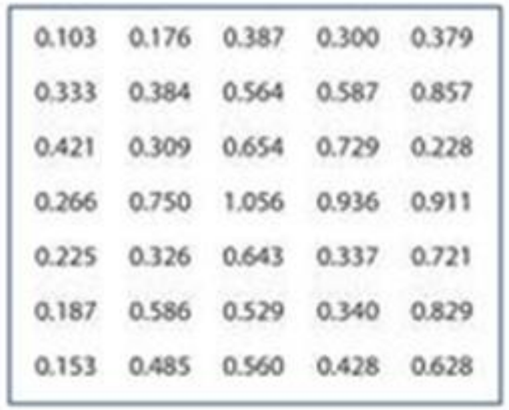
Relational databases are commonly referred to as SQL databases because they use SQL (structured query language) as a way of storing and querying the data.

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 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

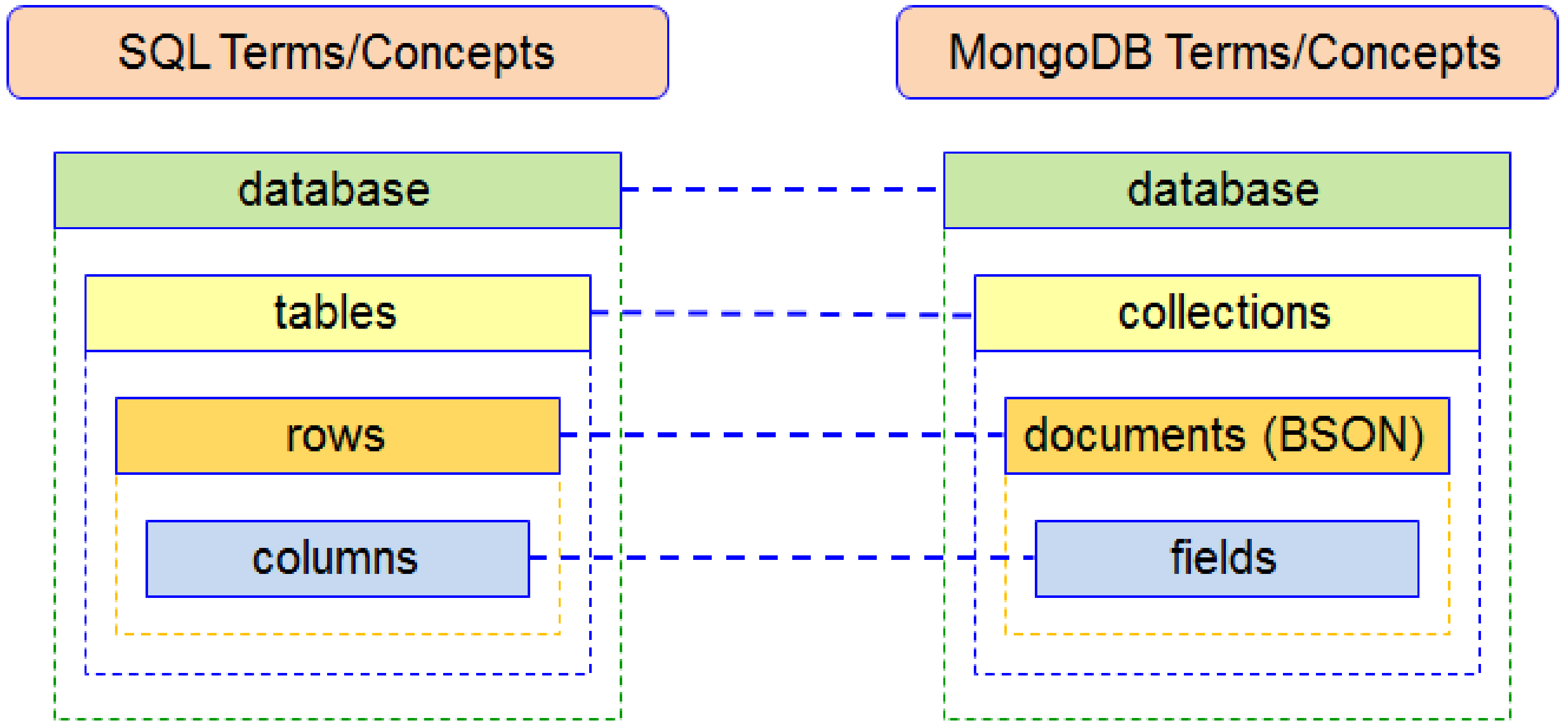
MongoDB is a cross-platform document-oriented database program. Classified as a NoSQL database.

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.

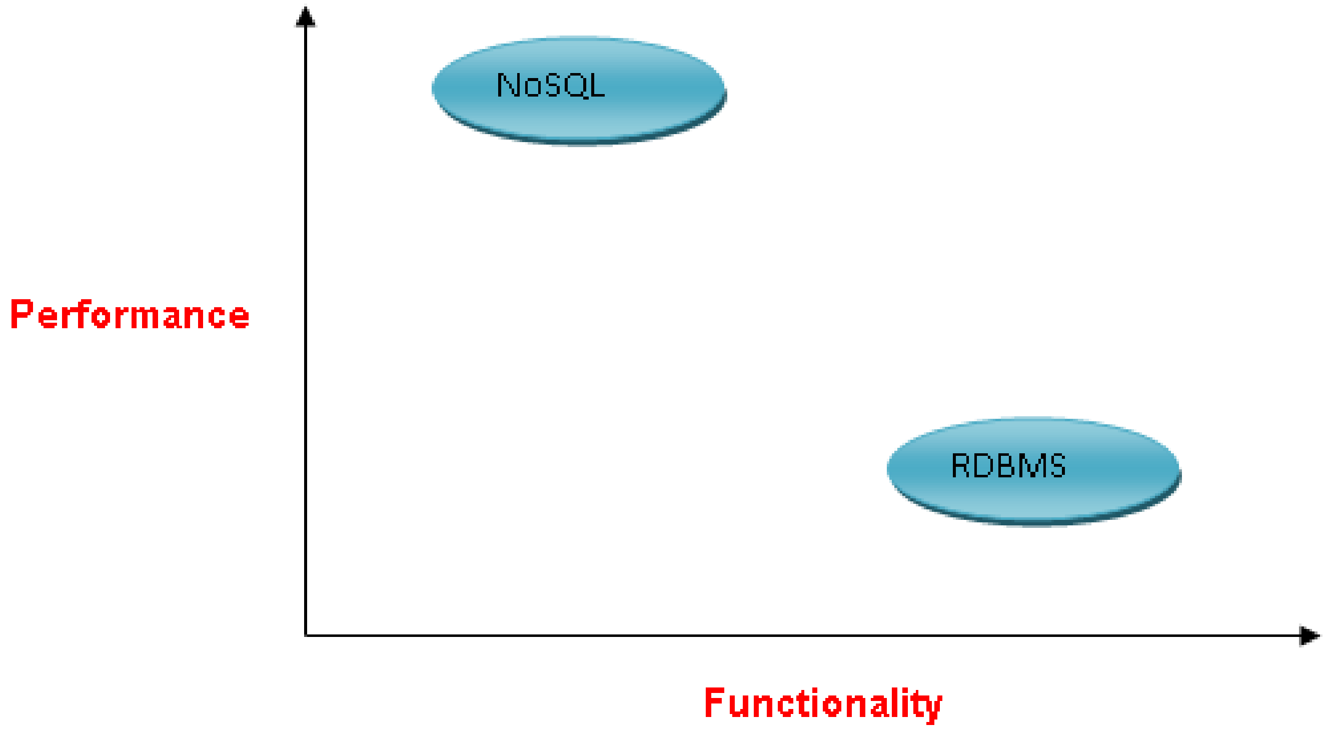
SQL/MongoDB Terms:



MongoDB

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.

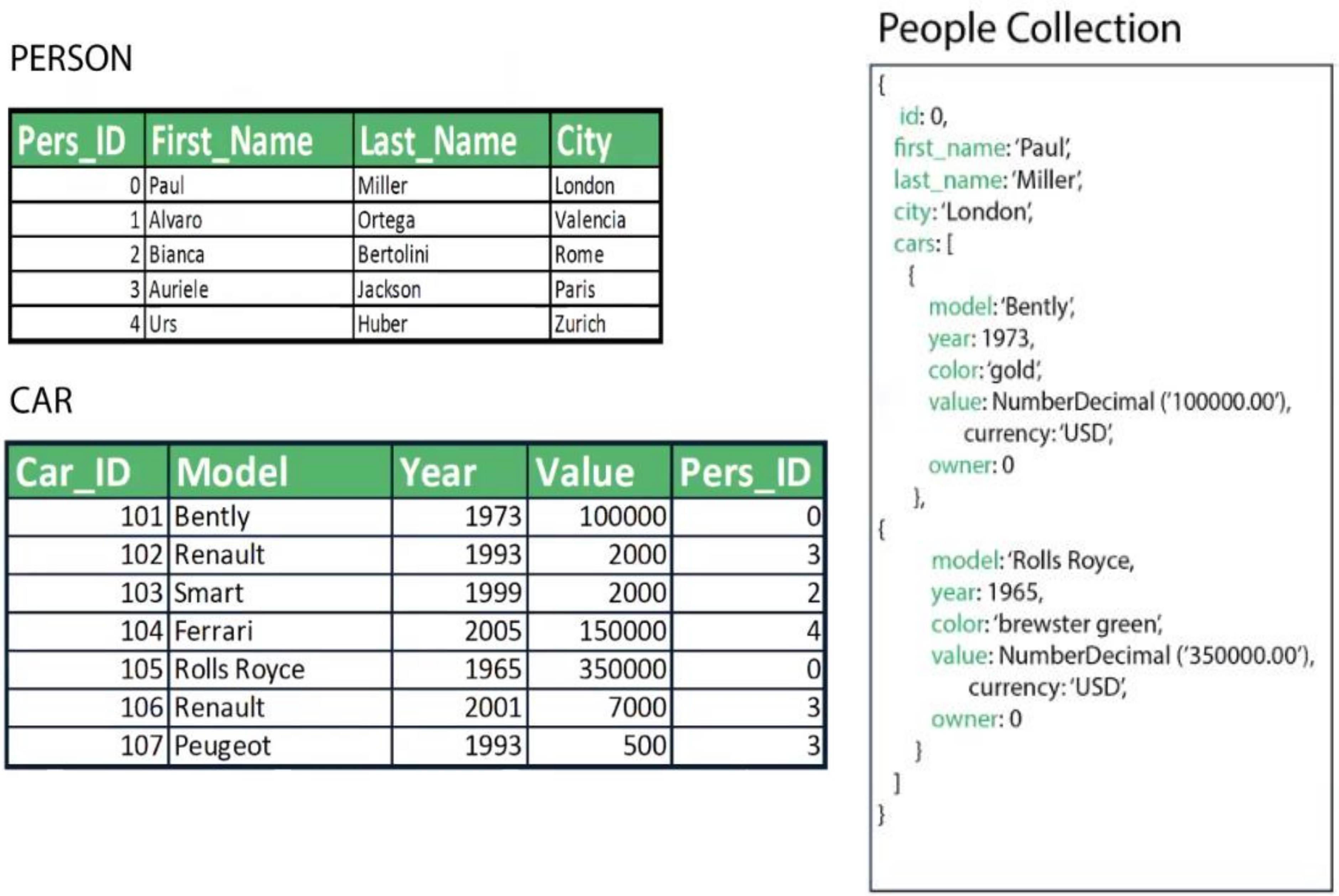


*** MongoDB does not support duplicate field names**

document

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

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
    }
  ]
}
```

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

Note: Always give --dbpath in ""

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

- 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:\database" --bind_ip_all --journal
mongod --dbpath "c:\database" --bind_ip stp10 --journal
mongod --dbpath "c:\database" --bind_ip 192.168.100.20 --journal
mongod --dbpath="c:\database" --bind_ip=192.168.100.20 –journal
mongod –auth --dbpath="c:\database" --bind_ip=192.168.100.20 --journal
mongod --storageEngine inMemory --dbpath="d:\tmp" --bind_ip=192.168.100.20
```

must be empty folder

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

```
mongo "192.168.100.20:27017/primaryDB"
mongo --host 192.168.100.20 --port 27017
mongo --host 192.168.100.20 --port 27017 primaryDB
mongo --host=192.168.100.20 --port=27017 primaryDB
mongo --host=192.168.100.20 --port=27017 –u user01 -p user01 --authenticationDatabase
primaryDB
```

```
•   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 } );
  ```

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()

```
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>]*" > < --headerline > < --useArrayIndexFields >
```

-
  ```
  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

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>]*" > < --headerline > < --useArrayIndexFields >
```

```
_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 --headerline --useArrayIndexFields`

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()

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

db.getCollectionNames()

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

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

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

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.`

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>)
```

- `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.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`

alias name

TODO

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

pattern matching with like in

TODO

- `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'}}]);`

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

TODO

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

- `db.emp.find()[0];`
- `db.emp.find()[0].ename;`
- `db.getCollection('emp').find()[0];`
- `db.emp.find()[db.emp.find().count()-1]`

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", { 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' });`

- 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.x.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([])

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.x.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();`

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("."));
                        };
                };
        });
};
```

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.collection.updateOne()

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

db.collection.updateOne()

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.

```
db.collection.updateOne({ 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.

db.collection.updateMany()

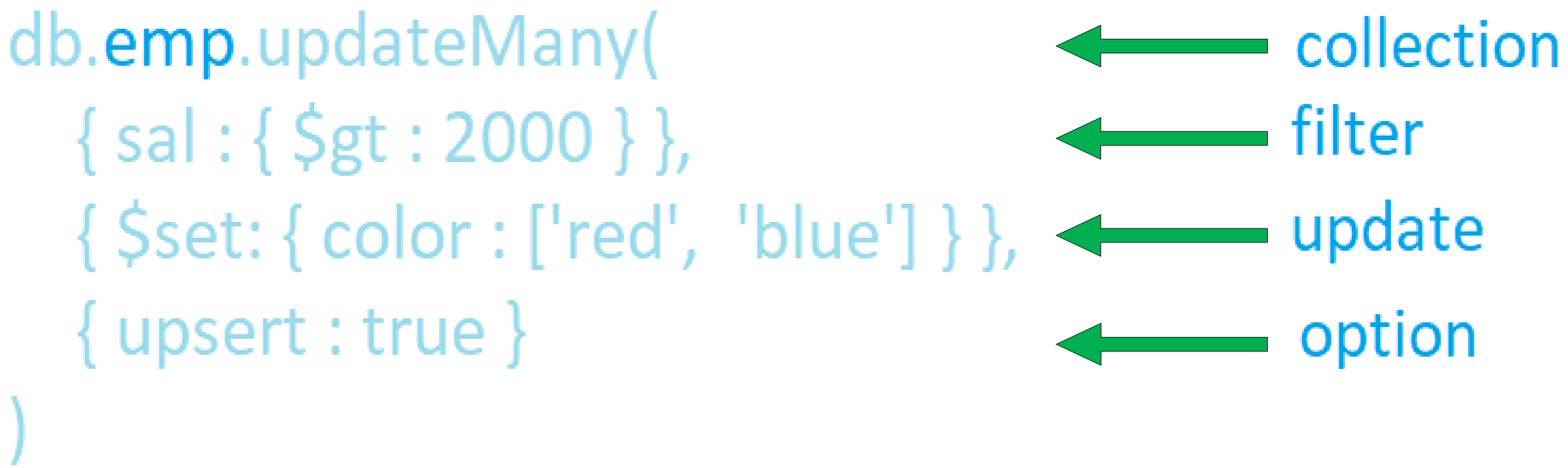
updateMany() operations can add fields to existing documents using the $set operator.

db.collection.updateMany()

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

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



```
db.emp.updateMany(
   { sal : { $gt : 2000 } },
   { $set: { color : ['red',  'blue'] } },
   { upsert : true }
)
```

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

$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>, ... } }
```

- `db.emp.updateMany({ sal: { $gt: 300 } }, { $inc: { sal: 1 } })`

$unset

$unset operator deletes a particular field.

$unset

The *$unset* operator deletes a particular field.

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

- `db.emp.update({ ename: 'saleel' }, { $unset: { comm: 0, ename: '', sal: true }})`
- `db.emp.updateOne({ ename: 'saleel' }, { $unset: { comm: 0, ename: '', sal: 0 }})`
- `db.emp.updateMany({ ename: 'saleel' }, { $unset: { comm: 0, ename: '', sal: 0 }})`

$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: [value, value, ...  ] } } }`
- `{ $pop: { <field>: <-1 | 1>, ... } }`
- `{ $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 $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: [value, value, ...  ] } } }`
- `{ $pop: { <field>: <-1 | 1>, ... } }`
- `{ $addToSet: { <field1>: <value1>, ... } }`

- `db.books.updateOne({ _id: 1 }, { $set: { publisher: 'abc publisher', founded: 1972 } });`
- `db.books.updateOne({ _id: 1 }, { $push: { languages: 'french' } });`
- `db.books.updateOne({ _id: 1 }, { $push: { email: { $each: [ "redis.com", "redis.io" ] } } });`

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 })
```

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 document 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.

$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	$out NEW Collection								

aggregate()

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

db.collection.aggregate()

stages *All stages are independent.*

$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	$out NEW Collection								

```
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.

```
'$<fieldName>'
```

Stage Operators	
`$match`	`$sort`
`$project`	`$limit`
`$addFields`	`$skip`
`$sample`	`$count`
`$group`	`$unset`
`$match`	`$out`
`$unwind`	

$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.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.

$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

TODO

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

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

$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.
```

$addFields or $set

TODO

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

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

$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 } } ])`

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

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

TODO

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

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

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

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

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

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

date operators

date operators

TODO

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

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

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

A.P.J. Abdul Kalam