MongoDB Assignment

- Install mongodb. Create data folder. Run mongod server and access it using mongo shell.
    - Basic CRUD operations
    - Create “blog” collection to store blog entry details and comments.
    - Insert a blog entry.
    - Update the comments in the entry.
    - Remove the entry.
- Practice the commands shared in the hands-on document
    - Create Employees Collection
    - Create Indexes
    - Insert Data/Update/Delete
    - Query data with conditions

DOCUMENT STORE - MongoDB

MongoDb is an Open Source database written in C++.

If the load increases, by adding more nodes (computers), the performance can be retained.
It can be used to store data for very high performance applications.
MongoDb stores data as documents. So it is a document oriented database.

To get started, there are six simple concepts we need to understand.

1. MongoDB has the same concept of a database with which you are likely already familiar. Within a MongoDB instance you can **have zero or more databases**, each acting as high-level containers for everything else.

2. A database **can have zero or more collections**. A collection shares enough in common with a traditional **table** that you can safely think of the two as the same thing.

3. **Collections** are made up of zero or more **documents**. Again, a document can safely be thought of as a **row**.

4. A **document** is made up of one or more **fields/keys**, which you can probably guess are a lot like columns. Each field/key is assigned a value.

5. **Indexes** in MongoDB function much like their RDBMS counterparts

6. **Cursors** are different than the other five concepts but they are important enough, and often overlooked, that I think they are worthy of their own discussion. The important thing to understand about **cursors** is that when you ask MongoDB for data, **it returns a**

cursor, which we can do things to, such as counting or skipping ahead, without actually pulling down data.

Storage model

RDBMS Vs. MongoDB

Table: Collection
Records / Rows: Document / Object
Column: Key
Value: Value

The core difference comes from the fact that **relational databases define columns at the table level** whereas a **document-oriented database defines its fields at the document level**. That is to say that each document within a collection can have its own unique set of fields.

In a relational database like MySQL, a schema defines the organization / structure of data in database. **MongoDB is a Schema-free database**.

MongoDB does not require such a set of formula defining structure of data. So, it is quite possible to store documents of varying structures in a collection.

Practically, you **don't need to define a column and it's datatype** unlike in RDBMS, while working with MongoDB.

Step-1: start the server

```
C:\Program Files\MongoDB\Server\3.4\bin>mongod
2017-11-14T16:55:34.235+0530 I CONTROL  [initandlisten] MongoDB starting : pid=2
760 port=27017 dbpath=C:\data\db\ 64-bit host=Priyanka-NewPC
2017-11-14T16:55:34.237+0530 I CONTROL  [initandlisten] targetMinOS: Windows 7/W
indows Server 2008 R2
2017-11-14T16:55:34.238+0530 I CONTROL  [initandlisten] db version v3.4.7
2017-11-14T16:55:34.238+0530 I CONTROL  [initandlisten] git version: cf38c1b8a0a
8dca4a11737581beafef4fe120bcd
2017-11-14T16:55:34.239+0530 I CONTROL  [initandlisten] OpenSSL version: OpenSSL
 1.0.1u-fips  22 Sep 2016
2017-11-14T16:55:34.239+0530 I CONTROL  [initandlisten] allocator: tcmalloc
2017-11-14T16:55:34.240+0530 I CONTROL  [initandlisten] modules: none
2017-11-14T16:55:34.240+0530 I CONTROL  [initandlisten] build environment:
2017-11-14T16:55:34.241+0530 I CONTROL  [initandlisten]     distmod: 2008plus-ss
l
2017-11-14T16:55:34.242+0530 I CONTROL  [initandlisten]     distarch: x86_64
2017-11-14T16:55:34.243+0530 I CONTROL  [initandlisten]     target_arch: x86_64
2017-11-14T16:55:34.244+0530 I CONTROL  [initandlisten] options: {}
2017-11-14T16:55:34.255+0530 W -        [initandlisten] Detected unclean shutdow
n - C:\data\db\mongod.lock is not empty.
```

```
2017-11-14T16:55:34.267+0530 I -        [initandlisten] Detected data files in C
:\data\db\ created by the 'wiredTiger' storage engine, so setting the active sto
rage engine to 'wiredTiger'.
2017-11-14T16:55:34.268+0530 W STORAGE  [initandlisten] Recovering data from the
 last clean checkpoint.
2017-11-14T16:55:34.273+0530 I STORAGE  [initandlisten] wiredtiger_open config:
create,cache_size=7645M,session_max=20000,eviction=(threads_min=4,threads_max=4)
,config_base=false,statistics=(fast),log=(enabled=true,archive=true,path=journal
,compressor=snappy),file_manager=(close_idle_time=100000),checkpoint=(wait=60,lo
g_size=2GB),statistics_log=(wait=0),
2017-11-14T16:55:35.379+0530 I CONTROL  [initandlisten]
2017-11-14T16:55:35.380+0530 I CONTROL  [initandlisten] ** WARNING: Access contr
ol is not enabled for the database.
2017-11-14T16:55:35.381+0530 I CONTROL  [initandlisten] **          Read and wri
te access to data and configuration is unrestricted.
2017-11-14T16:55:35.382+0530 I CONTROL  [initandlisten]
2017-11-14T16:55:35.660+0530 I FTDC     [initandlisten] Initializing full-time d
iagnostic data capture with directory 'C:/data/db/diagnostic.data'
2017-11-14T16:55:35.662+0530 I NETWORK  [thread1] waiting for connections on por
t 27017
2017-11-14T16:55:36.084+0530 I FTDC     [ftdc] Unclean full-time diagnostic data
 capture shutdown detected, found interim file, some metrics may have been lost.
 OK
```

Note: In case clean shutdown is not performed previously, you will not be able to start the server. In such cases you need to repair using:

C:\mongodb\bin>mongod –repair

Step-2: Start Mongo Shell

C:\>cd mongodb
C:\mongodb>cd bin
Next using mongo command you will be displaying mongo shell
C:\mongodb\bin>mongo

```
C:\Program Files\MongoDB\Server\3.4\bin>mongo
MongoDB shell version v3.4.7
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.7
Server has startup warnings:
2017-11-14T16:55:35.379+0530 I CONTROL  [initandlisten]
2017-11-14T16:55:35.380+0530 I CONTROL  [initandlisten] ** WARNING: Access control is not
enabled for the database.
2017-11-14T16:55:35.381+0530 I CONTROL  [initandlisten] **          Read and write access to
data and configuration is unrestricted.
2017-11-14T16:55:35.382+0530 I CONTROL  [initandlisten]
>
```

To view help at Mongo Shell

The **mongo** shell provides various help. The following table displays some common help methods and commands:

Help Methods and Commands

Description help Show help.

db.help()

Shows help for database methods.

db.<collection>.help() Show help on collection methods.
The**<collection>** can be the name of an existing collection or a non-existing collection.

show dbs

Print a list of all databases on the server.

use <db>

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

show collections

Print a list of all collections for current database

show users

Print a list of users for current database.

show profile

Print the five most recent operations that took 1 millisecond or more. See documentation on the database profiler for more information.

show databases

Print a list of all available databases.

load() Execute a JavaScript file. See Getting Started with the mongo Shell for more information.

Note: Because this is a JavaScript shell, if you execute a method and omit the parentheses (), you'll see the method body rather than executing the method. For example, if you enter db.help (without the parentheses), you'll see the internal implementation of the help method.

> help

db.help()

help on db methods

db.mycoll.help()

help on collection methods

rs.help()
help on replica set methods

help admin
administrative help

help connect
connecting to a db help

help keys
key shortcuts

help misc
misc things to know

help mr
mapreduce

show dbs
show database names

show collections
show collections in current database

show users
show users in current database

show profile
show most recent system.profile entries with time >= 1ms

show logs
show the accessible logger names

show log [name]
prints out the last segment of log in memory, 'global' is default

use <db_name>
set current database

db.foo.find()
list objects in collection foo

db.foo.find({ a : 1 })
list objects in foo where a == 1

it
result of the last line evaluated; use to further iterate

DBQuery.shellBatchSize = x
set default number of items to display on shell

exit
quit the mongo shell

> **show users**

> **show dbs**
local (empty)
prefs 0.03125GB

> **show logs**

global
startupWarnings

Step-3: create database and insert data

Create organization database

>use organization

Note: --need not be existing
switched to db organization

To know which database you are using currently
db

Next to insert records

```
a={ename:'SMITH',job:'CLERK',sal:800,deptno:20};
b={ename:'ALLEN',job:'SALESMAN',sal:1600,comm:300,deptno:30};
db.employees.save(a);
db.employees.save(b);
```

Alternate method using insert() method:

Used following statement in oracle for generating statement

```
select 'db.employees.insert({ename:'''||ename||''',job:'''||job
||''',sal:'||sal||',hiredate:'''||to_char(hiredate,'dd/mm/yyyy')
||''',comm:'||comm||',deptno:'||deptno||'})' from emp;
```

Following statements would add rest of the records

```
db.employees.insert(
{
ename:'WARD',
job:'SALESMAN',
sal:1250,
hiredate:'22/02/1981',
comm:500,
deptno:30
}
)

db.employees.insert(
{
ename:'JONES',
job:'MANAGER',
sal:2975,
hiredate:'02/04/1981',
deptno:20
}
)

db.employees.insert(
{
ename:'MARTIN',
job:'SALESMAN',
sal:1250,
hiredate:'28/09/1981',
comm:1400,
```

```
deptno:30
}
)

db.employees.insert(
{
ename:'BLAKE',
job:'MANAGER',
sal:2850,
hiredate:'01/05/1981',
deptno:30
}
)

db.employees.insert(
{
ename:'CLARK',
job:'MANAGER',
sal:2450,
hiredate:'09/06/1981',
deptno:10
}
)

db.employees.insert(
{
ename:'SCOTT',
job:'ANALYST',
sal:3000,
hiredate:'19/04/1987',
deptno:20
}
)

db.employees.insert(
{
ename:'KING',
job:'PRESIDENT',
sal:5000,
hiredate:'17/11/1981',
deptno:10
}
)
```

```
db.employees.insert(
{
ename:'TURNER',
job:'SALESMAN',
sal:1500,
hiredate:'08/09/1981',
deptno:30
}
)

db.employees.insert(
{
ename:'ADAMS',
job:'CLERK',
sal:1100,
hiredate:'23/05/1987',
deptno:20
}
)

db.employees.insert(
{
ename:'JAMES',
job:'CLERK',
sal:950,
hiredate:'03/12/1981',
deptno:30
}
)

db.employees.insert(
{
ename:'FORD',
job:'ANALYST',
sal:3000,
hiredate:'03/12/1981',
deptno:20
}
)

db.employees.insert(
{
ename:'MILLER',
job:'CLERK',
```

```
sal:1300,
hiredate:'23/01/1982',
deptno:10
}
)
```

List all the records in collection "employees"

```
> db.employees.find();
{ "_id" : ObjectId("5232f172272399812e5ec52b"), "ename" : "smith", "job" :
"clerk", "sal" : 800, "deptno" : 20 }
{ "_id" : ObjectId("5232f17d272399812e5ec52c"), "ename" : "allen", "job" :
"salesman", "sal" : 1600, "comm" : 300, "deptno" : 30 }
.....
```

To remove all the records

```
>db.employees.remove();
```

Caution: The above command deletes the entire contents of the database

To view collections in current database

```
show collections
employees
system.indexes  // deprecated after version 3.0, Since we are using v 3.4 this is changed
```

The collection "**system.indexes**" is created once per database and contains the information on our database's index.
Notice that, in addition to the data you specified, there's an _id field. Every document must have a unique _id field.

You can either generate one yourself or let MongoDB generate an ObjectId for you. Most of the time you'll probably want to let MongoDB generate it for you.

By default, the _id field is indexed - which explains why the system.indexes collection was created.

You can look at system.indexes:

```
db.system.indexes.find() //deprecated
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "employees.employees", "name" : "_id_" }
```

To view explicitly created indexes

```
db.employees.getIndexes()
```

For Creating Index
db.employees.createIndex({sal:1})

For explicitly creating an index
db.employees.ensureIndex({ename:1},{unique:true});

db.employees.getIndexes()

```
[
{
"v" : 1,
"key" : {
"_id" : 1
},
"ns" : "organization.employees",
"name" : "_id_"
},
{
"v" : 1,
"key" : {
"ename" : 1
},
"unique" : true,
"ns" : "organization.employees",
"name" : "ename_1"
}
]
```

To view indexes of specific database and collection

db.system.indexes.find({ ns: "organization.employees" }) //deprecated

To drop specific index

db.employees.dropIndex({ename:1});

To drop all indexes
db.employees.dropIndexes();

QUERY SELECTORS

Searching For Records - Selectors
"**Selectors**" are to Mongo what "**where clauses**" are to SQL.
As with where clauses,

Mongo selectors allow us to do the following:
- specify criteria that **MUST** match. i.e., an **AND** clause
- specify criteria that **CAN** optionally match. i.e., an **OR** clause
- specify criteria that **MUST** exist
- and much more…

A selector is a **JSON** object, the simplest of which is **{}** which matches all documents.

Comparison operators

Name	Description
$gt	greater than
$gte	equal to or greater than
$in	Values that exist in an array specified in the query.
$lt	less than
$lte	less than or equal to
$ne	not equal to
$nin	**do not** exist in an array specified to the query.

Logical Operators

Name	Description
$or	OR
$and	AND
$not	NOT

Records That MUST Match

Let's start with a basic selector. Say that we want to find all employees working in **dept 20**. To accomplish that, you'll need to run the following command:

If we wanted to find all employees in dept 20:

```
db.employees.find({deptno:20});
```

To list employees earning salary 800

```
db.employees.find({sal:800});
```

To list employees with salary > 1000

```
db.employees.find({sal:{$gt:1000}});
```

To find number of employees earning salary > 700

```
db.employees.find({sal:{$gt:700}}).count();
2
```

To list employees earning salary > 900 or employee name=smith

```
db.employees.find({$or:[{ename:"smith"},{sal:{$gt:900}}]});
```

To List employees earning salary > 700 and ename is smith

```
db.employees.find({ename:"smith",sal:{$gt:700}});
```

NOTE: In MongoDB, restricting the output to a few columns or attributes is not a smart strategy because each document is always returned in its entirety with all its attributes when fetched.

Field Selection

find takes a second optional parameter called "projection". This parameter is the list of fields we want to retrieve or exclude

To Display only ename column

```
db.employees.find(null,{ename:1})
```

```
{ "_id" : ObjectId("5232f172272399812e5ec52b"), "ename" : "SMITH" }
{ "_id" : ObjectId("5232f17d272399812e5ec52c"), "ename" : "ALLEN" }
```

Note : By default, the _id field is always returned. We can explicitly exclude it by specifying {name:1, _id: 0}.

To avoid displaying of objectId

```
db.employees.find(null,{ename:1,_id:0})
```

```
{ "ename" : "smith" }
{ "ename" : "allen" }
```

Aside from the _id field, you cannot mix and match inclusion and exclusion

USING CURSORS

The result from find is a cursor.

```
var cursor=db.employees.find()
while(cursor.hasNext()) printjson(cursor.next())
{
"_id" : ObjectId("5232f172272399812e5ec52b"),
"ename" : "SMITH",
"job" : "CLERK",
"sal" : 800,
"deptno" : 20
}
{
"_id" : ObjectId("5232f17d272399812e5ec52c"),
"ename" : "ALLEN",
"job" : "SALESMAN",
"sal" : 1600,
```

"comm" : 1200,
"deptno" : 30
}

find returns a cursor whose execution is delayed until needed. However, what you've no doubt observed from the shell is that find executes immediately. This is a behavior of the shell only. We can observe the true behavior of cursors by looking at one of the methods we can chain to find. The first that we'll look at is sort. We specify the fields we want to sort on as a JSON document, using 1 for ascending and -1 for descending.

Ordering Records

What if we want to sort records, say by first name or nationality? Similar to SQL, Mongo provides the sort command. The command, like the find command takes a list of options to determine the sort order.
Unlike SQL, however we specify ascending and descending differently. We do that as follows:
Ascending: 1
Descending: -1:

To sort employee documents sorted on deptno in ascending

```
db.employees.find().sort({deptno: 1});
```

To list SALESMEN working in dept 30 sorted on ename in ascending

```
db.employees.find({deptno: 30, $or: [{job: 'SALESMAN'}]}).sort({ename: 1});
```

Note : MongoDB limits the size of your sort without an index. That is, if you try to sort a very large result set which can't use an index, you'll get an error!

Limiting Records

What if we had a pretty big data set and we wanted to limit the results to just 2?
Mongo provides the limit command, similar to MySQL.

```
db.employees.find().limit(2);
```

Restricting number of rows along with restrictive condition

```
db.employees.find({deptno: 30, $or: [{job: 'SALESMAN'}]}).sort({ename: 1}).limit(2);
```

If we wanted the third and fourth records, i.e., skip over the first two?

```
db.employees.find().limit(2).skip(2);
```

Count

The shell makes it possible to execute a count directly on a collection, such as:
db.employees.count({sal: {$gt: 600}})

To count all the documents in a collection
db.employees.count()

Updating Records ($set,$inc, upsert)

As expected, MongoDB provides an option to update records as well.
As with the find method and SQL queries, you need to specify the criteria for the record that you want to modify, then the data in that record that's going to be modified.

$set

update takes two parameters: the selector (where) to use and what updates to apply to fields

Let's say that we need to update the record SMITH specifying that his SAL=900. Well for that we run the update function.

db.employees.update({ename: 'SMITH'}, {$set: { sal : 900 , job:'MANAGER'}});

$inc

to increment a field by a certain positive or negative amount.

db.employees.update({name: 'SMITH'}, {$inc: {sal: -200}})

Upserts

An upsert updates the document if found or inserts it if not.
To enable upserting we pass a third parameter to update {upsert:true}.

A mundane example is a hit counter for a website. If we wanted to keep an aggregate count in real time, we'd have to see if the record already existed for the page, and based on that decide to run an update or insert.

With the upsert option omitted (or set to false), executing the following won't do anything:

db.employees.update({dept: 'ABC'}, {$inc: {sal: 50}}); db.employees.find();

However, if we add the upsert option, the results are quite different:

```
db.employees.update({dept: 'ABC'}, {$inc: {sal: 50}}, {upsert:true});
db.employees.find();
```

Since no documents exists with a field dept equal to ABC, a new document is inserted. If we execute it a second time, the existing document is updated and hits is incremented to 100.

```
db.employees.update({dept: 'ABC'}, {$inc: {sal: 50}}, {upsert:true});
db.employees.find();
```

Multiple Updates

by default, update a single document
So we have to use multi:true for multiple updates

```
db.employees.update({}, {$set: {currently_working: true }}, {multi:true});
db.employees.find({currently_working: true});
```

Deleting Records

If you want to delete a record, you have to pass in a set of selectors, as you also would with SQL, to determine the set of records to delete.

```
db.employees.remove({ename: 'SMITH'});
```

To remove all the documents

```
db.employees.remove();
```

Caution: If you don't do this, you will delete all records – and the database.

Remove a collection using drop()

```
db. employees.drop()
```

Remove an entire Database using dropDatabase()

```
db.dropDatabase()
```