
mongoDB

Lecture 1

Course Objectives

Learn how to use Mongo database with the main operations and how to deal with the mongo shell.

Course Prerequisites

- **Database Fundamentals.**
- **Familiarity with Linux terminal.**
- **Basic Knowledge of JavaScript.**

Agenda

- Types of Databases.
- SQL VS. NoSQL Database.
- Why NoSQL?
- Why MongoDB ?
- MongoDB.
- Getting Started.
- MongoDB CRUD Operations.
- Mongo query operators (Comparison, Logical, Arrays, Element)
- Lab 1

Types of Databases

- **RDBMS (Relational Database Management System).**
  Example: MySql, Oracle, Sqlite, Postgres.

- **NoSQL.**
  Example: MongoDB, Cassandra, Hbase.

SQL VS. NoSQL Database

SQL

- RDBMS (Relational Databases).
- Structured query language for defining and manipulating the data.
- Table based.
- Predefined schema.

NoSQL

- Non-relational or distributed database.
- Unstructured query language (syntax varies from database to database).
- Document based, key-value pairs, graph databases or wide-column stores.
- Dynamic schema for unstructured data.

mongoDB

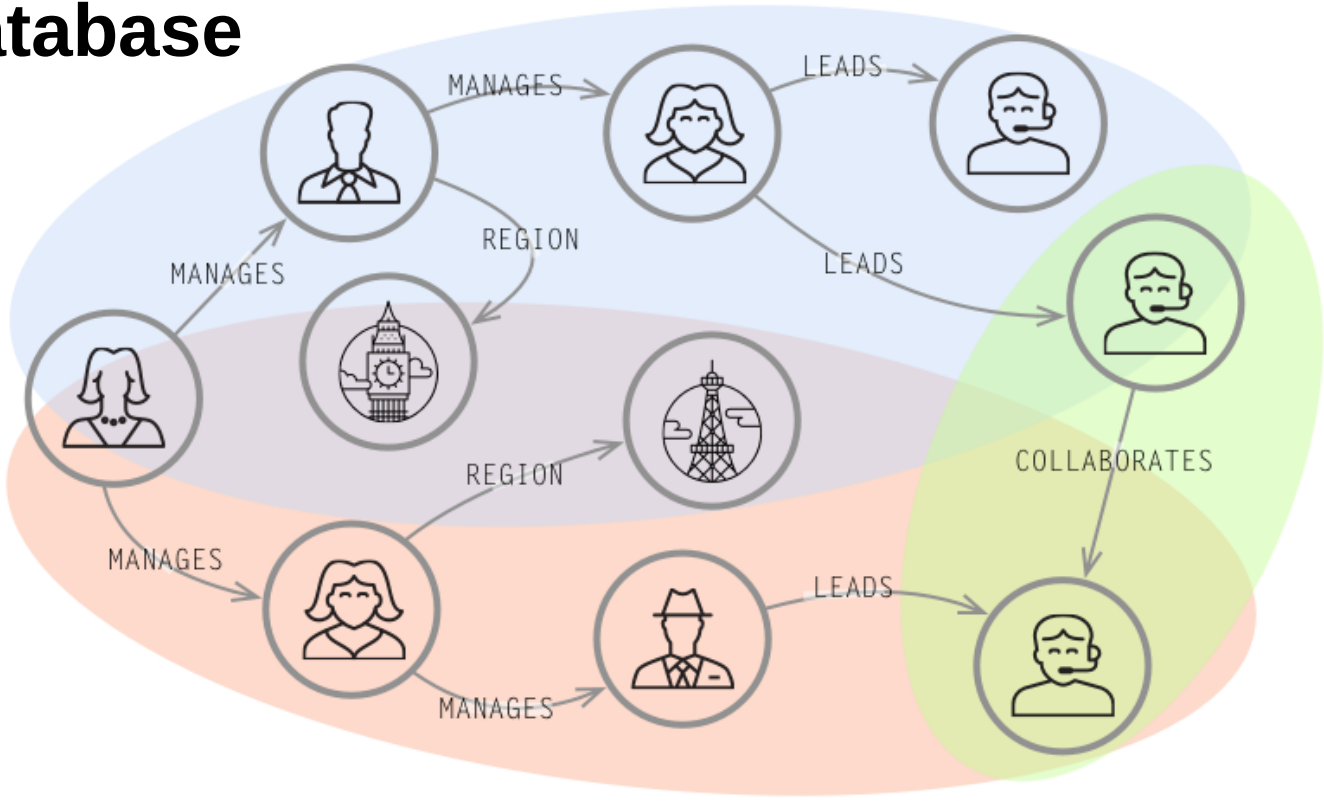
NoSQL Database

- Key-Value
- wide-column
- Graph database
- Document based

mongoDB

NoSQL Database

Graph database



NoSQL Database

Relational

Person:

Pers_ID	Surname	First_Name	City
0	Miller	Paul	London
1	Ortega	Alvaro	Valencia
2	Huber	Urs	Zurich
3	Blanc	Gaston	Paris
4	Bertolini	Fabrizio	Rom

no relation

Car:

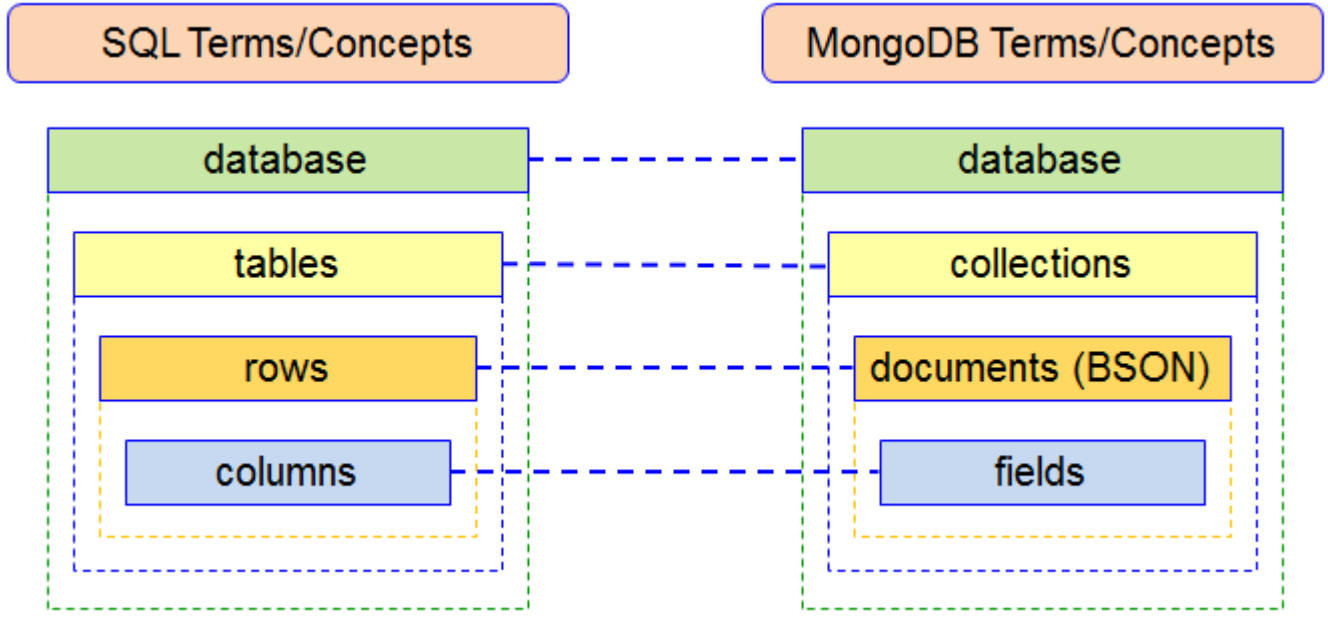
Car_ID	Model	Year	Value	Pers_ID
101	Bentley	1973	100000	0
102	Rolls Royce	1965	330000	0
103	Peugeot	1993	500	3
104	Ferrari	2005	150000	4
105	Renault	1998	2000	3
106	Renault	2001	7000	3
107	Smart	1999	2000	2

MongoDB Document

```
{
  first_name: 'Paul',
  surname: 'Miller'
  city: 'London',
  location: [45.123,47.232],
  cars: [
   { model: 'Bentley',
     year: 1973,
     value: 100000, ... },
   { model: 'Rolls Royce',
     year: 1965,
     value: 330000, ... }
  ]
}
```

mongoDB

SQL VS. NoSQL Database



SQL VS. NoSQL Database

SQL

Vertically scalable (You can manage increasing load by increasing the CPU, RAM, etc, on a single server).

NoSQL

Horizontally scalable (You can just add few more servers easily in your NoSQL database infrastructure to handle the large traffic).

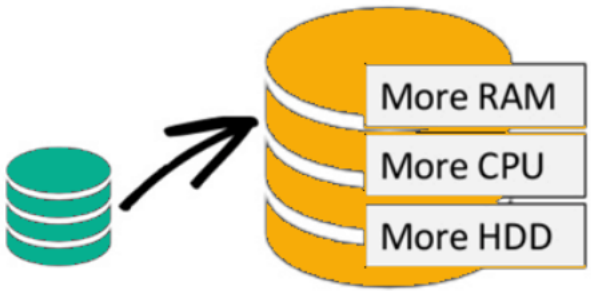
Sharding
Replication

mongoDB

SQL VS. NoSQL Database

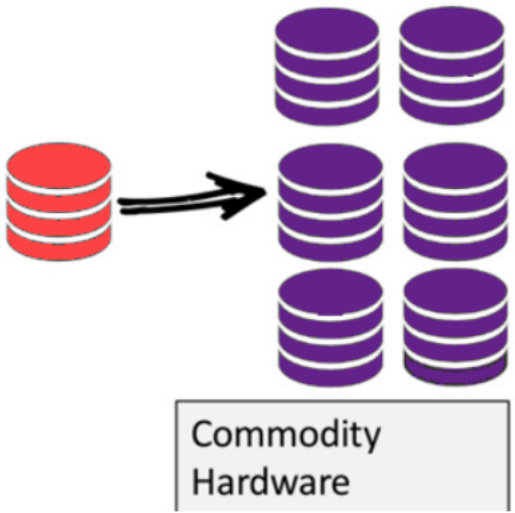
SQL

Scale-Up (*vertical* scaling):



NoSQL

Scale-Out (*horizontal* scaling):



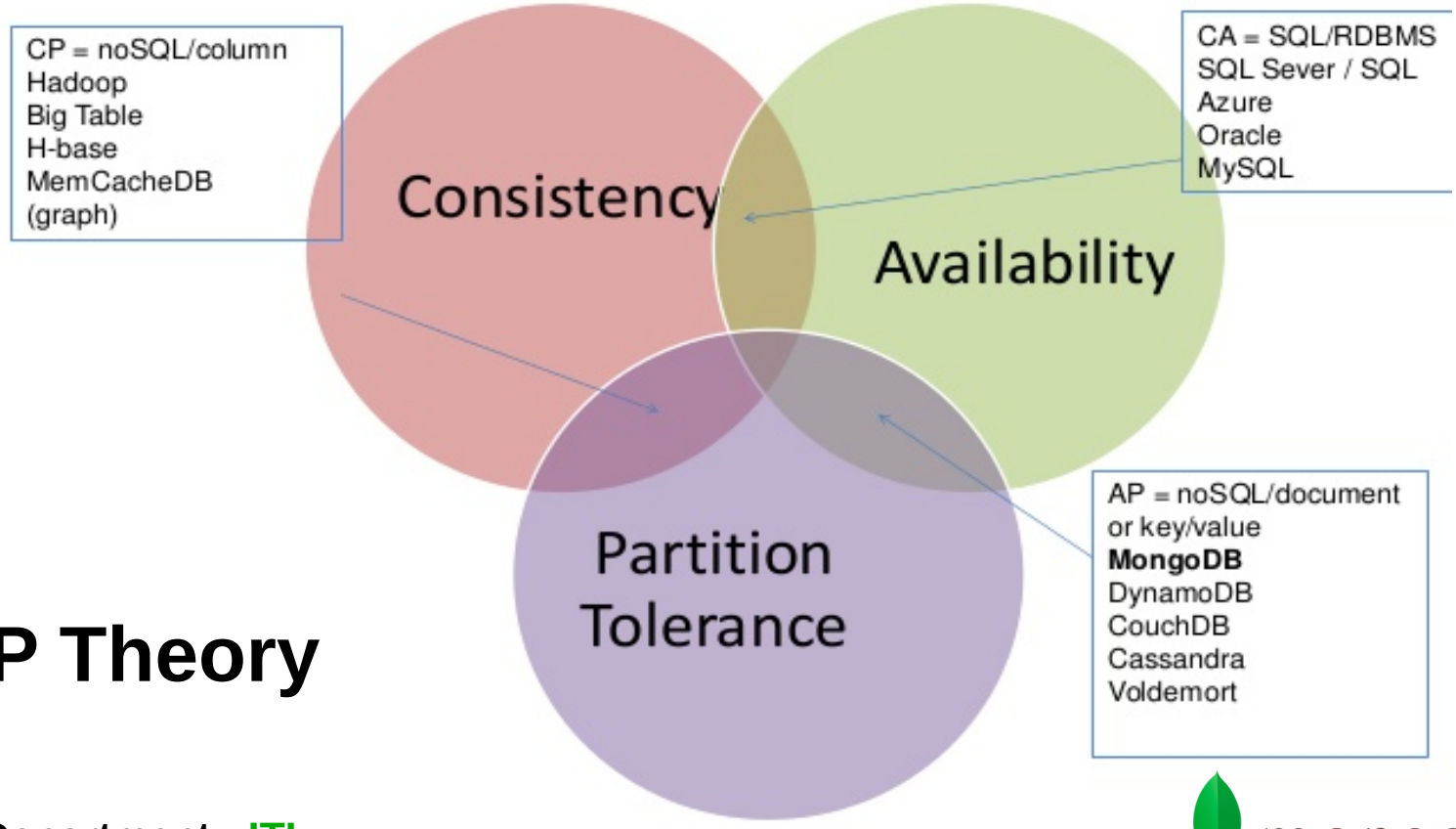
Why NoSQL ?

- Big Data is one of the key forces driving the growth and popularity of NoSQL for business.

A Big Data project is normally typified by:

- **High data velocity:** lots of data coming in very quickly, possibly from different locations.
- **Data variety:** storage of data that is structured, semi-structured and unstructured.
- **Data volume:** data that involves many terabytes or petabytes in size.
- **Data complexity:** data that is stored and managed in different locations or data centers.

Why NoSQL ?



CAP Theory

Why NoSQL ?

Scenarios where NoSQL SHOULD be used:

- Your relational database will not scale to your traffic at an acceptable cost.
- In a NoSQL database, there is no fixed schema and no joins. NoSQL can take advantage of “scaling out”. Scaling out refers to spreading the load over many commodity systems.
- It’s useful for creating prototypes or fast applications as it provides a tool to develop new features easily.
- You have local data transactions which do not have to be very durable. e.g. “liking” items on websites.
- Agile sprints, quick iteration, and frequent code pushes.
- Object-oriented programming that is easy to use and flexible.

mongoDB

Why NoSQL ?

Scenarios where NoSQL SHOULD NOT be used:

- **It cannot necessarily guarantee the ACID (Atomicity, Consistancy, Isolation, Durability) properties for your transactions.**
- **Normally an interface is provided for storing your data. Do not try to use a complicated query in that interface.**
- **The developer should always keep in mind that NoSQL database is not built on tables and usually doesn't use structured query language.**
- **If consistency is mandatory and there will be no drastic changes in terms of the data volume.**

Why MongoDB ?

- **It is Open Source.**
- **High Write Load :**
  MongoDB by default prefers high insert rate over transaction safety. If you need to load tons of data lines with a low business value for each one.
- **High Availability in an Unreliable Environment :**
  Setting replicaSet (set of servers that act as Master-Slaves) is easy and fast. Moreover, recovery from a node (or a data center) failure is instant, safe and automatic.
- **You need to Grow Big:**
  Databases scaling is hard (a single MySQL table performance will degrade when crossing the 5-10GB per table).

mongoDB

Why MongoDB ?

- **Location Based**:
  MongoDB has built in spacial functions, so finding relevant data from specific locations is fast and accurate.

- **Data Set is Going to be Big (from 1GB) and Schema is Not Stable:**
  Adding new columns to RDBMS can lock the entire database in some database, or create a major load and performance degradation in other.

- **You Don't have a DBA:**
  If you don't have a DBA, and you don't want to normalize your data and do joins, you should consider MongoDB.

Note: If you are expecting to go big, please notice that you will need to follow some best practices to avoid pitfalls.

mongoDB

MongoDB Basics

- MongoDB is a NoSQL database, Open Source, cross-platform and document-oriented database written in C++ that provides, high performance, high availability, and easy scalability.
- Database is a physical container for collections. Each database gets its own set of files on the file system.
- Collection is a group of MongoDB documents.
- Documents within a collection can have different fields.
- Document is a set of key-value pairs.

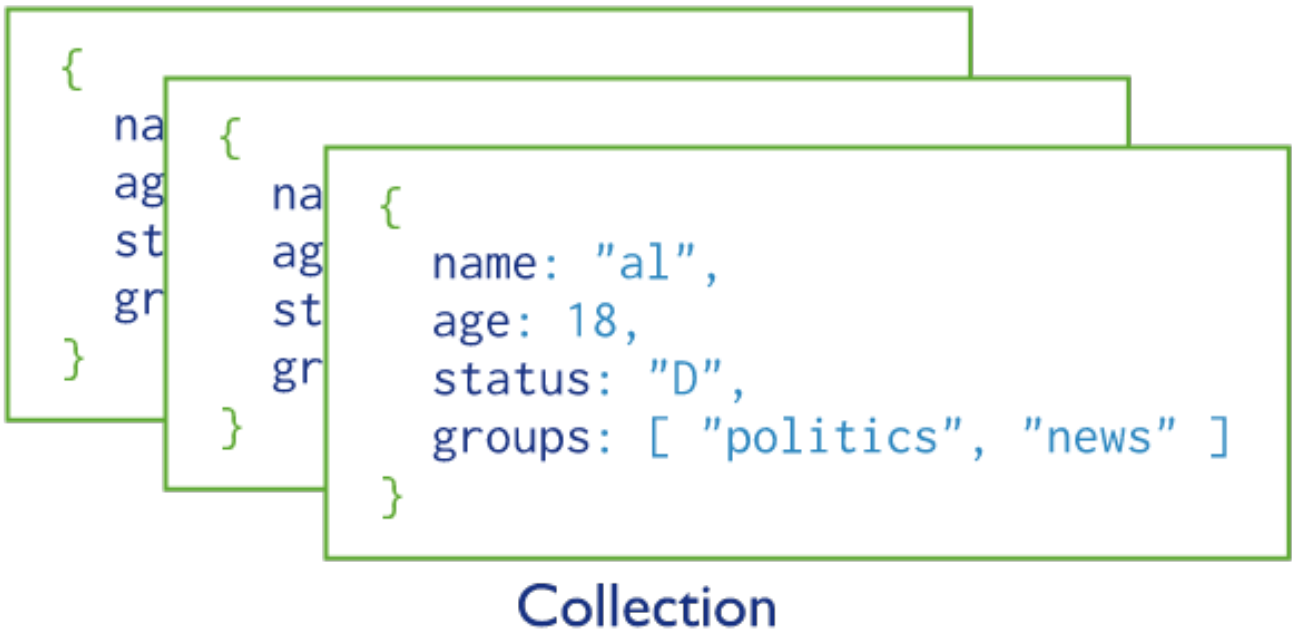
mongoDB

MongoDB Basics

- Documents have dynamic schema.
- Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection’s documents may hold different types of data.

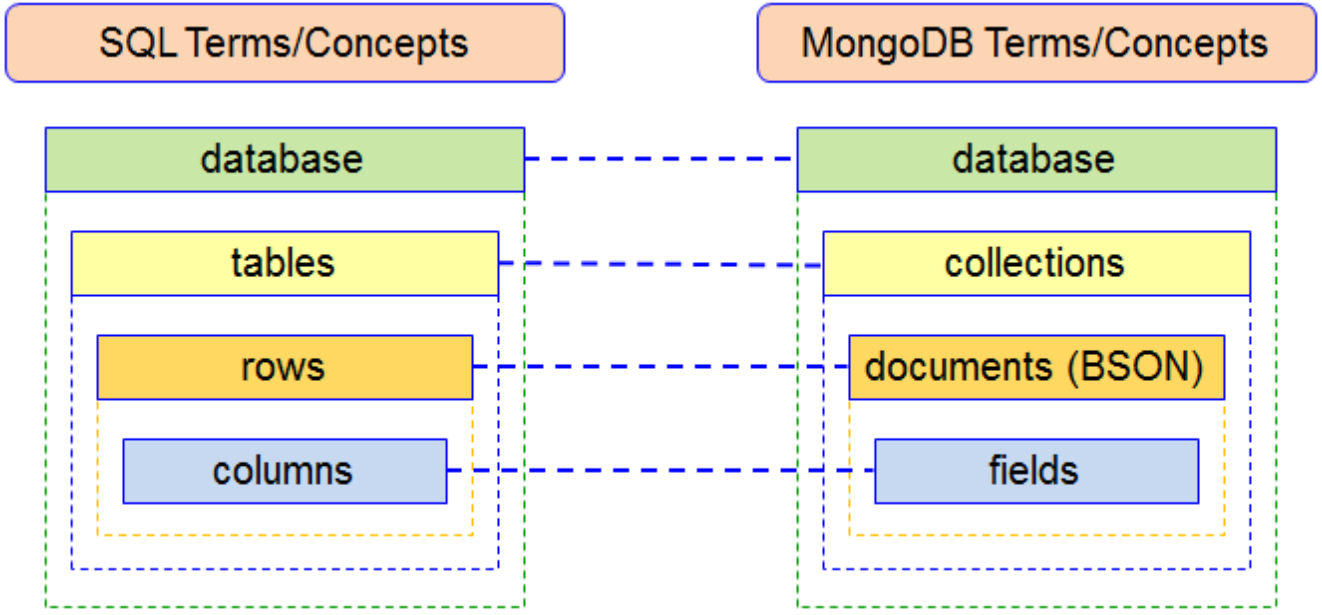
mongoDB

MongoDB Basics



mongoDB

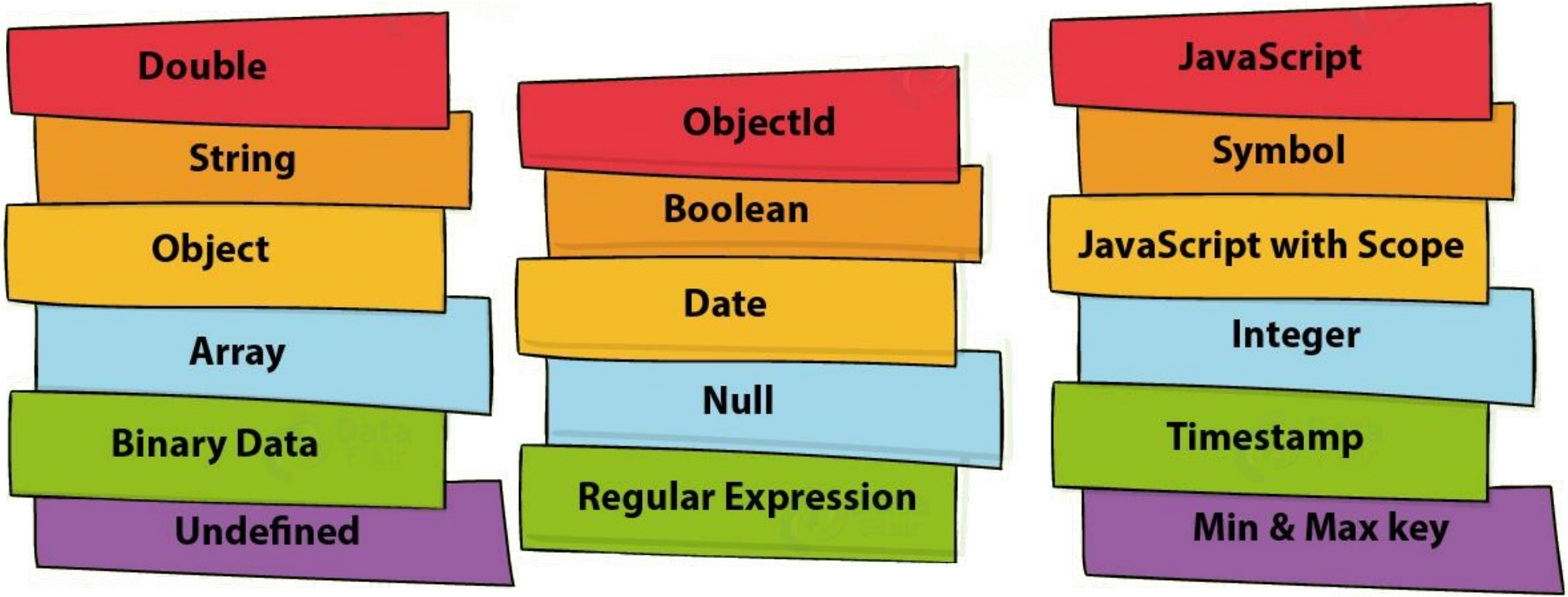
MongoDB Basics



MongoDB Document

```
{
  first_name: 'Paul',
  surname: 'Miller'
  city: 'London',
  location: [45.123,47.232],
  cars: [
   { model: 'Bentley',
     year: 1973,
     value: 100000, ... },
   { model: 'Rolls Royce',
     year: 1965,
     value: 330000, ... }
  ]
}
```

MongoDB Data types



MongoDB Data types

Double 1	{"double data type": 3.1415}
String 2	{"string data type" : “ message"}
Object 3	{"Object data type" : "This is Object", "Marks" : embedded Object}

mongoDB

MongoDB Data types

Array 4	{"Array data type": [1 ,5 ,17 ,28]}
Binary data 5	
Undefined 6	
ObjectId 7	
Boolean 8	{"Boolean data type": true}
Date 9	{"Date data type": new Date('Jun 17 2020')}

mongoDB

MongoDB Data types

- **String**: It is used to store data. It must be UTF8 valid in mongoDB.
- **Integer**: It is used to the number value. It can be 32 bit or 64 bit depending on the server you are using.
- **Boolean**: It is used to store boolean values. It just shows YES/NO values.
- **Double**: It stores floating point values.

- **Min/ Max keys** :It is used to compare a value against the lowest and highest BSON elements.
- **Arrays**: It is used to store arrays or list or multiple values into one key.

MongoDB Data types

- **String**: It is used to store data. It must be UTF8 valid in mongoDB.
- **Integer**: It is used to the number value. It can be 32 bit or 64 bit depending on the server you are using.
- **Boolean**: It is used to store boolean values. It just shows YES/NO values.
- **Double**: It stores floating point values.

- **Min/ Max keys** :It is used to compare a value against the lowest and highest BSON elements.
- **Arrays**: It is used to store arrays or list or multiple values into one key.

MongoDB Data types

- **Timestamp – ctimestamp:** This can be handy for recording when a document has been modified or added.
- **Object** : It is used for embedded documents.
- **Null**: It is used to store a Null value.
- **Symbol**: It is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
- **Date**: It is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
- **Object ID**: It is used to store the document's ID.

MongoDB Basics

```
{
"_id" : 1,
"name" : { "first" : "John", "last" : "Backus" },
"contribs" : [ "Fortran", "ALGOL", "Backus-Naur Form", "FP" ],
"awards" : [
          {
            "award" : "W.W. McDowell Award",
            "year" : 1967,
            "by" : "IEEE Computer Society"
          },
          { "award" : "Draper Prize",
            "year" : 1993,
            "by" : "National Academy of Engineering"
          }
]
}
```

mongoDB

MongoDB Basics

- If you don’t provide _id then MongoDB provides a unique id for every document.
- _id is a 12 bytes hexadecimal number which assures the uniqueness of every document.
  - first 4 bytes: for the current timestamp.
  - next 3 bytes: for machine id.
  - next 2 bytes: for process id of MongoDB server.
  - remaining 3 bytes: are simple incremental value.

MongoDB Basics

```
{
    _id: ObjectId("5099803df3f4948bd2f98391"),
    name: { first: "Alan", last: "Turing" },
    birth: new Date('Jun 23, 1912'),
    death: new Date('Jun 07, 1954'),
    contribs: [ "Turing machine", "Turing test", "Turingery" ],

 }
```

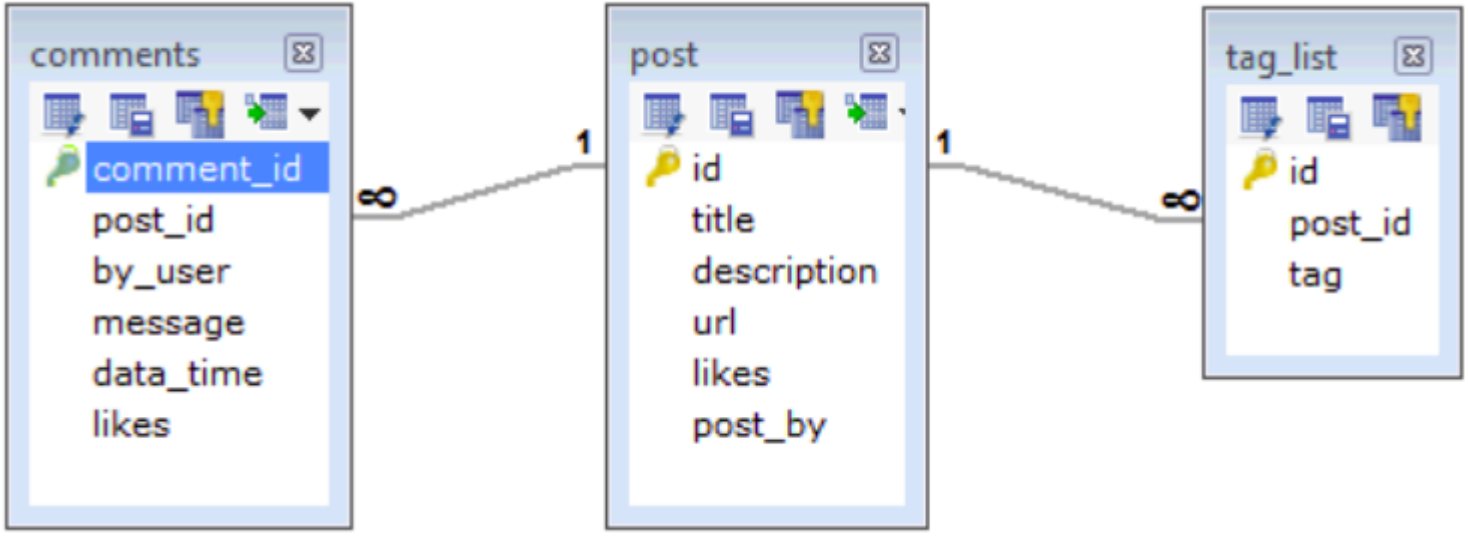
Example

- Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.

Example


comments
comment_id
post_id
by_user
message
data_time
likes
1
∞
post
id
title
description
url
likes
post_by
1
∞
tag_list
id
post_id
tag

Example

- **While in MongoDB schema, design will have one collection post and the following structure −**

```
{
   _id: POST_ID
   title: TITLE_OF_POST,
   description: POST_DESCRIPTION,
   by: POST_BY,
   url: URL_OF_POST,
   tags: [TAG1, TAG2, TAG3],
   likes: TOTAL_LIKES,
   comments: [
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      },
      {
         user:'COMMENT_BY',
         message: TEXT,
         dateCreated: DATE_TIME,
         like: LIKES
      }
   ]
}
```

mongoDB

MEAN stack

Getting Started

Install on Centos 7

- **Step 1 – Adding the MongoDB Repository**

  The mongodb-org package does not exist within the default repositories for CentOS. However, MongoDB maintains a dedicated repository. Let's add it to our server.

```
$ sudo vi /etc/yum.repos.d/mongodb-org.repo
```

```
[mongodb-org-3.4]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/
$releasever/mongodb-org/3.4/x86_64/
gpgcheck=1
enabled=1
gpgkey=https://www.mongodb.org/static/pgp/server-3.4.asc
```

Getting Started

Install on Centos 7

- **Step 2 – Installing MongoDB**

```
$ sudo yum install mongodb-org
```

```
Configuration file
/etc/mongod.conf
```

- **Step 3 – start the MongoDB service with the systemctl utility:**

```
$ sudo systemctl start mongod
```

```
$ sudo systemctl stop mongod
```

```
$ sudo systemctl reload mongod
```

```
mongodb-org
mongodb-org-server
mongodb-org-mongos
mongodb-org-shell
mongodb-org-tools
```

mongoDB

Getting Started

Install on Centos 7

- We can check that the service started by viewing the end of the mongod.log file with the tail command:

```
$ sudo tail /var/log/mongodb/mongod.log
```

```
Output
[initandlisten] waiting for connections on port 27017
```

```
$ sudo systemctl status mongod
```

Mongod (mongodb-org-server)

mongod is the primary daemon process for the MongoDB system. It handles data requests, manages data access, and performs background management operations.

By default, mongo looks for a database server listening on port 27017 on the localhost interface. To connect to a server on a different port or interface, use the --port and --host options.

```
$ mkdir /data/os_39
$ mongod --dbpath=/data/os_39  --port=27019
```

mongoDB

Mongo (mongodb-org-shell)

mongo is an interactive JavaScript shell interface to MongoDB, which provides a powerful interface for systems administrators as well as a way for developers to test queries and operations directly with the database. mongo also provides a fully functional JavaScript environment for use with a MongoDB. This document addresses the basic invocation of the mongo shell and an overview of its usage.

Open your terminal and write "mongo" with some options such as
mongo --port=27019 --username <username> --password <password>

$ mongo

mongoDB

MongoDB Basics

Binary JSON (BSON)

MongoDB represents JSON documents in binary-encoded format called BSON behind the scenes. BSON extends the JSON model to provide additional data types and to be efficient for encoding and decoding within different languages.

JSON

JavaScript Object Notation (JSON) is an open, human and machine-readable standard that facilitates data interchange, and along with XML is the main format for data interchange used on the modern web. JSON supports all the basic data types you'd expect: numbers, strings, and boolean values, as well as arrays and hashes.

help

```
>db.help()
```

This will give you a list of commands.

MongoDB CRUD Operations

- Create Operations.
- Read Operations.
- Update Operations.
- Delete Operations.
- Bulk Write.

mongoDB

MongoDB CRUD Operations

Create Operations:

Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.

mongoDB

Create Database

>db : to check your currently selected database.
will display “test” which is default database.

>show dbs : to check your database list.

>use DATABASE_NAME : used to create database if it doesn’t exist,otherwise it will return the existing database.

>use openSource

mongoDB

Create collection

After creating database, you need to insert at least one document into it, to display database.

Implicitly creation

```
>db.collection_name.insert ({“key”:”value”})

>db.students.insert ({“ first_name”:”Fatma”,
                       “last_name”:”Khaled”
                     })
```

mongoDB

Create collection

explicitly creation

```
>db.createCollection(“name”, options)
```

- Name is name of collection to be created.
- Options is a document and is used to specify configuration of collection.

Create collection

explicitly creation

```
>db.createCollection(“name”, options)
```

Parameter	Type	Description
Name	String	“Name of the collection to be created”
Options	Document	(Optional) Specify options about memory size and indexing

Create collection

Field	Type	Description
Capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size. If you specify true, you need to specify size parameter also.

Create collection

Field	Type	Description
autoIndexId	Boolean	(Optional) If true, automatically create index on _id field.s Default value is false.
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. If capped is true, then you need to specify this field also.
max	number	(Optional) Specifies the maximum number of documents allowed in the capped collection.

Create collection

```
>db.createCollection(name, options)

    db.createCollection(<name>, {
                        capped: <boolean>,
                       autoIndexId: <boolean>,
                       size: <number>,
                       max: <number>,
                        validator: <document>,
                        } )
```

Create collection

```
>db.createCollection("name", options)

    db.createCollection("opensource", {
                        capped: true,
                       autoIndexId: true,
                       size: 6142800,
                       max: 10000,
                         } )
```

Capped collection

- **Capped collections are fixed-size circular collections that follow the insertion order to support high performance for create, read, and delete operations.**

- **By circular, it means that when the fixed size allocated to the collection is exhausted, it will start deleting the oldest document in the collection without providing any explicit commands.**

Capped collection

- **Capped collections restrict updates to the documents if the update results in increased document size.**
- **Since capped collections store documents in the order of the disk storage, it ensures that the document size does not increase the size allocated on the disk.**
- **Capped collections are best for storing log information, cache data, or any other high volume data.**

Capped collection

```
>db.createCollection("users",{capped:true,
                              size:1000 ,
                              max:5})
```

```
db.users.insert({ _id: 1, "user":"Fatma", "email" : "f@gmail.com" })
db.users.insert({ _id: 2, "user":"Saeed", "email" : "s@gmail.com" })
db.users.insert({ _id: 3, "user":"Mohamed", "email" : "M@gmail.com" })
db.users.insert({ _id: 4, "user":"Khaled", "email" : "K@gmail.com" })
db.users.insert({ _id: 5, "user":"Ahmed", "email" : "A@gmail.com" })
```

Capped collection

```
> db.users.find()

{ "_id" : 1, "user" : "Fatma", "email" : " f@gmail.com" }
{ "_id" : 2, "user" : "Saeed", "email" : "s@gmail.com" }
{ "_id" : 3, "user" : "Mohamed", "email" : "M@gmail.com" }
{ "_id" : 4, "user" : "Khaled", "email" : "K@gmail.com" }
{ "_id" : 5, "user" : "Ahmed", "email" : "A@gmail.com" }
```

Capped collection

```
>db.users.insert({ _id: 6, "user":"Islam", "email" : "I@gmail.com" })
>db.users.insert({ _id: 7, "user":"Esraa", "email" : "E@gmail.com" })
```

Since we have a limit of 5 documents set while creating this capped collection, any further inserts remove the oldest entry automatically.

```
> db.users.find()
{ "_id" : 3, "user" : "Mohamed", "email" : "M@gmail.com" }
{ "_id" : 4, "user" : "Khaled", "email" : "K@gmail.com" }
{ "_id" : 5, "user" : "Ahmed", "email" : "A@gmail.com" }
{ "_id" : 6, "user" : "Islam", "email" : "I@gmail.com" }
{ "_id" : 7, "user" : "Esraa", "email" : "E@gmail.com" }
```

Capped collection

```
>db.collectionName.isCapped()
> db.users.isCapped()
  True
```

By default, a find query on a capped collection will display results in insertion order. But if you want the documents to be retrieved in reverse order.

```
>db.users.find().sort({_id:-1})
```

Capped collection

> if there is an existing collection which you are planning to convert to capped.

```
>db.runCommand({"convertToCapped":"posts",size:10000})
```

>This code would convert our existing collection posts to a capped collection.

Capped collection

- **We can’t shard capped collections.**
- **There is no delete option available on capped collections.**
- **Instead, you can drop() capped collection and recreate.**
- **while reading documents MongoDB returns the documents in the same order as present on disk. This makes the read operation very fast.**

Create Collection With Validation

```
db.createCollection( "contacts",
   {
      validator: { $or:
         [
            { phone: { $type: "string" } },
            { email: { $regex: /@mongodb\.com$/ } },
            { status: { $in: [ "Unknown", "Incomplete" ] } }
         ]
      }
   }
)
```

Insert collection

>db.collectionName.insert()

>db.collectionName.insertOne()

>db.collectionName.insertMany()

MongoDB CRUD Operations

Read Operations:

db.collection.find()
db.collection.find().pretty()

mongoDB

MongoDB CRUD Operations

Update Operations:

db.collection.update()
db.collection.updateOne()
db.collection.updateMany()
db.collection.replaceOne()

```
db.users.update(                    <-- collection
   { age: { $gt: 18 } },            <-- update criteria
   { $set: { status: "A" } },       <-- update action
   { multi: true }                  <-- update option
)
```

mongoDB

MongoDB CRUD Operations

Delete Operations:

db.collection.remove()
db.collection.deleteOne()
db.collection.deleteMany()
db.collection.replaceOne()

```
db.users.remove(          <-- collection
   { status: "D" }        <-- remove criteria
)
```

mongoDB

Drop collection

```
>db.Collection_name.drop()

>db.users.drop()
```

Drop Database

>db.dropDatabase() : used to drop existing database.

mongoDB

Bulk Write

Performs multiple write operations with controls for order of execution.

db.collection.bulkWrite()

```
db.collection.bulkWrite(
   [
      { insertOne : <document> },
      { updateOne : <document> },
      { updateMany : <document> },
      { replaceOne : <document> },
      { deleteOne : <document> },
      { deleteMany : <document> }
   ],
   { ordered : false }
)
```

Mongo query operators

- Query Comparison Operators.
- Query Logical Operators.
- Query Arrays Operators.
- Query Element Operators.

Query Comparison Operators

Name	Description
`$gt`	Matches values that are greater than the value specified in the query.
`$gte`	Matches values that are greater than or equal to the value specified in the query.
`$in`	Matches any of the values that exist in an array specified in the query.
`$lt`	Matches values that are less than the value specified in the query.
`$lte`	Matches values that are less than or equal to the value specified in the query.
`$ne`	Matches all values that are not equal to the value specified in the query.
`$nin`	Matches values that **do not** exist in an array specified to the query.

Query Comparison Operators

- db.students.find({“age”:{$gt:20}})
- db.students.find({“age”:{$lt:20}})
- db.students.find({“age”:{$ne:20}})
- db.students.find({“age”:{$in:[10,20,30]}})
- db.students.find({“age”:{$nin:[10,20,30]}})

mongoDB

Query Comparison Operators

- db.inventory.find({type:{$in:[‘food’,’snacks’]}})
- db.inventory.find({qty:{$gt:20}})

Query Comparison Operators

- The $in operator can specify matching values using regular expressions of the form /pattern/.
- You cannot use $regex operator expressions inside an $in.
- Example:
  selects all documents in the inventory collection where the tags field holds an array that contains at least one element that starts with either be or st.

Query Comparison Operators

- **The $in operator can specify matching values using regular expressions of the form /pattern/.**

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

- **Example:**
  selects all documents in the inventory collection where the tags field holds an array that contains at least one element that starts with either be or st.

  >db.inventory.find({ tags: { $in: [/^be/, /^st/] } })

Query Logical Operators

Name	Description
`$and`	Joins query clauses with a logical **AND** returns all documents that match the conditions of both clauses.
`$nor`	Joins query clauses with a logical **NOR** returns all documents that fail to match both clauses.
`$not`	Inverts the effect of a query expression and returns documents that do *not* match the query expression.
`$or`	Joins query clauses with a logical **OR** returns all documents that match the conditions of either clause.

Query Logical Operators

- db.students.find({$and:[{name:"Ali"},
  {age:{$gt:15}}]})

- db.students.find({$or:[{name:"Ali"},
  {age:{$gt:28}}]})

Query Logical Operators

- **Select all documents from inventory collections where price is not greater than 1.99**

  > db.inventory.find({price: { $not:{ $gt: 1.99 } } })

  the price field value is less than or equal to 1.99 or the price field does not exist.

Query Comparison Operators

- **select all documents in the inventory collection where:**

  the price field value is not equal to 1.99 and
  the price field exists.

```
> db.inventory.find({ $and: [ { price: { $ne: 1.99 } },
                              { price: { $exists: true } } ] })
```

mongoDB

Query Comparison Operators

- **Select all documents from the collection "student" which satisfying the condition (age of the student is at least 12)**

Query Comparison Operators

- **Select all documents from the collection "student" which satisfying the condition (age of the student is at least 12)**

```
>db.student.find( {"age": { $not: {$lt : 12}}})
```

Query Arrays Operators

Name	Description
`$all`	Matches arrays that contain all elements specified in the query.
`$elemMatch`	Selects documents if element in the array field matches all the specified `$elemMatch` conditions.
`$size`	Selects documents if the array field is a specified size.

Query Arrays Operators

- { "_id" : 10, "name" : "ali", "love" : ["HTML", "php", "js"] }
- { "_id" : 11, "name" : "alaa", "love" : ["HTML", "php", "python"] }

- db.users.find({"love":{$all:["HTML"]}},{})
  { "_id" : 10, "name" : "ali", "love" : ["HTML", "php", "js"] }
  { "_id" : 11, "name" : "alaa", "love" : ["HTML", "php", "python"] }
- db.users.find({"love":{$all:["HTML",”js"]}},{})
  { "_id" : 10, "name" : "ali", "love" : ["HTML", "php", "js"] }

Query Arrays Operators

- { "_id" : 5, "name" : "ahmed", "score" : [82, 85, 88] }
- { "_id" : 6, "name" : "mohamed", "score" : [75, 88, 89]}

- db.users.find({score:{$elemMatch:{$gt:20}}})
  { "_id" : 5, "name" : "ahmed", "score" : [82, 85, 88] }
  { "_id" : 6, "name" : "mohamed", "score" : [75, 88, 89]}
- db.users.find({score:{$elemMatch:{$lte:75}}})
  { "_id" : 6, "name" : "ahmed", "score" : [75, 88, 89] }
- db.users.find({score:{$elemMatch:{$gt:80, $lt:85}}})
  { "_id" : 5, "name" : "ahmed", "score" : [82, 85, 88] }

Query Arrays Operators

- db.inventory.find({ tags: { $all: ["ssl" , "security"] } })
- The tags field value is an array and should contain ssl, security elements
- db.students.find({ score: { $elemMatch: { $gte: 80, $lte: 85 }}})
- The score is an array and should contains at least one element matches this query (i.e. 82)

Query Arrays Operators

- db.inventory.find({ tags: { $size:2 } })

- Return all documents in inventory collection where field tags is an array with 2 elements.

Query Elements Operators

Name	Description
`$exists`	Matches documents that have the specified field.
`$type`	Selects documents if a field is of the specified type.

Query Elements Operators

- db.inventory.find({qty: {$exists: true,
  $nin: [5, 15] } })
- This query will select all documents in the inventory collection where the qty field exists and its value does not equal 5 or 15.

- db.inventory.find({tags: {$type: 2 } });

- This will list all documents containing a tags field that is either a string or an array holding at least one string.

mongoDB

Lab 1

1)Install Mongo Database.
2)open mongo shell and view the help.
3)identify your current working database and show list of available databases.
4) create a new database called “Facebook” and use it.
5)Create Collection with name “posts” which has facebook post properties
[“post_text”,”images”,”likes”,”comments”,
”Datetime”,”owner”,”live”]

mongoDB

Lab 1

6)Create Capped Collection with name users with Size 5 MB , 10 users Maximum and must has username field “String” and email end with “@gmail.com”.

7)Insert 20 post “ordered Insert”.

8)Insert 10 users.

9)Display all users.

10)Display user “Mohamed” posts

11)Update Mohamed ‘s posts set likes 10000

12)delete Mohamed ‘s posts

Lab 1

13)what is sharding ?
14)What is replication?
15)what is failover mechanism ?
16)what is embedded documents ?
17)What ACID?
18) User Management Methods?

mongoDB

Lab 1

19)Import Inventory Database using this command in terminal
mongorestore --db Inventory path_to_Inventory_folder.

20)Select products with price less than 1000 or greater than 5000.

21)Select products where name field contains at least one element that starts with LG , To ,Sa.

22)Select products where the stock field value is an array and should contain numbers 99 , 999.

23)Select products with stock field contains value is greater than 99.

24)Select products where stock field contains 3 elements.

26)Select products where vendor is not Apple , Sony, LG or HP.

27)Select products where the price field is not exists.