A day without new knowledge is a lost day.

Database Technologies - MongoDB

Link1 Link2

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

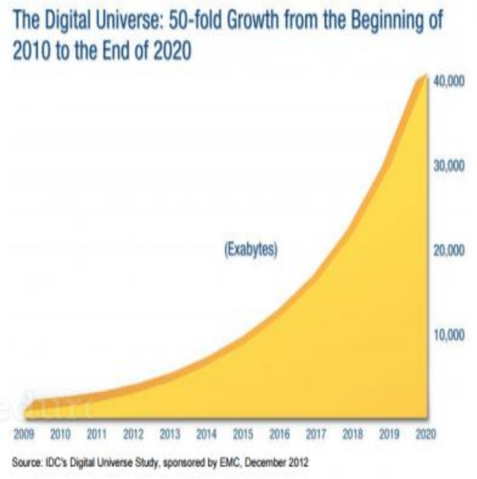
Big data is often characterized by the 3Vs: the extreme ***VOLUME*** of data, the wide ***VARIETY*** of data and the ***VELOCITY*** at which the data must be processed.

Link1 **Link2**

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.



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.

Web Logs
Images
Sensor Data
Audios
Videos

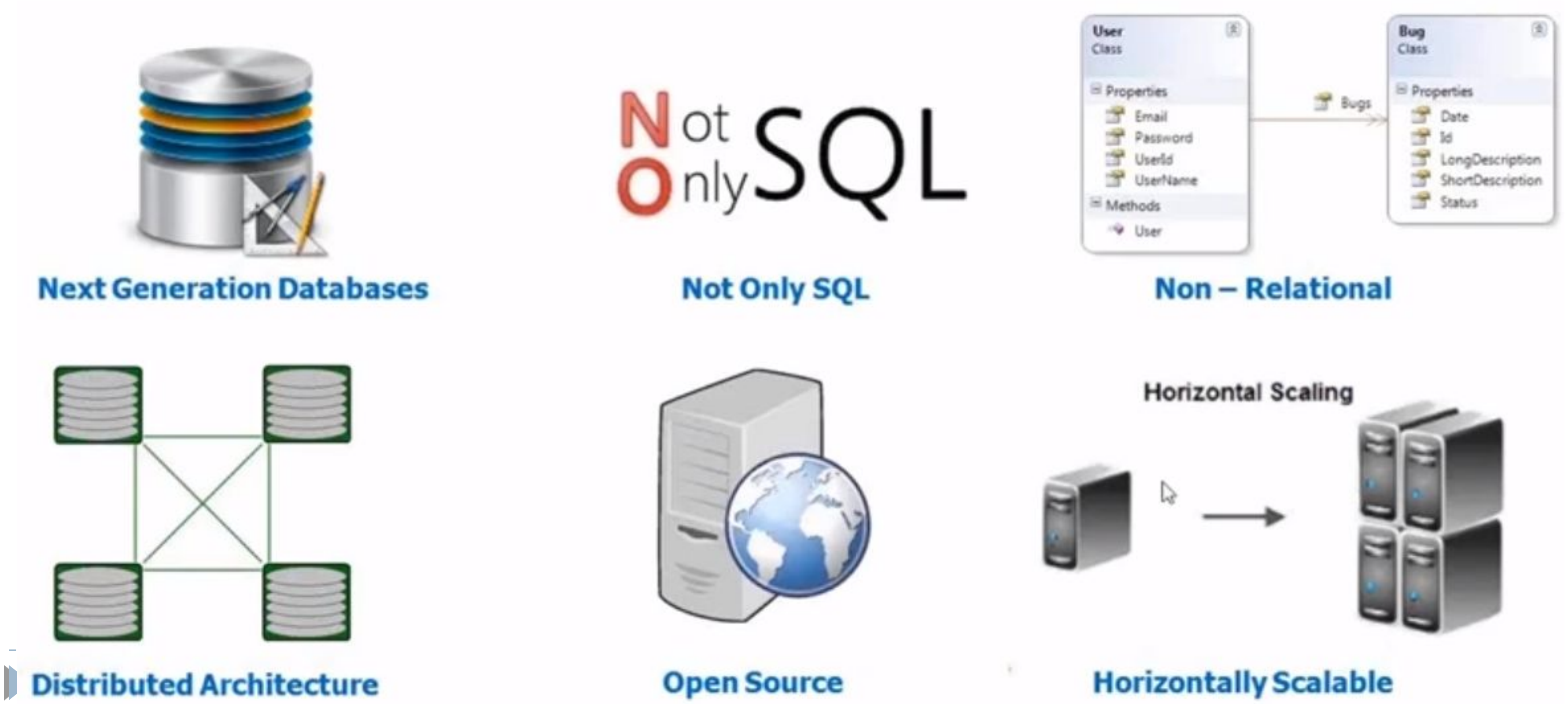
Link1 Link2

Horizontal scaling means that you scale by adding more machines into your pool of resources.

Vertical scaling means that you scale by adding more powerfull hardware / resources to an existing machine.

why NoSQL

NoSQL database are primarily called as non-relational database.



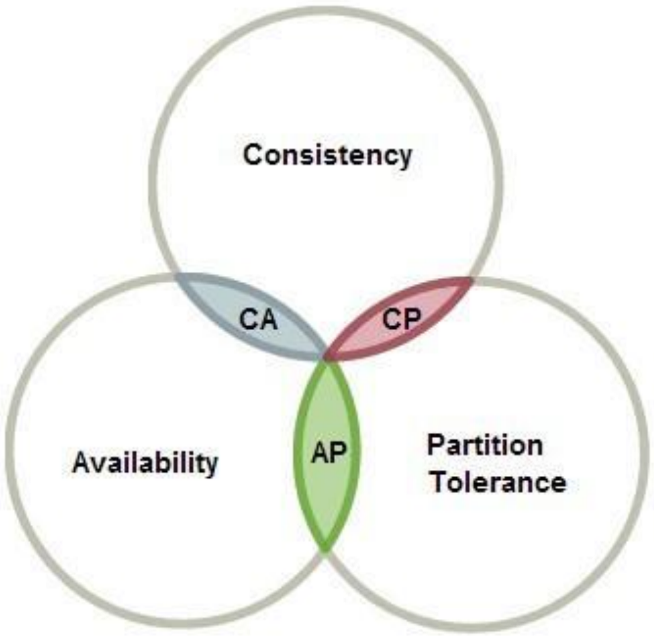
NoSQL Categories

NoSQL Categories

There are 4 basic types of NoSQL databases.

Key-value stores	Redis, Riak
Column-oriented	HBase, Cassandra
Document oriented	MongoDB, CouchDB
Graph	Neo4j, Infinite Graph

What is CAP Theorem?



CAP theorem states that any database system can only attain two out of following states which is ***Consistency, Availability and Partition Tolerance***.

- ***Consistency***: Any changes to a particular record stored in database, in form of inserts, updates or deletes is seen as it is, by other users accessing that record at that particular time.
- ***Availability***: The system continues to work and serve data inspite of node failures.
- ***Partition Tolerance***: The database system could be stored based on distributed architecture such as Hadoop (HDFS).

Link1 Link2

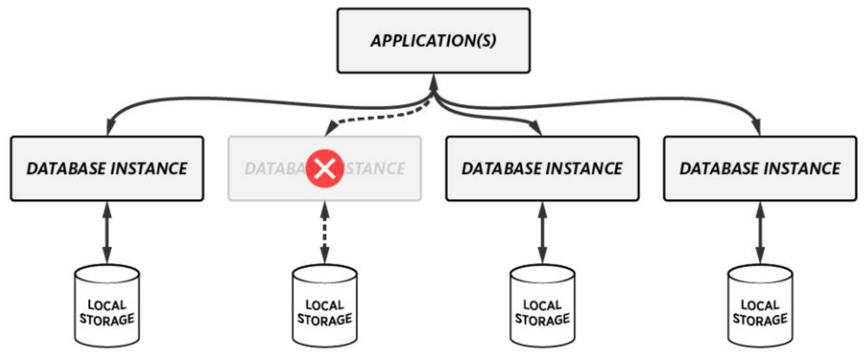
NoSQL databases are used in real-time web applications and big data and their use are increasing over time. NoSQL systems are also sometimes called Not only SQL.

NoSQL

MongoDB is scalable, open-source, high-perform, document-oriented database. **NoSQL** database are primarily called as **non-relational database**.

Figure 9

NoSQL – If an instance fails, the application can send requests to a different one.



Link1 **Link2**

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.
- If the data is not structured.
- Support of Constraints and Joins is not required.

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.

- SQL databases stores data in form of tables which consists of n number of rows of data. NoSQL databases are document based, key-value pairs, or wide-column stores.
- 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 manipulating the data. In NoSQL database, queries are focused on collection of documents.

Types of Data

Structured

The data that can be stored and processed in a fixed format is called as Structured Data. Data stored in RDBMS is 'structured'. SQL is often used to manage such kind of Data.

first_name	last_name	order_id	order_total
Saleel	Bagde	123456	12.34
Sharmin	Bagde	198765	98.76
Vrushali	Bagde	155775	127.75

Types 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, but it has some properties like tags and other markers to separate elements that makes it easier to analyze. XML files or JSON documents are examples of semi-structured data.

```
[
    {
         first_name : "Saleel",
         last_name : "Bagde",
         order_id : 123456,
         order_total : 12.34
     },
     {
         first_name : "Sharmin",
         last_name : "Bagde",
         order_id : 198765,
         order_total : 98.76
     }
]
```

Link1 Link2

Types of 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. The unstructured data is growing quicker than others, experts say that 80 percent of the data in an organization are unstructured.

MongoDB

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

Core MongoDB Operations (CRUD)

CRUD stands for ***create***, ***read***, ***update***, and ***delete*** which are the four core database operations.

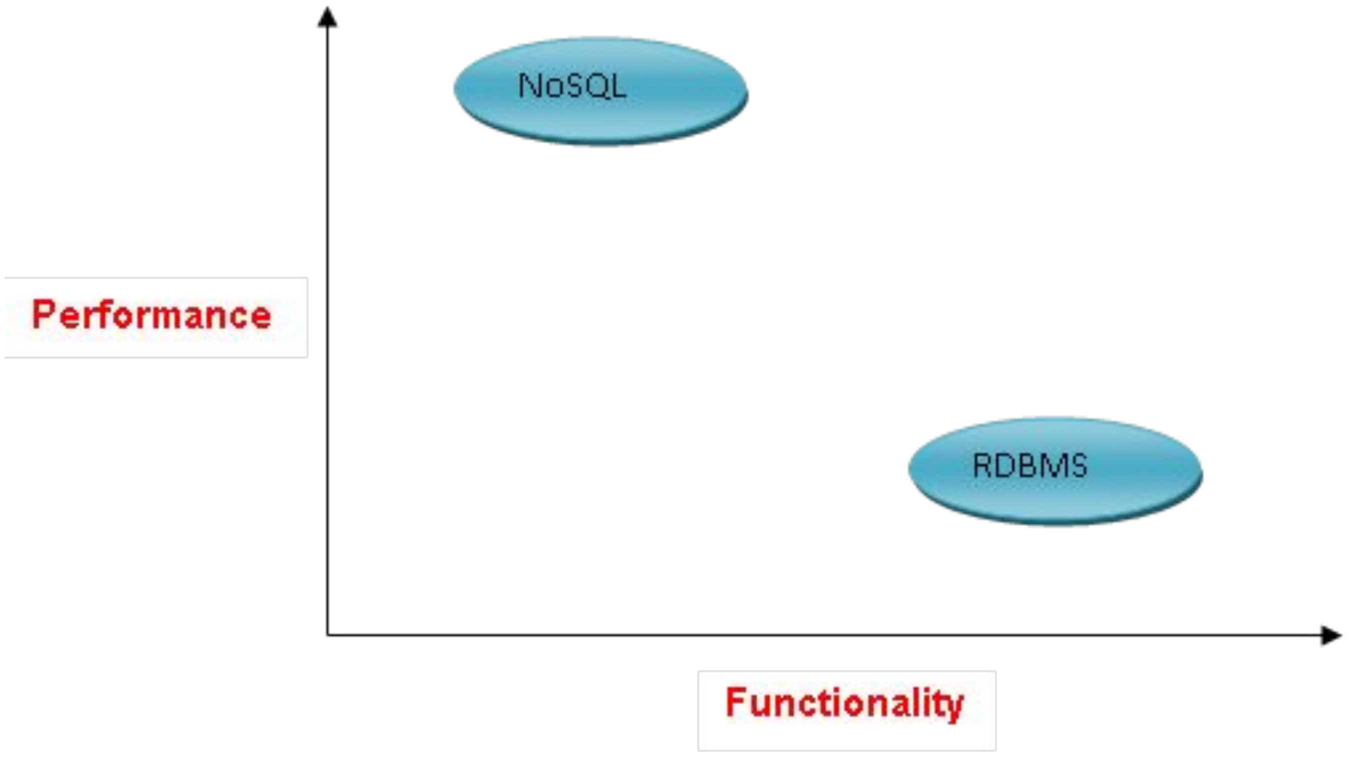
RDBMS	MongoDB
database	database
tables	collections
rows	Documents or BSON document
column	Field

Link1 Link2

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.



Link1 Link2

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
    }
  ]
}
```

Link1 Link2

document

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:
value1,
   field2:
value2,
   field3:
value3,
   ...
   fieldN:
valueN
}
```

The primary key _*id* is automatically added if _*id* field is not specified.

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**.

- 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
```

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

```
mongo "192.168.100.20:27017/db1"
mongo --host "192.168.100.20" --port "27017"
mongo --host "192.168.100.20" --port "27017" primaryDB
```

- db.version()   // Returns mongoDB version.
- db.getMongo();   // connection to 192.168.100.20:27017
- db.hostInfo()   // Returns a document with information about the underlying system that the mongod runs on.
- getHostName()   // Computer name [stp5]

comparison operator

Link1 Link2

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}
}
```

$ne

```
{ field: {$in: [<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.

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

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

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

Link1 Link2

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'}}})

Link1 Link2

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.getName()
```

```
db
db.getName()      // Returns: the current database name.
```

Link1 Link2

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.

Link1 Link2

use database

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

```
use <d
b>
```

```
use db1
```

Link1 Link2

mongoimport

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

mongoimport

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

```
mongoimport < --host > < --port > < --db > < --collection > < --file>
```

```
mongoimport  --host 192.168.0.3 --port 27017  --db db1 --collection emp
--file "d:\emp.json"
```

Link1 Link2

mongoimport

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

```
mongoimport < --host > < --port > < --db > < --type csv >
< --collection > < --file> < --fields "Field-List">
```

```
mongoimport --host 192.168.100.20 --port 27017  --db db1 --collection o
--type csv  --file "d:\o.csv" --fields "EMPNO, ENAME, JOB, MGR,HIREDATE,
SAL, COMM, DEPTNO,BONUSID, USERNAME, PWD"
```

```
mongoimport --db db1 --collection o --type csv  --file "d:\o.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()"
```

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 > < --out >
```

```
mongoexport  --host "192.168.0.3" --port 27017  --db "db1" --collection emp –out "d:\e.json"
```

Link1 Link2

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()

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

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

show collection

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.

Link1 Link2

db.collection.isCapped()

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

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

```
db.log.isCapped();
```

Link1 Link2

db.getCollection()

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

db.getCollection()

TODO

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

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

db.getSiblingDB()

To access another database without switching databases.

Link1 Link2

db.getSiblingDB()

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

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

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

Link1 Link2

db.collection.renameCollection()

Renames a collection.

Link1 Link2

db.collection.renameCollection()

TODO

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

```
db.emp.renameCollection('e', 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()

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

```
db.collection.drop(<options>)
```

```
db.emp.drop();
```

Method

.pretty()

db.collection.find()

The find() method always returns the _id field unless you specify _id: 0 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.

db.collection.find()

TODO

A projection cannot contain both include and exclude specifications, except for the exclusion of the _id field.

```
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 ({}).

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.

Projection

```
{ field1: <value>, field2: <value> ... }
```

- ***1*** or ***true*** to include the field in the return documents.
- ***0*** or ***false*** to exclude the field.

db.collection.find()

TODO

```
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:1, job: true});

db.emp.find({sal:{ $gt:4}})
```

Link1 Link2

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({ })

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

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

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

Link1 Link2

limit

Use the 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()

Use the ***limit()*** method to 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);
```

Link1 Link2

skip

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

db.collection.find().skip()

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

```
cursor.skip(<offset_number>)
db ['emp'].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()

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_name'].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()

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)
 }
```

Link1 Link2

db.collection.count[Documents]()

TODO

db.collection.count[Documents]()

TODO

```
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

The find() method always returns the _id field unless you specify _id: 0 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 ['emp'].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()()

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

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

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

Link1 Link2

db.collection.insert()

Inserts a document or documents into a collection.

Link1 Link2

db.collection.insert()

Inserts a document or documents into a collection.

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

```
db.e.insert({})

db.e.insert({ ename: 'a', job: 'abc', salary: 2000 })

db.e.insert([ { ename: 'x'} , { ename: 'y' } ])      // for multiple documents.
```

Link1 Link2

db.collection.insertOne()

Inserts a document into a collection.

Link1 Link2

db.collection.insertOne()

Inserts a document or documents into a collection.

```
db.collection.insertOne({<do
cument>})
```

```
db.emp.insertOne({ ename: 'x', job: 'pqr', salary: 2000 })
```

db.collection.insertMany()

Inserts multiple documents into a collection.

Link1 Link2

db.collection.insertMany()

Inserts a document or documents into a collection.

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

```
db.e.insertMany([ { ename: 'x', salary: 2000}, { ename : 'y', job: 'hr' } ])
```

Link1 Link2

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);
```

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.

Link1 Link2

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")
```

Link1 Link2

javascript function

TODO

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

Link1 Link2

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.

```
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: 'abc1' }, { job: 'sales' }, { upsert: true } );

db.emp.update({ job: 'bbc' }, { $set: { job:'abc' } }, { upsert : true,  multi: true });

db.emp.update({ ename: 'saleel' }, { $set : { size: 'small', color: ['red', 'blue'] } },
{ multi: true } );
```

db.collection.updateOne()

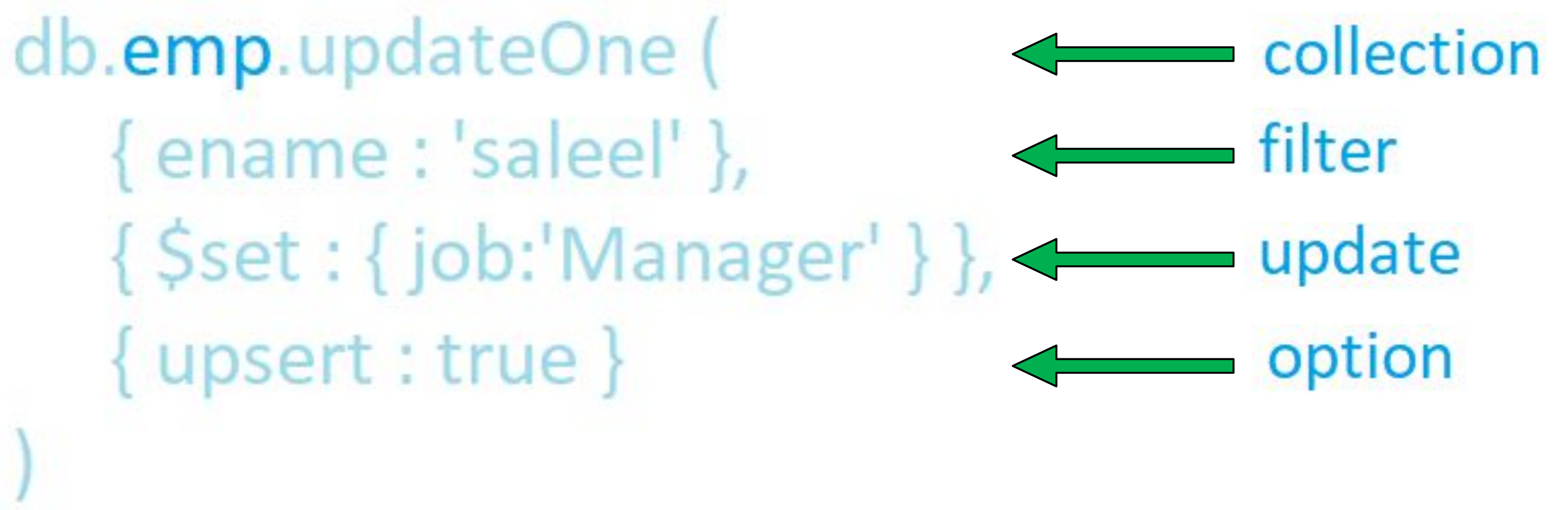
updateOne() operations can add fields to existing documents using the $set operator.

db.collection.updateOne()

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

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

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



```
db.emp.updateOne (
    { ename : 'saleel' },
    { $set : { job:'Manager' } },
    { upsert : true }
)
```

```
db.emp.updateOne({ ename : 'saleel1' }, { $set : { job : 'A' } })
db.emp.updateOne({ename : 'saleel2' }, { $set : { job : 'A' } }, { upsert: true })
```

Link1 Link2

db.collection.updateMany()

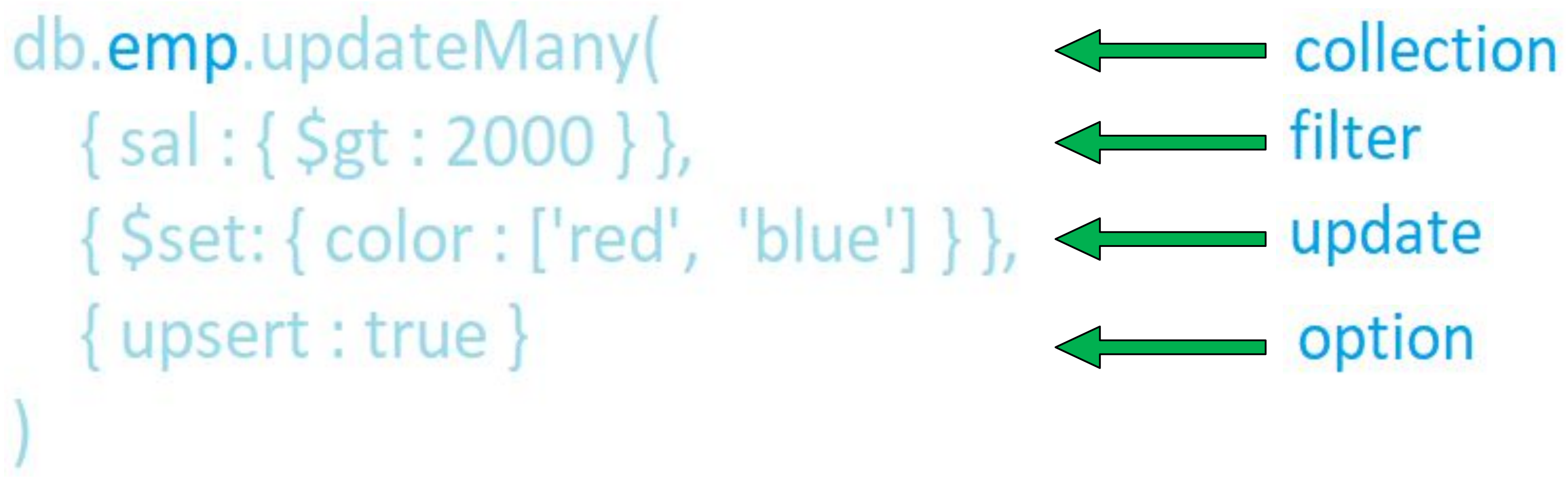
updateMany() operations can add fields to existing documents using the $set operator.

db.collection.updateMany()

Updates multiple documents within the collection based on 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'] } }, { upsert: true });

Link1 Link2

$inc

The $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 } })
```

Link1 Link2

$unset

The $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: 0}})

db.emp.updateOne({ename: 'saleel'}, {$unset: {comm: 0, ename: '', sal: 0}})

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

db.collection.findOneAndUpdate()

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

Link1 Link2

db.collection.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 })
```

Link1 Link2

db.collection.replaceOne()

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

db.collection.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 })
```

Link1 Link2

db.collection.deleteOne()

Removes a single document from a collection.

Link1 Link2

db.collection.deleteOne()

Removes a single document from a collection. Specify an empty document { } to delete the first document returned in the collection.

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

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

```
db.emp.deleteOne({job: 'manager'})
```

db.collection.deleteMany()

Removes all documents that match the filter from a collection.

Link1 Link2

db.collection.deleteMany()

Removes **all** documents that match the filter from a collection.

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

```
db.emp.deleteMany({});

db.emp.deleteMany({job: 'manager'})
```

Link1 Link2

db.collection.findOneAndDelete()

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

Link1 Link2

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}})
```

All stages are independent.

stages

$match	$project	$unwind	$group	$match	$sort	$limit	$skip
WHERE clause	SELECT clause	PIVOT an array	GROUP BY clause	HAVING clause	ORDER BY clause	TOP clause	

aggregate()

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

Link1 Link2

db.collection.aggregate()

stages

$match	$project	$unwind	$group	$match	$sort	$limit	$skip
WHERE	SELECT	PIVOT	GROUP BY	HAVING	ORDER BY	TOP	
clause	clause	an array	clause	clause	clause	clause	

```
db.collection.aggregate( [ { <stage1> }, { <stage2> }, { <stage3> }
... , { <stageN> } ] )
```

db.emp.aggregate([])

Link1 Link2

aggregation <stageOperators> and aggregation <expression>

Each sage starts with stage operator.

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

{ $match : { job: 'manager' } }

{ $group : { _id : '$job' } }

Stage Operators	
$match	$sort
$project	$limit
$unwind	$skip
$group	$count
$match	

Each aggregation expression starts with $ sign.

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

Link1 Link2

$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: {_id: false, sal: true, comm: true } } ])

db.emp.aggregate ([ {$project: {sal: true, sm: {$sum: '$sal'} } } ])

db.emp.aggregate ([ {$project: {_id: false, sal: true, comm: true, xx: {$max:
['$sal', '$comm'] } } } ])

db.emp.aggregate([ {$project :{_id: false, indexID: true, favouriteFruit:
{$size: '$favouriteFruit'} } } ])
```

arithmetic expression operators

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>' }`

```
db.emp.aggregate ([ {$project : { op: { $trunc: "$sal" } } } ])
```

```
db.emp.aggregate([ {$project: { sal: true,  op : { $add: ['$sal', 1000] } } } ])
```

Link1 Link2

$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: { $concat:[ '$<expression1>', '$<expression2>', ... ] }`
`x: { $substr: [ <string>, <start>, <length> ] }`
`x: { $size: '$<expression>' }`
`x: { $arrayElemAt: ['$<array>', <idx> ] }`

Link1 Link2

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

```
db.emp.aggregate([ { $project: {comm : { $ifNull: ['$comm', 'NA'] } } } ])

db.emp.aggregate([ { $project: {sal: true, comm: true, "Gross Salary":  { $add: ['$sal', { $ifNull: [ '$comm', 0] } ] } } } ])

db.emp.aggregate([ { $project: { ename : { $toUpper : '$ename'} } } ])

db.emp.aggregate([ { $project: { ename : { $toLower : '$ename'} } } ])

db.emp.aggregate([ { $project: { ename : { $concat : ['$ename', '$job'] } } } ])

db.emp.aggregate([ { $project: { favouriteFruit: { $size: '$favouriteFruit'} } }])

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

Link1 Link2

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'} } } ])
```

$unwind

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

Link1 Link2

$unwind

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

```
{ $unwind: '$<field path>' }
```

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

$group

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

$group

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

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

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

```
db.emp.aggregate([ {$group: {_id: null, count: {$sum: 1} } } ])

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

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

$group on multiple fields

$ group

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

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

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

Link1 Link2

$sort

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

$sort

TODO

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

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

$limit

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

Link1 Link2

$limit

TODO

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

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

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

$skip

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

$skip

TODO

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

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

$count

Passes a document to the next stage that contains a count of the number of documents input to the stage.

$count

TODO

```
{ $count: <string> }
```

Database Security and Authentication

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

db.getUser() / db.getUsers()

db.getUser() / db.getUsers()

Returns user information for a specified user.

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

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

Returns information for all the users in the database.

```
db.getUsers()
```

```
db.getUsers()
```

db.createUser

db.createUser

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

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

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

Link1 Link2

db.grantRolesToUser / db.revokeRolesFromUser

db.grantRolesToUser
db.revokeRolesFromUser

TODO

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

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

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

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

The role provides the following actions on those collections

1. Read :- [dbStats, find, listIndexes, listCollections, etc...]
2. readWrite :- [collStats, convertToCapped, createCollection, dbHash, dbStats, dropCollection, createIndex, dropIndex, find, insert, killCursors, listIndexes, listCollections, remove, renameCollectionSameDB, update]
3. userAdmin :- [TODO]
4. readAnyDatabase :- [TODO]
5. readWriteAnyDatabase :- [TODO]

db.dropAllUser() / db.dropUser()

db.dropUser() / db.dropAllUsers()

Removes the user from the current database.

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

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

Removes all users from the current database.

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

```
db.dropAllUser()
```

IMP

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

$$7 * 4 = 28$$
$$7 + 7 + 7 + 7 = 28$$

$$5 * 6 = 30$$
$$5 + 5 + 5 + 5 + 5 + 5 = 30$$

2. Square

$$1^2 = (1) = 1$$

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

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

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

Link1 Link2

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

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

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

"If someone is strong enough to bring you down, show them you are strong enough to get up."

Link1 Link2

```
create table book (id raw(16) primary key, data clob check(data is json));

select book.*
    from books,
    json_table(data,'$'
    columns(isbn   varchar2(20) path '$.isbn',
        title  varchar2(20) path '$.title',
        price  varchar2(10) path '$.price',
        author varchar2(20) path '$.author',
        phone  varchar2(10) path '$.phone')) book
```