

**Instructor Notes:**

Add instructor notes here.

# Node JS

Lesson 03 : Database

Capgemini

**Instructor Notes:**

Add instructor notes here.

**Lesson Objectives**

Mongoose Library  
Introduction to Mongoose Library  
Connecting MongoDB using Mongoose Library  
Object Modeling  
RDBMS Schema Vs Mongoose Schema  
Simple Schema  
Complex Schema  
Validations in Model  
Finding documents  
Query API



**Instructor Notes:**

Add instructor notes here.

**3.1: MongoDB Basics  
Introduction**

- MongoDB is one of the most popular and fastest growing open source NoSQL database.
- MongoDB is written in C++. It is fast and scalable.
- Most of the MongoDB functionalities can be accessed directly through JavaScript notation and we can make use of all of the standard JavaScript libraries within it.
- It uses JSON for storing and manipulating the data. A JSON database returns query results that can be easily parsed, with little or no transformation, directly by JavaScript and most popular programming languages which reduces the amount of logic need to build into the application layer.
- MongoDB represents JSON documents in binary-encoded format called BSON(Binary JSON) behind the scenes. BSON extends the JSON model to provide additional data types and to be efficient for encoding and decoding within different languages.

July 31, 2018

Proprietary and Confidential

- 3 -

The MongoDB BSON implementation is lightweight, fast and highly traversable.

BSON is the binary encoding of JSON-like documents that MongoDB uses when storing documents in collections.

It adds support for data types like Date and binary that aren't supported in JSON.

In practice, you don't have to know much about BSON when working with MongoDB, you just need to use its data types when constructing documents.

## Instructor Notes:

Add instructor notes here.

### 3.1: MongoDB Basics Why MongoDB?



- MongoDB is a cross-platform, document oriented database that provides, high performance, high availability and easy scalability
- It supports a wide range of Operating Systems (Windows, OSX, Linux)
- There are drivers for nearly any language including C/C++, Python, PHP, Ruby, Perl, .NET and Node.js.
- Document Oriented Storage i.e. Data is stored in the form of JSON style documents
- Index on any attribute
- Replication & High Availability
- Auto-Sharding
- Rich Queries
- Migrations and a constantly evolving schemas can be managed easier

July 31, 2018

Proprietary and Confidential

- 4 -

#### **Indexing :**

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require the mongod to process a large volume of data. Indexes are special data structures, that store a small portion of the data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in index.

#### **Replication :**

Replication is the process of synchronizing data across multiple servers. Replication provides redundancy and increases data availability with multiple copies of data on different database servers, replication protects a database from the loss of a single server. Replication also allows us to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup

#### **Auto-Sharding :**

Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, we can add more machines to support data growth and the demands of read and write operations.

#### **Rich Queries :**

- Arbitrary sorting on one or more fields, ascending or descending.
- Projection, i.e. retrieving only specified fields from each document.
- skip() and limit() to easily implement pagination if desired.
- explain(), which returns a wealth of information including full details on query path analysis, document and index counts, and estimated vs. actual processing time.

**Instructor Notes:**

Add instructor notes here.

**3.1: MongoDB Basics**  
**MongoDB key terminologies****➤ Database**

- Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases

**➤ Collection**

- A *collection* may be considered as a table except there are no aligned columns.

**➤ Document**

- Each of the entries or rows inside a collection is called a *document*. Each entry (row) can use varying dynamic schemas in key-value pairs.
- Collections have dynamic schemas. This means that the documents within a single collection can have any number of different "shapes."
- Inside a collection of Users there may be one entry with First name & Last name. Then another entry with First, Last, and Middle name, along with e-mail address and date of birth.
- Documents are basically JSON data blocks stored in memory-mapped files which behave as separate entries in collections.

**Instructor Notes:**

Add instructor notes here.

3.1: MongoDB Basics

**SQL Terminology vs MongoDB Terminology**

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	embedded documents and linking
primary key	primary key In MongoDB primary key is automatically set to the <code>_id</code> field.
aggregation (e.g. group by)	aggregation pipeline.

**Instructor Notes:**

Add instructor notes here.

## 3.1: MongoDB Basics

## MongoDB vs Relational Databases



- Relational databases save data in tables and rows.
- But in application development our objects are not simply tables and rows. It forces an application developer to write a mapping layer or use an ORM, to translate the object in memory and what is saved in the database. Mapping those to tables and rows can be quite a bit of pain.
- In MongoDB, there is no schema to define. There are no tables and no relationships between collections of objects.
- Every document you save in Mongo can be as flat and simple, or as complex as your application requires. This makes developer life much easier and your application code much cleaner and simpler.
- Further, two documents in the same collection may be different from each other since there is no schema governing the collection.
- Structuring a single object is clear and no complex joins

**Instructor Notes:**

Add instructor notes here.

**3.2: Getting started with MongoDB  
Starting the mongo shell**

➤ To start the mongo shell and connect to your MongoDB instance running on localhost(127.0.0.1) with default port(27017)

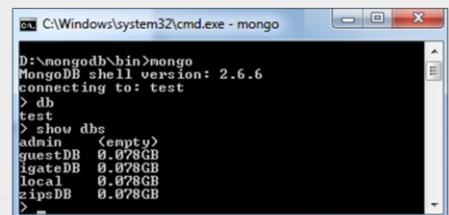
- **Go to <mongodb installation dir> : cd <mongodb installation dir>**
- **To start mongo goto bin directory and type mongo ex:-**  
*D:\mongodb\bin>mongo*

➤ To check your currently selected database, use the command db.  
Default(test)

- *db*

➤ To list the databases, use the command show dbs.

- *show dbs*



```
C:\Windows\system32\cmd.exe - mongo
D:\mongodb\bin>mongo
MongoDB shell version: 2.6.6
connecting to: test
> db
test
> show dbs
admin   <empty>
guestDB  0.078GB
logateDB  0.078GB
local    0.078GB
zipsDB   0.078GB
> _
```

July 31, 2018

Proprietary and Confidential

- 8 -

**Instructor Notes:**

Add instructor notes here.

**3.2: Getting started with MongoDB  
Creating and Dropping Database**

- MongoDB *use DATABASE\_NAME* is used to create database.
- If the database doesn't exists it creates a new database, otherwise it will return the existing database which can be used using *db*.
  - *use DATABASE\_NAME*
- To display database you need to insert at least one document into it.
- *db.dropDatabase()* command is used to drop a existing database.

```
C:\Windows\system32\cmd.exe - mongo
MongoDB shell version: 2.6.6
connecting to: test
> show dbs
admin      <empty>
guestDB   0.078GB
igateDB   0.078GB
local     0.078GB
zipsDB    0.078GB
> use zipsDB
switched to db zipsDB
> db
zipsDB
> db.dropDatabase()
{
  "dropped" : "zipsDB",
  "ok" : 1
}
>
```

July 31, 2018

Proprietary and Confidential

- 9 -

**Instructor Notes:**

Add instructor notes here.

### 3.2: Getting started with MongoDB Creating and Dropping Collection

➤ MongoDB `db.createCollection(collectionname, options)` is used to create collection. Options parameter is optional. Following is the list of options

- **capped** : Capped collection is a collection fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size.
- **autoIndexID**: If true, If true, automatically create index on `_id` field.
- **size**: Specifies a maximum size in bytes for a capped collection.
- **max**: Specifies the maximum number of documents allowed in the capped collection.

➤ `db.<collectionname>.drop()` command is used to drop a collection

```

C:\Windows\system32\cmd.exe - mongo
> use sampleDB
switched to db sampleDB
> db
sampleDB
> show collections
> db.createCollection("mycol", { capped : true, autoIndexID : true, size : 61428
 0, "max": 10000 })
{
  "_id": 1
}
> show collections
mycol
system.indexes
> db.mycol.drop()
true
> show collections
system.indexes
>

```

July 31, 2018 | Proprietary and Confidential | ~ 10 ~

MongoDB supports many datatypes whose list is given below:

**String** : This is most commonly used datatype to store the data. String in mongodb must be UTF-8 valid.

**Integer** : This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.

**Boolean** : This type is used to store a boolean (true/ false) value.

**Double** : This type is used to store floating point values.

**Min/ Max keys** : This type is used to compare a value against the lowest and highest BSON elements.

**Arrays** : This type is used to store arrays or list or multiple values into one key.

**Timestamp** : ctimestamp. This can be handy for recording when a document has been modified or added.

**Object** : This datatype is used for embedded documents.

**Null** : This type is used to store a Null value.

**Symbol** : This datatype is used identically to a string however, it's generally reserved for languages that use a specific symbol type.

**Instructor Notes:**

Add instructor notes here.

**3.2: Getting started with MongoDB  
Importing and Exporting Collection**

- To import the collection from the file use *mongoimport* command
  - *mongoimport --db <DBName> --collection <CollectionName> --file <FileName>*
- To export the collection from the DB use *mongoexport* command
  - *mongoexport --db <DBName> --collection <CollectionName> --out <OutputFileName>*

```
D:\mongod\bin>mongo SampleDB
MongoDB shell version: 2.6.6
connecting to: SampleDB
> db
SampleDB
> show collections
> quit<
D:\mongod\bin>mongoimport --db SampleDB --collection employees --file d:\Karthik\NodeJS\Lesson03\employees.json
connected to: 127.0.0.1
2015-02-01T14:08:05.135+05:30 check 9 75
2015-02-01T14:08:05.135+05:30 imported 75 objects
D:\mongod\bin>mongoexport --db SampleDB --collection employees --out d:\Karthik\NodeJS\Lesson03\employees-backup.json
connected to: 127.0.0.1
exported 75 records
D:\mongod\bin>mongo SampleDB
MongoDB shell version: 2.6.6
connecting to: SampleDB
> show collections
employees
system.indexes
>
```

July 31, 2018 | Proprietary and Confidential | - 11 -

**Date :** This datatype 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 :** This datatype is used to store the document's ID.

**Binary data :** This datatype is used to store binay data.

**Code :** This datatype is used to store JavaScript code into document.

**Regular expression :** This datatype is used to store regular expression

**Instructor Notes:**

Add instructor notes here.

## 3.3: MongoDB Queries

## Querying MongoDB Documents – find()



➤ MongoDB's `find()` method is used to query data from MongoDB collection

- **`db.COLLECTION_NAME.find()`**
- **`db.COLLECTION_NAME.find().pretty()`** is used to display the results in a formatted way.
- **`db.COLLECTION_NAME.count()`** is used to count the number of documents in a collection.

```
D:\mongodb\bin>mongoimport --db SampleDB --collection locations --file d:\Karthik\NodeJS\Lesson03\locations.json
connected to: 127.0.0.1
2015-02-01T14:43:42.868+0530 imported 7 objects
D:\mongodb\bin>mongo SampleDB
MongoDB shell version: 2.6.6
connecting to: SampleDB
> db
SampleDB
> show collections
employees
locations
system.indexes
> db.locations.find()
< "id" : 1, "location" : "Bangalore" >
< "id" : 2, "location" : "Chennai" >
< "id" : 3, "location" : "Gandhinagar" >
< "id" : 4, "location" : "Hyderabad" >
< "id" : 5, "location" : "Mumbai" >
< "id" : 6, "location" : "Noida" >
< "id" : 7, "location" : "Pune" >
> db.locations.count()
7
>
```

July 31, 2016 | Proprietary and Confidential | 12 |

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Querying MongoDB Documents – pretty()**

➤ To print the results in formatted way use pretty()

```
D:\mongod\bin>mongo SampleDB
MongoDB shell version: 2.6.6
connecting to: SampleDB
> db
SampleDB
> show collections
employees
locations
system.indexes
> db.employees.find({ _id: "714709" }).pretty()
{
  "_id": "714709",
  "name": {
    "first": "Karthik",
    "last": "Muthukrishnan"
  },
  "date": "2010-04-12T00:00:00.000Z",
  "location": "Bangalore",
  "isactive": true,
  "email": "Karthik.Muthukrishnan@igate.com",
  "qualifications": [
    "B.Sc(GS)",
    "M.C.A"
  ]
}
> db.employees.find({ _id: "714709" }, { name: 1, email: 1 }).pretty()
{
  "_id": "714709",
  "name": {
    "first": "Karthik",
    "last": "Muthukrishnan"
  },
  "email": "Karthik.Muthukrishnan@igate.com"
}
> db.employees.find({ _id: "714709" }, { name: 1, email: 1, _id: 0 }).pretty()
{
  "name": {
    "first": "Karthik",
    "last": "Muthukrishnan"
  },
  "email": "Karthik.Muthukrishnan@igate.com"
}
```

**Instructor Notes:**

Add instructor notes here.

## 3.3: MongoDB Queries

## Querying MongoDB Documents – findOne()



➤ `findOne()` returns a single document, whereas `find()` returns a cursor.

```
> db
SampleDB
> show collections
employees
locations
system.indexes
> db.employees.find().limit(1).doj
2010-04-12T00:00:00.000Z
> new Date(db.employees.find().limit(1).doj).toLocaleDateString()
Monday, April 12, 2010
> db.employees.findOne()
{
    "_id" : 2072,
    "name" : {
        "first" : "Rohini",
        "last" : "Vijayan"
    },
    "doj" : "1995-07-31T00:00:00.000Z",
    "location" : "Pune",
    "isActive" : true,
    "email" : "rohini.vijayan@igate.com",
    "qualifications" : [
        "B.Com",
        "D.C.A"
    ]
}
> db.employees.findOne().doj
1995-07-31T00:00:00.000Z
> new Date(db.employees.findOne().doj).toLocaleDateString()
Monday, July 31, 1995
>
```

July 31, 2018

**Instructor Notes:**

Add instructor notes here.

3.3: MongoDB Queries  
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.
\$le	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.

**Instructor Notes:**

Add instructor notes here.

3.4: MongoDB Queries  
Comparison Operators – gte & lte

```
> db
SampleDB
> show collections
employees
locations
system.indexes
> db.employees.find({ _id: { $gte: 700000, $lte: 715000 } }, { name: 1 })
[ { "_id": "705062", "name": { "first": "Rashmi", "last": "Keshavanurthy" } },
  { "_id": "707224", "name": { "first": "Latha", "last": "Subrananian" } },
  { "_id": "714709", "name": { "first": "Karthik", "last": "Muthukrishnan" } } ]
```

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
MongoDB Comparison Operators – in & nin**

```
> db
> SampleDB
> show collections
employees
locations
system.indexes
>
> db.employees.find({ 'name.first' : { $in: ['Vaishali','Veena'] } }, { name : 1, _id : 0 })
< [
  { "name" : { "first" : "Veena", "last" : "Deshpande" } },
  { "name" : { "first" : "Vaishali", "last" : "Kulkarni" } },
  { "name" : { "first" : "Vaishali", "last" : "Kunchur" } },
  { "name" : { "first" : "Vaishali", "last" : "Kasture" } },
  { "name" : { "first" : "Veena", "last" : "Meshavalu" } },
  { "name" : { "first" : "Vaishali", "last" : "Srivastava" } }
]
> db.employees.find({ 'location' : { $in: ['Bangalore','Mumbai','Pune'] } }, { name : 1, location : 1, _id : 0 })
< [
  { "name" : { "first" : "Karthikkeyan", "last" : "Ramanathan" }, "location" : "Chennai" },
  { "name" : { "first" : "Rathnajothi", "last" : "Perumalsamy" }, "location" : "Chennai" },
  { "name" : { "first" : "Hema", "last" : "Gandhi" }, "location" : "Chennai" },
  { "name" : { "first" : "Selvarajakshi", "last" : "Palanichelvan" }, "location" : "Chennai" },
  { "name" : { "first" : "Balachander", "last" : "Meghraj" }, "location" : "Chennai" },
  { "name" : { "first" : "Obihewk", "last" : "Radhakrishnan" }, "location" : "Chennai" },
  { "name" : { "first" : "Kalaivani", "last" : "Rajabhadhar" }, "location" : "Chennai" }
]
> db.employees.find({ 'qualifications' : { $in: ['M.E(CSE)'] } }, { name : 1, qualifications : 1, _id : 0 }).pretty()
<
  {
    "name" : {
      "first" : "Anil",
      "last" : "Patil"
    },
    "qualifications" : [
      "B.Sc(GS)",
      "M.E(CSE)"
    ]
  },
  {
    "name" : {
      "first" : "Roshi",
      "last" : "Saxena"
    },
    "qualifications" : [
      "B.Tech(CSE)",
      "M.E(CSE)"
    ]
  }
>
```

July 31, 2018

Proprietary and Confidential

- 17 -

**Instructor Notes:**

Add instructor notes here.

3.3: MongoDB Queries  
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.

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Logical Operators – or, not & and**

```
> db
> db
SampleDB
>
> db.employees.find({ $or : [ { 'name.first' : 'Vaishali' }, { 'name.last' : 'Kulkarni' } ] }, { name : 1, _id : 0 })
{ "name" : { "first" : "Vaishali", "last" : "Kulkarni" } }
{ "name" : { "first" : "Vaishali", "last" : "Kunchum" } }
{ "name" : { "first" : "Vaishali", "last" : "Kasture" } }
{ "name" : { "first" : "Shanika", "last" : "Kulkarni" } }
{ "name" : { "first" : "Vaishali", "last" : "Srivastava" } }
{ "name" : { "first" : "Zainab", "last" : "Kulkarni" } }
>
> db.employees.find({ $nor : [ { 'isActive' : true }, { 'location' : 'Bangalore' } ] }, { name : 1, isActive : 1, _id : 0 })
{ "name" : { "first" : "Anagha", "last" : "Narvekar" }, "isActive" : false }
{ "name" : { "first" : "Shrilata", "last" : "Iavargeri" }, "isActive" : false }
{ "name" : { "first" : "Peavin", "last" : "Surve" }, "isActive" : false }
{ "name" : { "first" : "Rajit", "last" : "Jog" }, "isActive" : false }
{ "name" : { "first" : "Sanant", "last" : "Gour" }, "isActive" : false }
{ "name" : { "first" : "Hareshkumar", "last" : "Chandiranani" }, "isActive" : false }
{ "name" : { "first" : "Mandar", "last" : "Randas" }, "isActive" : false }
{ "name" : { "first" : "Pushpendra", "last" : "Misha" }, "isActive" : false }
{ "name" : { "first" : "Bhavna", "last" : "Beri" }, "isActive" : false }
>
> db.employees.find({ $and : [ { location : 'Bangalore' }, { 'name.last' : 'Kulkarni' } ] }, { name : 1, location : 1, _id : 0 })
{ "name" : { "first" : "Zainab", "last" : "Kulkarni" }, "location" : "Bangalore" }
```

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries**  
**MongoDB Additional operators**

Name	Description
\$all	Matches arrays that contain all elements specified in the query.
\$exists	Matches documents that have the specified field.
\$regex	Selects documents where values match a specified regular expression.
\$where	Matches documents that satisfy a JavaScript expression.
\$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
\$limit	Passes the first n documents unmodified to the pipeline where n is the specified limit. For each input document, outputs either one document (for the first n documents) or zero documents (after the first n documents).
\$skip	Skips the first n documents where n is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first n documents) or one document (if after the first n documents).

**Instructor Notes:**

Add instructor notes here.

3.3: MongoDB Queries  
MongoDB Additional operators - all

```
> db
SampleDB
> show collections
employees
locations
system.indexes
>
> db.employees.find({ 'qualifications' : { $all: ['B.Sc(CS)', 'M.E(CSE)'] } },
  { name : 1, qualifications : 1, _id : 0 }).pretty()
{
  "name" : {
    "first" : "Anil",
    "last" : "Patil"
  },
  "qualifications" : [
    "B.Sc(CS)",
    "M.E(CSE)"
  ]
}
>
```

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries**  
**MongoDB Additional operators – exists & not**

```
> db
> db.sampleDB
>
> db.employees.find({ qualifications : { $exists : false } }, { name : 1, _id : 0 })
< "name" : { "first" : "Pranod", "last" : "Patwardhan" }
< "name" : { "first" : "Anagha", "last" : "Narvekar" }
< "name" : { "first" : "Shrilata", "last" : "Tavargeri" }
< "name" : { "first" : "Vinay", "last" : "Gupta" }
< "name" : { "first" : "Ajit", "last" : "Jog" }
< "name" : { "first" : "Samant", "last" : "Gour" }
< "name" : { "first" : "Harehkumar", "last" : "Chandiramani" }
< "name" : { "first" : "Satyen", "last" : "Nande" }
< "name" : { "first" : "Mandar", "last" : "Randas" }
< "name" : { "first" : "Suresh", "last" : "Kumar" }
< "name" : { "first" : "Naveen", "last" : "Bandi" }
< "name" : { "first" : "Sudhip", "last" : "Rao" }
< "name" : { "first" : "Pushpendra", "last" : "Misra" }
< "name" : { "first" : "Bhavna", "last" : "Beri" }
< "name" : { "first" : "Bhushan", "last" : "Bhupta" }
< "name" : { "first" : "Shefali", "last" : "Pathak" }
< "name" : { "first" : "Anjana", "last" : "Pathare" }
>
> db.employees.find( { qualifications : { $not : { $exists : true } } }, { name : 1, _id : 0 })
< "name" : { "first" : "Pranod", "last" : "Patwardhan" }
< "name" : { "first" : "Anagha", "last" : "Narvekar" }
< "name" : { "first" : "Shrilata", "last" : "Tavargeri" }
< "name" : { "first" : "Vinay", "last" : "Gupta" }
< "name" : { "first" : "Ajit", "last" : "Jog" }
< "name" : { "first" : "Samant", "last" : "Gour" }
< "name" : { "first" : "Harehkumar", "last" : "Chandiramani" }
< "name" : { "first" : "Satyen", "last" : "Nande" }
< "name" : { "first" : "Mandar", "last" : "Randas" }
< "name" : { "first" : "Suresh", "last" : "Kumar" }
< "name" : { "first" : "Naveen", "last" : "Bandi" }
< "name" : { "first" : "Sudhip", "last" : "Rao" }
< "name" : { "first" : "Pushpendra", "last" : "Misra" }
< "name" : { "first" : "Bhavna", "last" : "Beri" }
< "name" : { "first" : "Bhushan", "last" : "Bhupta" }
< "name" : { "first" : "Shefali", "last" : "Pathak" }
< "name" : { "first" : "Anjana", "last" : "Pathare" }
```

**Instructor Notes:**

Add instructor notes here.

3.3: MongoDB Queries  
MongoDB Additional operators - regex

```
> db
SampleDB
> db.employees.find( { 'name.first' : /Ka+/ }, { name : 1, _id : 0 } )
[ { "name" : { "first" : "Karthickeyan", "last" : "Ramanathan" } },
  { "name" : { "first" : "Karthik", "last" : "Muthukrishnan" } },
  { "name" : { "first" : "Kalaivani", "last" : "Rajabadhar" } },
  { "name" : { "first" : "Kavita", "last" : "Arora" } }
]
> db.employees.find( { 'name.first' : { $regex : /[K]+/ } }, { name : 1, _id : 0 } )
[ { "name" : { "first" : "Karthickeyan", "last" : "Ramanathan" } },
  { "name" : { "first" : "Karthik", "last" : "Muthukrishnan" } },
  { "name" : { "first" : "Kalaivani", "last" : "Rajabadhar" } },
  { "name" : { "first" : "Kavita", "last" : "Arora" } }
]
> db.employees.find( { 'name.first' : { $regex : /[K]+/ } }, { name : 1, _id : 0 } ).count()
4
```

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries**  
**MongoDB Additional operators - where**

```
> db
> db
SampleDB
>
> db.employees.find( { $where : 'this.name.last === "Kulkarni" ' }, { name : 1, _id : 0 } )
< "name" : { "first" : "Vaishali", "last" : "Kulkarni" }
< "name" : { "first" : "Shamika", "last" : "Kulkarni" }
< "name" : { "first" : "Zainab", "last" : "Kulkarni" }
>
> db.employees.find( { $where : 'this.location === "Chennai" ' }, { name : 1, _id : 0 } )
< "name" : { "first" : "Karthikeyan", "last" : "Bananathan" }
< "name" : { "first" : "Rathnajothi", "last" : "Perumalsamy" }
< "name" : { "first" : "Hema", "last" : "Gandhi" }
< "name" : { "first" : "Selvalakshmi", "last" : "Palanicelvam" }
< "name" : { "first" : "Balachander", "last" : "Meghraj" }
< "name" : { "first" : "Abishek", "last" : "Radhakrishnan" }
< "name" : { "first" : "Kalaiani", "last" : "Rajabadhar" }
>
> var getKulkarni = function(){ return this.name.last === 'Kulkarni' }
>
> db.employees.find( { $where : getKulkarni }, { name : 1, _id : 0 } )
< "name" : { "first" : "Vaishali", "last" : "Kulkarni" }
< "name" : { "first" : "Shamika", "last" : "Kulkarni" }
< "name" : { "first" : "Zainab", "last" : "Kulkarni" }
>
> db.employees.find( getKulkarni, { name : 1, _id : 0 } )
< "name" : { "first" : "Vaishali", "last" : "Kulkarni" }
< "name" : { "first" : "Shamika", "last" : "Kulkarni" }
< "name" : { "first" : "Zainab", "last" : "Kulkarni" }
```

**Instructor Notes:**

Add instructor notes here.

3.3: MongoDB Queries  
MongoDB Additional operators - sort

```
>
> db
SampleDB
> db.employees.find( { 'name.first' : { $regex : /[K]+/ } },
  { name : 1, _id : 0 }).sort( { 'name.first' : 1 })
[{"name": {"first": "Kalaivani", "last": "Rajabadhar"}, "last": "Rajabadhar"}, {"name": {"first": "Karthik", "last": "Muthukrishnan"}, "last": "Muthukrishnan"}, {"name": {"first": "Karthikeyan", "last": "Ramanathan"}, "last": "Ramanathan"}, {"name": {"first": "Kavita", "last": "Arora"}, "last": "Arora"}]
>
> db.employees.find( { 'name.first' : { $regex : /[K]+/ } },
  { name : 1, _id : 0 }).sort( { 'name.first' : -1 })
[{"name": {"first": "Kavita", "last": "Arora"}, "last": "Arora"}, {"name": {"first": "Karthikeyan", "last": "Ramanathan"}, "last": "Ramanathan"}, {"name": {"first": "Karthik", "last": "Muthukrishnan"}, "last": "Muthukrishnan"}, {"name": {"first": "Kalaivani", "last": "Rajabadhar"}, "last": "Rajabadhar"}]
>
> db.employees.find( { 'name.first' : { $regex : /[K]+/ } },
  { name : 1, _id : 0 }).sort( { location: 1, 'name.first': -1 })
[{"name": {"first": "Karthik", "last": "Muthukrishnan"}, "last": "Muthukrishnan"}, {"name": {"first": "Karthikeyan", "last": "Ramanathan"}, "last": "Ramanathan"}, {"name": {"first": "Kalaivani", "last": "Rajabadhar"}, "last": "Rajabadhar"}, {"name": {"first": "Kavita", "last": "Arora"}, "last": "Arora"}]
```

**Instructor Notes:**

Add instructor notes here.

3.3: MongoDB Queries  
MongoDB Additional operators – limit & skip

```
> db
SampleDB
> db.locations.find().sort({ location : -1 })
< "_id" : 7, "location" : "Pune" >
< "_id" : 6, "location" : "Noida" >
< "_id" : 5, "location" : "Mumbai" >
< "_id" : 4, "location" : "Hyderabad" >
< "_id" : 3, "location" : "Gandhinagar" >
< "_id" : 2, "location" : "Chennai" >
< "_id" : 1, "location" : "Bangalore" >
>
> db.locations.find().sort({ location : -1 }).limit(2)
< "_id" : 7, "location" : "Pune" >
< "_id" : 6, "location" : "Noida" >
>
> db.locations.find().sort({ location : 1 }).skip(0 * 2).limit(2)
< "_id" : 1, "location" : "Bangalore" >
< "_id" : 2, "location" : "Chennai" >
>
> db.locations.find().sort({ location : 1 }).skip(1 * 2).limit(2)
< "_id" : 3, "location" : "Gandhinagar" >
< "_id" : 4, "location" : "Hyderabad" >
>
> db.locations.find().sort({ location : 1 }).skip(2 * 2).limit(2)
< "_id" : 5, "location" : "Mumbai" >
< "_id" : 6, "location" : "Noida" >
>
> db.locations.find().sort({ location : 1 }).skip(3 * 2).limit(2)
< "_id" : 7, "location" : "Pune" >
>
```

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Inserting Document(s)**

- In MongoDB, to Insert a document or documents into a collection we need to use db.collection.insert()

```
> db
> SampleDB
> db.persons.count()
0
> db.persons.insert({name:'Karthik'})
WriteResult({ "nInserted": 1 })
> db.persons.find()
< "_id" : ObjectId("54ce287fdce2920aa624a655"), "name" : "Karthik" >
> db.persons.insert({_id: 1, name:'Abishek'})
WriteResult({ "nInserted": 1 })
> db.persons.find()
< "_id" : ObjectId("54ce287fdce2920aa624a655"), "name" : "Karthik" >
< "_id" : 1, "name" : "Abishek" >
> db.persons.insert({_id: new ObjectId(), name:'Latha'})
WriteResult({ "nInserted": 1 })
> db.persons.find()
< "_id" : ObjectId("54ce287fdce2920aa624a655"), "name" : "Karthik" >
< "_id" : 1, "name" : "Abishek" >
< "_id" : ObjectId("54ce2968dce2920aa624a656"), "name" : "Latha" >
> db.persons.find().sort({_id: 1}).getTimestamp()
ISODate("2015-02-01T13:22:07Z")
> db.persons.find().sort({_id: 1}).getTimestamp().toLocaleDateString()
Sunday, February 01, 2015
>
>
```

July 31, 2018 | Proprietary and Confidential | - 27 -

**To Insert Multiple Documents**

```
db.products.insert([
  {
    "_id": 11, item: "pencil", qty: 50, type: "no.2" },
  { item: "pen", qty: 20 },
  { item: "eraser", qty: 25 }
])
```

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
ObjectId**

- In MongoDB, documents stored in a collection require a unique `_id` field that acts as a primary key.
- MongoDB uses ObjectIds as the default value for the `_id` field, if the `_id` field is not specified by the user.
- ObjectId is a 12-byte BSON type constructed using:
  - 4-byte value representing the seconds since the Unix time
  - 3-byte machine identifier
  - 2-byte process id
  - 3-byte counter, starting with a random value.
- In the mongo shell, you can access the creation time of the ObjectId, using the `getTimestamp()` method. Sorting on an `_id` field that stores ObjectId values is roughly equivalent to sorting by creation time.
- To generate a new ObjectId, use the `ObjectId()` constructor with no argument

July 31, 2018

Proprietary and Confidential

- 28 -

**To access the creation time**

```
> db.guests.find()[0]._id.getTimestamp()  
ISODate("2015-01-31T06:46:15Z")
```

**To print the time in current Locale**

```
> db.guests.find()[0]._id.getTimestamp().toLocaleDateString()  
Saturday, January 31, 2015
```

```
> db.guests.find()[0]._id.getTimestamp().toLocaleTimeString()  
12:16:15
```

**To create a new ObjectId**

```
> new ObjectId()  
ObjectId("54cc7fd2935e1001cd8a4d06")
```

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Updating Document(s)**

- 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

```
db.collection.update(  
  <query>,  
  <update>,  
  {  
    upsert: <boolean>,  
    multi: <boolean>  
})
```

- **upsert** : creates a new document when no document matches the query criteria.
- **multi** : updates multiple documents that meet the query criteria.

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Updating Document(s)**

➤ \$inc field update operator is used to increment and the \$set field update operator is used to replace the value of the field.

```
> db
SampleDB
> db.employees.find({_id:3861},{name:1,location:1})
{
  "_id": 3861,
  "name": {
    "first": "Veena",
    "last": "Deshpande"
  },
  "location": "Pune"
}
> db.employees.update({_id:3861},{$set:{location:'Bangalore'}})
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
> db.employees.find({_id:3861},{location:1})
{
  "_id": 3861,
  "location": "Bangalore"
}
>
> db.employees.update({_id:3861},{department:'Training'})
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
> db.employees.find({_id:3861})
{
  "_id": 3861,
  "department": "Training"
}
> db.employees.find({_id:1000})
>
> db.employees.update({_id:1000},{age:25})
WriteResult({
  "nMatched": 0,
  "nUpserted": 0,
  "nModified": 0
})
> db.employees.update({_id:1000},{age:25},true)
WriteResult({
  "nMatched": 0,
  "nUpserted": 1,
  "nModified": 0,
  "_id": 1000
})
>
> db.employees.find({_id:1000})
{
  "_id": 1000,
  "age": 25
}
>
> db.employees.update({_id:1000},{$inc:{age:3}})
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
> db.employees.find({_id:1000})
{
  "_id": 1000,
  "age": 28
}
```

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Updating Document - save**

➤ db.collection.save() updates an existing document (when \_id is present already) or inserts a new document (with new ObjectId)

```
> var karthik = db.employees.findOne({_id:714709})
> 
> karthik
{
  "_id": "714709",
  "department": "Training"
}
> 
> karthik.location = 'Bangalore'
> 
> karthik
{
  "_id": "714709",
  "department": "Training",
  "location": "Bangalore"
}
> 
> db.employees.findOne({_id:714709})
{
  "_id": "714709",
  "department": "Training"
}
> 
> db.employees.save(karthik)
WriteResult({
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
})
> 
> db.employees.findOne({_id:714709})
{
  "_id": "714709",
  "department": "Training",
  "location": "Bangalore"
}
> 
> var logith = {}
> 
> logith.name = 'Logith Karthik'
> 
> logith
{
  "name": "Logith Karthik"
}
> 
> logith.location = 'Bangalore'
> 
> logith
{
  "name": "Logith Karthik",
  "location": "Bangalore"
}
> 
> db.employees.findOne({name:'Logith Karthik'})
null
> 
> db.employees.save(logith)
WriteResult({
  "nInserted": 1
})
> 
> db.employees.findOne({name:'Logith Karthik'})
{
  "_id": ObjectId("54ce32b5dce2920aa624a657"),
  "name": "Logith Karthik",
  "location": "Bangalore"
}
```

July 31, 2018

Proprietary and Confidential

- 31 -

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Updating Document - findAndModify**

- db.collection.findAndModify(<document>) modifies and returns a single document. By default, the returned document does not include the modifications made on the update. To return the document with the modifications made on the update, use the new option.
- The findAndModify() method has the following form

```
db.collection.findAndModify({  
  query: <document>,  
  sort: <document>,  
  remove: <boolean>,  
  update: <document>,  
  new: <boolean>,  
  fields: <document>,  
  upsert: <boolean>  
});
```

**Instructor Notes:**

Add instructor notes here.

3.3: MongoDB Queries  
Updating Document - findAndModify

```
> db
> SampleDB
> db.employees.findAndModify({
... query : { name : 'Logith Karthik' },
... update : { $set : { age : 5 } },
... new : true });
{
  "_id" : ObjectId("54ce32b5dce2920aa624a657"),
  "name" : "Logith Karthik",
  "location" : "Bangalore",
  "age" : 5
}
>
> db.employees.findAndModify({
... query : { name : 'Logith Karthik' },
... update : { $set : { gender : 'Male' } },
... new : false });
{
  "_id" : ObjectId("54ce32b5dce2920aa624a657"),
  "name" : "Logith Karthik",
  "location" : "Bangalore",
  "age" : 5
}
> db.employees.findOne({name : 'Logith Karthik'});
{
  "_id" : ObjectId("54ce32b5dce2920aa624a657"),
  "name" : "Logith Karthik",
  "location" : "Bangalore",
  "age" : 5,
  "gender" : "Male"
}
```

July 31, 2018

Proprietary and Confidential

- 33 -



**Instructor Notes:**

Add instructor notes here.

### 3.3: MongoDB Queries Auto-Incrementing Sequence Field



- In MongoDB, by default we cannot use an auto-increment pattern for the `_id` field, or any field, because it does not scale for databases with large numbers of documents. Typically the default value `ObjectId` is more ideal for the `_id`.
- But still we can create an auto-Incrementing sequence field by creating a collection on our own to maintain the sequence.

```
function generateSequence(name) {
  var ret = db.counters.findAndModify(
    {
      query: { _id: name },
      update: { $inc: { seq: 1 } },
      new: true,
      upsert:true
    }
  );
  return ret.seq;
}
```

- By calling `generateSequence('test')` will create a document under `counters` collection and maintains the sequence for the same.

July 31, 2018 | Proprietary and Confidential | - 34 -

```
> function generateSequence(name){
  var ret = db.counters.findAndModify(
    {
      query: { _id: name },
      update: { $inc: { seq: 1 } },
      new: true,
      upsert:true
    }
  );
  return ret.seq;
}
> generateSequence('test')
1
> generateSequence('test')
2
> generateSequence('product')
1
> generateSequence('product')
2
> db.departments.insert({_id:generateSequence('department'),name:'IT'})
> db.departments.insert({_id:generateSequence('department'),name:'Training'})
> db.departments.find()
{ "_id": 1, "name": "IT" }
{ "_id": 2, "name": "Training" }

> db.counters.find()
{ "_id": "test", "seq": 2 }
{ "_id": "product", "seq": 2 }
{ "_id": "department", "seq": 2 }
```

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Array Update Operators**

Name	Description
\$	Acts as a placeholder to update the first element that matches the query condition in an update.
\$addToSet	Adds elements to an array only if they do not already exist in the set.
\$pop	Removes the first or last item of an array.
\$pull	Removes all matching values from an array.
\$pullAll	Removes all array elements that match a specified query.
\$pushAll	Deprecated. Adds several items to an array.
\$push	Adds an item to an array.

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Array Update Operators – Adding array items**

```
> db
SampleDB
> db.persons.findOne({name:'Karthik'})
{
  "_id": ObjectId("54ce287fdce2920aa624a655"),
  "name": "Karthik"
}
> db.persons.update({name:'Karthik'}, {$set:{hobbies:['Programming']}}, true)
WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.persons.update({name:'Karthik'}, {$push: { hobbies:'Music' } })
WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.persons.update({name:'Karthik'}, {$pushAll:{hobbies:['Cricket','Chess']}})
WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.persons.findOne({name:'Karthik'})
{
  "_id": ObjectId("54ce287fdce2920aa624a655"),
  "name": "Karthik",
  "hobbies": [
    "Programming",
    "Music",
    "Cricket",
    "Chess"
  ]
}
> db.persons.update({name:'Karthik'}, {$addToSet:{hobbies:'Cricket'}})
WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 0 })
> db.persons.update({name:'Karthik'}, ... {$addToSet: { hobbies: { $each: ['Music','Chess','Tennis'] } } })
WriteResult({ "nMatched": 1, "nUpserted": 0, "nModified": 1 })
> db.persons.findOne({name:'Karthik'})
{
  "_id": ObjectId("54ce287fdce2920aa624a655"),
  "name": "Karthik",
  "hobbies": [
    "Programming",
    "Music",
    "Cricket",
    "Chess",
    "Tennis"
  ]
}
```

July 31, 2018

Proprietary and Confidential

- 36 -



**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Array Update Operators – Removing array items**

```
> db.persons.findOne({name:'Karthik'})
{
    "_id" : ObjectId("54ce287fdce2920aa624a655"),
    "name" : "Karthik",
    "hobbies" : [
        "Programming",
        "Music",
        "Cricket",
        "Chess",
        "Tennis"
    ]
}
> db.persons.update({name:'Karthik'},{$pull:{hobbies:'Music'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.persons.update({name:'Karthik'},{$pullAll:{hobbies:['Cricket','Chess']}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.persons.findOne({name:'Karthik'})
{
    "_id" : ObjectId("54ce287fdce2920aa624a655"),
    "name" : "Karthik",
    "hobbies" : [
        "Programming",
        "Tennis"
    ]
}
> db.persons.update({name:'Karthik'},{$pop:{hobbies:-1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.persons.findOne({name:'Karthik'})
{
    "_id" : ObjectId("54ce287fdce2920aa624a655"),
    "name" : "Karthik",
    "hobbies" : [
        "Tennis"
    ]
}
```

July 31, 2018

Proprietary and Confidential

- 37 -

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Renaming and Deleting fields and documents**

- The \$unset field update operator is used to delete a particular field.
- The \$rename field update operator updates the name of a field
- db.collection.remove() is used to remove documents from a collection

```
> db.persons.findOne({name:'Karthik'})  
< _id: ObjectId('54ce287fdce2920aa624a655'),  
  "name": "Karthik",  
  "hobbies": [  
    "Tennis"  
  ]  
> db.persons.update({name:'Karthik'}, { $rename : { 'hobbies' : 'intrests'}})  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
> db.persons.findOne({name:'Karthik'})  
< _id: ObjectId('54ce287fdce2920aa624a655'),  
  "name": "Karthik",  
  "intrests": [  
    "Tennis"  
  ]  
> db.persons.update({name:'Karthik'}, { $unset : { 'intrests' : ''}})  
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
> db.persons.findOne({name:'Karthik'})  
< _id: ObjectId('54ce287fdce2920aa624a655'), "name": "Karthik" >  
> db.persons.find({name:'Karthik'}).count()  
1  
> db.persons.remove({name:'Karthik'})  
WriteResult({ "nRemoved" : 1 })  
> db.persons.find({name:'Karthik'}).count()  
0
```

July 31, 2018

Proprietary and Confidential

- 38 -

**Instructor Notes:**

Add instructor notes here.

3.3: MongoDB Queries  
Aggregation

- db.collection.aggregate() calculates aggregate values for the data in a collection.

```
> db.employees.aggregate(
...   {
...     $group: { _id : "$location", employeeCount : { $sum : 1 } }
...   },
...   {
...     $sort: { employeeCount : -1 }
...   }
... );
< "_id" : "Pune", "employeeCount" : 24 >
< "_id" : "Mumbai", "employeeCount" : 23 >
< "_id" : "Bangalore", "employeeCount" : 21 >
< "_id" : "Chennai", "employeeCount" : 7 >

>
> db.employees.aggregate(
...   {
...     $match : { location : "Bangalore" }
...   },
...   {
...     $group : { _id : "$location", employeeCount : { $sum : 1 } }
...   }
... );
< "_id" : "Bangalore", "employeeCount" : 21 >
```

**Instructor Notes:**

Add instructor notes here.

3.3: MongoDB Queries  
Stored JavaScript

```
C:\Windows\system32\cmd.exe - mongo.exe
> use guestDB
switched to db guestDB
> db.system.js.save({"_id": "generateSequence", "value": function(name) {
...   var ret = db.counters.findAndModify(
...     {
...       query: { _id: name },
...       update: { $inc: { seq: 1 } },
...       new: true,
...       upsert:true
...     }
...   );
...   return ret.seq;
... });
... >>>
WriteResult({
  "nMatched" : 0,
  "nUpserted" : 1,
  "nModified" : 0,
  "_id" : "generateSequence"
})
> db.eval("generateSequence('userid')")
1
> db.loadServerScripts();
> generateSequence('userid')
2
```

- Do not store application logic in the database. There are performance limitations to running JavaScript inside of MongoDB.

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Indexing**

- Indexing is an important part of database management.
- In MongoDB, if we do a query with non-indexed field, it uses "BasicCursor". BasicCursor indicates a full collection scan where as "BtreeCursor" indicates that the query used is an index field.
- `explain()` method returns a document that describes the process used to return the query results. MongoDB stores its indexes in `system.indexes` collection. We can view the indexes of DB using `db.system.indexes.find()`

```
> db.employees.find({name.first:'Ueena'}).explain()
{
  "cursor": "BasicCursor",
  "isMultiKey": false,           Searching with Non-indexed Field
  "n": 1,
  "nscanned": 67,
  "nscannedObjects": 67,
  "nscannedObjectsAllPlans": 67,
  "nscannedAllPlans": 67
}
> db.employees.find({_id:3861}).explain()
{
  "cursor": "IDCursor",
  "n": 1,                      Searching with Indexed Field
  "nscanned": 1,
  "nscannedObjects": 1,
  "nscannedObjectsAllPlans": 1,
  "nscannedAllPlans": 1,
  "millis": 0
}
```

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Indexing**

- To create an index on the specified field if the index does not already exist.
  - db.<collection>.ensureIndex({<field>:1(asc)/-1(desc)})
- To create a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index.
  - db.<collection>.ensureIndex({<field>:1(asc)/-1(desc)},{unique: true})
- To creates a unique index on a field that may have duplicates
  - db.<collection>.ensureIndex({<field>:1(asc)/-1(desc)},{unique: true, dropDups:true})
- To create an Index on a Multiple Fields
  - db.collection.ensureIndex( {<field1>:1(asc)/-1(desc), <field2>:1(asc)/-1(desc) } )
- To create index which only references the documents with specified field
  - db.employees.ensureIndex({nonexistfield:1},{sparse:true});
- To drop index
  - db.employees.dropIndex('<indexname>')

**Instructor Notes:**

Add instructor notes here.

**3.3: MongoDB Queries  
Indexing**

```
> db.employees.find({name.first:'Veena'})
< "_id" : 3861, "name" : { "first" : "Veena", "last" : "Deshpande" }, "doj" : "1998-05-07T00:00:00.000Z", "location" : "Pune", "isactive" : true, "email" : "veena.deshpande@gate.com", "qualifications" : [ "M.Sc(CSE)", "M.E(Electronics)" ]
> db.employees.ensureIndex({name.first:1},{unique: true, dropDups:true})
<
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
>
> db.employees.find({name.first:'Veena'}).explain()
<
    "cursor" : "BtreeCursor name.first_1",
    "isMultiKey" : false,
    "n" : 1,
    "nscannedObjects" : 1,
    "nscanned" : 1,
    "nscannedAllPlans" : 1,
    "server" : "BLRMFL2913:27017",
    "filterSet" : false
>
> db.system.indexes.find()
< {"v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "SampleDB.employees" }
< {"v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "SampleDB.locations" }
< {"v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "SampleDB.persons" }
< {"v" : 1, "key" : { "_id" : 1 }, "name" : "_id_", "ns" : "SampleDB.guests" }
< {"v" : 1, "unique" : true, "key" : { "name.first" : 1 }, "name" : "name.first_1" },
    "ns" : "SampleDB.employees", "dropDups" : true
> db.employees.dropIndex("name.first_1")
< "nIndexesWas" : 2, "ok" : 1 }
```

**Instructor Notes:**

Add instructor notes here.

**3.4: Mongoose Library**  
**Introduction to Mongoose Library**

- There are multiple ways to connect to MongoDB from Node.js.
- The native driver's npm package is simply `mongodb`. It provides a fundamental connectivity and data manipulation APIs
- The most popular ways to connect MongoDB is by using the Node.js native driver and the `Mongoose.js ODM`.
- `Mongoose` is an object data modeling (ODM) library that provides a modeling environment for data which enforces the structure as needed. It is the officially supported ODM for Node.js.
- Using `Mongoose` we can define schemas for MongoDB collections, which are then enforced at the ODM layer.
- `Mongoose` makes it even easier to use MongoDB with Node.js.

July 31, 2018

Proprietary and Confidential

- 44 -

`Mongoose` has a thriving open source community and includes advanced schema-based features such as `asyncvalidation`, `casting`, `object life-cycle management`, `pseudo-joins`, and `rich query builder support`

**Instructor Notes:**

Add instructor notes here.

3.4: Mongoose Library

**Connecting MongoDB using Mongoose Library**

- Connecting to a MongoDB instance with Mongoose is straightforward, requiring only the resource URL of the database.
- `mongoose.connect()` is used to connect the MongoDB
- To establish connection in `app.js`

```
var mongoose = require('mongoose');
mongoose.connect(<uri>, function (err, res) {
  if (err) {
    console.log ('ERROR connecting to MongoDB : ' + err);
  }
  else {
    console.log ('Connected to: MongoDB');
  }
});
```

**Instructor Notes:**

Add instructor notes here.

**3.4: Mongoose Library  
Object Modeling**

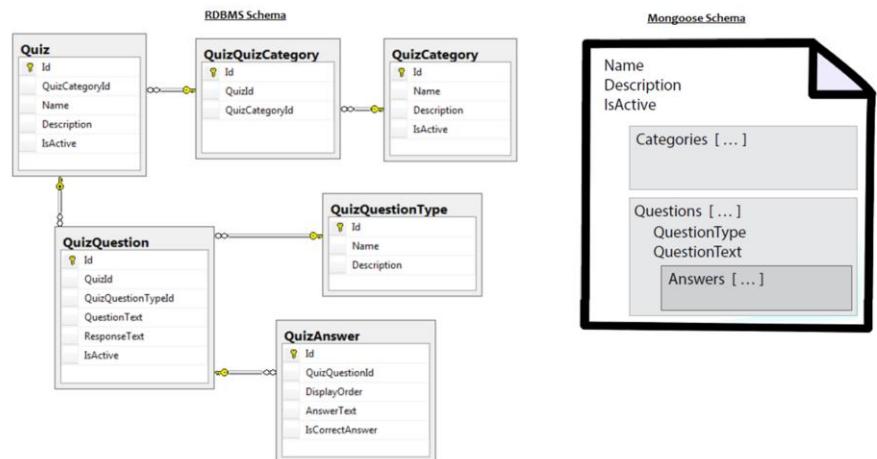
- MongoDB has a flexible schema that allows for variability between different documents in the same collection. But validations were not enforced.
- Using Mongoose we can perform this when new objects are getting created.
- Models are defined by passing a Schema instance to mongoose.model()

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var userSchema = new mongoose.Schema({
  name: {type: String, trim: true},
  age: { type: Number, min: 0 },
  company: { type: String, default: 'IGATE' }
});
```

- To instantiate a new user, create a Mongoose model from the schema in the users collection(MongoDB) so that we can populated user's details.
- var User = mongoose.model('users', userSchema);

**Instructor Notes:**

Add instructor notes here.

**3.4: Mongoose Library  
RDBMS Schema vs Mongoose Schema**

**Instructor Notes:**

Add instructor notes here.

3.4: Mongoose Library  
Simple Schema

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var customerSchema = new Schema({
    name:      String,
    address:   String,
    city:      String,
    state:     String,
    country:   String,
    zipCode:   Number,
    createdOn: Date,
    isActive:  Boolean
});

var simpleSchema = new Schema({ fieldName: SchemaType });
```



**Instructor Notes:**

Add instructor notes here.

**3.4: Mongoose Library  
Complex Schema**

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

// child address schema...
var addressSchema = new Schema({
  type: String,
  street: String,
  city: String,
  state: String,
  country: String,
  postalCode: Number
});

// parent customer schema...
var customerSchema = new Schema({
  name: {
    first: String,
    last: String
  },
  address: [ addressSchema ],
  createdOn: { type: Date, default: Date.now },
  isActive: { type: Boolean, default: true},
});

```



**Instructor Notes:**

Add instructor notes here.

**3.4: Mongoose Library  
Validations in Model**

- Validation is defined in the Schema, it's an internal piece of middleware
- Validation occurs when a document attempts to be saved, after defaults have been applied.
- The path and value that triggered the error can be accessed in the ValidationError object.
- For Strings *enum* and for Numbers *min* and *max* are built in validators.
  - var Person = new Schema({ gender : { type: String, enum: ['M', 'F', 'T'] } })
  - var Person = new Schema({ age : { type: Number, min: 1, max : 100 } })
- We can also validate a value against a regular expression
  - var schema = new Schema({ name: { type: String, validate: /[a-zA-Z]+/ } });
- Validation is asynchronously recursive. i.e. when Model#save is called, embedded documents validation is executed. If an error occurs Model#save callback receives it.

July 31, 2018

Proprietary and Confidential

- 50 -

**Simple validation**

Simple validation is declared by passing a function to validate and an error type to your SchemaType

```
function validator(v){  
  return v.length > 5;  
}  
  
new Schema({  
  name: { type: String, validate: [validator, 'my error type'] }  
})
```

**Asynchronous validation**

If you define a validator function with two parameters, like:

```
schema.path('name').validate(function (v, fn) {  
  // my logic  
}, 'my error type');
```

Then the function fn has to be called with true or false, depending on whether the validator passed. This allows for calling other models and querying data asynchronously from your validator.

**Instructor Notes:**

Add instructor notes here.

**3.4: Mongoose Library  
Finding documents**

➤ Documents can be retrieved through `find`, `findOne` and `findById`. These methods are executed on Models.

➤ `Model.find`

- `Model.find(query, fields, options, callback)` // fields and options can be omitted
- `Model.find({ '_id': 5 }, 'name age', function (err, docs) { // docs is an array });`

➤ `Model.findOne` : receives a single document as second parameter

- `Model.findOne({ age: 5}, function (err, doc){ // doc is a Document });`

➤ `Model.findById` : Same as `findOne`, but receives a value to search a document by their `_id` key

- `Model.findById(obj._id, function (err, doc){ // doc is a Document });`

July 31, 2018

Proprietary and Confidential

- 51 -

**Model.count** : Counts the number of documents matching conditions.

`Model.count(conditions, callback);`

**Model.remove** : Removes documents matching conditions.

`Model.remove(conditions, callback);`

**Model.distinct** : Finds distinct values of field for documents matching conditions.

`Model.distinct(field, conditions, callback);`

**Model.\$where** : Querying Mongodb using a JavaScript expression

`Model.$where('this.firstname === this.lastname').exec(callback)`

**Instructor Notes:**

Add instructor notes here.

**3.4: Mongoose Library  
Query API**

- A Query is what is returned when calling any Model methods.
- Query objects provide a chaining api to specify search terms, cursor options, hints and other behavior.

```
var query = Model.find({});  
query.where('field', 5);  
query.limit(5);  
query.skip(100);  
  
query.exec(function (err, docs) {  
    // called when the `query.complete` or `query.error` are called internally  
});
```

- Each of these methods returns a Query. If you don't pass a callback to these methods, the Query can be continued to be modified (such as adding options, fields, etc), before it's executed.

July 31, 2018

Proprietary and Confidential

- 52 -

**For Code Snippets refer : Mongoose ODM-Queries.pdf**

**Instructor Notes:**

Demo

nodewithmongo



July 31, 2018 | Proprietary and Confidential | - 53 -

**Instructor Notes:**

Add instructor notes here.

**Lab****Lab 2**

July 31, 2018

Proprietary and Confidential

- 54 -

Add the notes here.