MongoDB

Naveen Kumar K S
Adith.naveen@gmail.com

Introduction

- MongoDB is an open-source document database.

  Humongous DB

- Falls into the category of NOSQL databases

Introduction

- Stores the data in un-normalized format
- Data is stored as collections of documents
- MongoDB works on concept of collection and document.

Document database

- A record in an RDBMS is equivalent to a document in MongoDB
  - A data structure composed of field and value pairs.
- MongoDB documents are similar to JSON objects.
- The values of fields may include scalar data, other documents, arrays, and arrays of documents.
- Internally, MongoDB stores these documents in the binary format, called BSON (Binary JSON)

Terminology

- Database is a physical container for collections. A single MongoDB server typically has multiple databases.

Terminology

- Collection is a group of MongoDB documents.
- It is the equivalent of an RDBMS table.
- A collection exists within a single database.
- Collections do not enforce a schema. Documents within a collection can have different fields.
- Typically, all documents in a collection are of similar or related purpose.

Terminology

- A document is a set of key-value pairs.
- MongoDB documents are similar to JSON objects.
  - The values of fields may include scalar data, other documents, arrays, and arrays of documents.
  - Internally, MongoDB stores these documents in the binary format, called BSON (Binary JSON)

RDBMS	NoSQL(MongoDB)
Database	Database
Table	Collection
Tuple/Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	_id provided by MongoDb

mongod

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

View Options on

http://docs.mongodb.org/manual/reference/program/mongod/#bin.mongod

mongos

- mongos for “MongoDB Shard,” is a routing service for MongoDB shard configurations that processes queries from the application layer, and determines the location of this data in the sharded cluster, in order to complete these operations.

http://docs.mongodb.org/manual/reference/program/mongos/

mongo

- 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.

http://docs.mongodb.org/manual/reference/program/mongo/

Sample Document

```
{
    id : 7747,
    name : "Naveen Kumar",
    phones : [ "9900524255", "08041513956"],
    emails : [
    {
        type : "personal",
        address : "adith.naveen@gmail.com"
    },
    {
        type : "official",
        address : "naveenkumar.ks@exilant.com"
    }
    ]
}
```

ObjectId(<hexadecimal>)

- Returns a new ObjectId value. The 12-byte ObjectId value consists of:

- a 4-byte value representing the seconds since the Unix epoch,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

https://docs.mongodb.com/manual/reference/method/ObjectId/

Java Equivalent

```
public class Email{
    private String type;
    private String address;
    // getters & setters
    // constructor
}

public class Person {
    private int id;
    private String name;
    private String[] phones;
    private List<Email> emails = new ArrayList<Email>();
    // constructors
    // getters/setters
}
```

Java Equivalent

```
Person p1 = new Person();
p1.setId(7747);
p1.setName(”NaveenKumar");
p1.setPhones(new String[]{”9900524255",”08041513956"});

p1.getEmails().add(
new Email("personal", ”adith.naveen@gmail.com"));

p1.getEmails().add(
new Email("official", ”naveenkumar.ks@exilant.com"));
```

Advantages

- Schema less
- Documents correspond to native data types in
- many programming languages.
- Structure of a single object is clear
- No complex joins
- Support for embedded data models reduces I/O activity on database system.
- Indexes support faster queries and can include keys from embedded documents and arrays.

High Availability

- MongoDB's replication facility, called replica
- sets, provide:
  - automatic failover.
  - data redundancy.
- A replica set is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability.

Automatic scaling

- MongoDB provides horizontal scalability as part of its core functionality.
  - Automatic sharding distributes data across a cluster of machines.
  - Replica sets can provide eventually-consistent reads for low-latency high throughput deployments.

Two Ways of Scaling

- Vertical Scaling adds more CPU and storage resources to increase capacity.

- Sharding, or horizontal scaling, by contrast, divides the data set and distributes the data over multiple servers, or shards. Each shard is an independent database, and collectively, the shards make up a single logical database.

*Will Discuss later

Install/Setup

Download the binary for your operating system

https://fastdl.mongodb.org/osx/mongodb-osx-x86_64-3.0.2.tgz

Unzip in any directory

/home/user1

Finally export the path (will be discussed in Class)

Default DB location

- MongoDB requires a data directory to store all data.
- MongoDB's default data directory path is \data \db(windows) or /data/db(Unix Variants).
  - You can create this folder structure in the same drive as mongodb's installation drive
  - Or specify another location when starting the server

Starting the server

- The server can be started by running the mongod.exe executable
- Some of the useful options are:

  --dbpath=PATH-TO-YOUR-DB

  --port=27017
- Example:

./mongod –-dbpath=/Users/naveen/mongodb-data

(Note: you must create the folder manually before running this command)

JavaScript Shell

- The executable “mongo.exe” provides an interface to issue direct commands on the db.
- By default the “mongo” command tries to connect to “localhost” and port “27017”
- You can connect to different ones using –host and --port options:
- **`./mongo --port 12345 --host naveen_macbook`**

Lets Start Working

- `show dbs`
  - displays the list of databases
- `use mydb`
  - switches to the database “mydb” if exists, or creates a new with the same name and switches to it
- `db`
  - displays the current database in use
- `db.dropDatabase()`
  - Deletes the current database

Importing external data

- Use the mongoimport.exe tool to import
- external data into a database

```
mongoimport
     --host localhost
     --port 27017
     --db mydb
     --jsonArray
     --collection orders
     --file /Users/naveen/emps.json
```

Exporting data

- Use the mongoexport.exe tool to export

Some commands to start with..

- `db.<collection>.findOne()`
  - Displays the first document in the collection
- `db.<collection>.find()`
  - Displays the first 20 documents in the collection
- `db.<collection>.find().pretty()`
  - displays the first 20 documents in an indented format
- `it` to iterate

More Commands

- `show collections`
  - Displays the list of collections (tables in RDBMS)
- `db.<collection>.insert(data)`
  - If the collection exists, inserts the data, else creates a new collection with the same name and inserts the data

Insert Record

```
p1 = {
   id : 6789,
   name : "John Doe",
   city : "Dallas"
}
db.persons.insert(p1);
```

```
db.persons.insert(
{
   id : 6789,
   name : "John Doe",
   city : "Dallas"
}
);
```

Some More Commands

- Since “mongo” or”mongo.exe” is a JavaScript shell, you can use a script to process the cursor returned by the find() method

```
var d = db.emps.find(); // returns the cursor
// now loop through the cursor to get one
document at a time
while(d.hasNext()){
   var s = d.next();
   print(s.date + " >> " + s.area + " Rs." +
   s.sales);
}
```

Create Collection

- In MongoDB We can explicitly create collection
  - Syntax: `db.createCollection(name, options)`
- Options parameter is optional, so you need to specify only name of the collection. Following is the list of options you can use:

Collection Options

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a collection 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.
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 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.

Example

Creating simple Collection

```
>db.createCollection(“navcollection”)
```

Drop Collection

```
>db.<<CollectionName>>.drop();
```

With options

```
>db.createCollection("mycol", { capped : true,
autoIndexID : true, size : 6142800, max : 10000 } )
```

Knowledge Check

- Insert 5 Persons in the collection called "person",
  - Note: Use one statement

Multiple Insert

```
db.persons.insert([
{
    id : 6789,
    name : "John Doe",
    city : "Dallas"
},
{
    id : 1234,
    name : "Naveen Kumar",
    city : "Bengaluru"
},
{
    id : 2233,
    name : "Kanchan",
    city : "California"
}]);
```

Finding documents

- For query operations, MongoDB provide a db.collection.find() method.
- The method accepts both the query criteria and projections and returns a cursor to the matching documents.
- You can optionally modify the query to impose limits, skips, and sort orders.

Querying

```
db.mycollection.find()
```

Or

```
db.mycollection.find({})
```

The `db.mycollection.findOne()` method also performs a read operation to return a single document.

MongoDB query operation

```
db.users.find(
    { age: 18 },
    { name: 1, address: 1 }
).limit(5)
```

- collection
- query criteria
- projection
- cursor modifier

```
db.users.find(
    { age: { $gt: 18 } },
    { name: 1, address: 1 }
).limit(5)
```

- collection
- query criteria
- projection
- cursor modifier

MongoDB query operation

- All queries in MongoDB address a single collection
- You can modify the query to impose limits, skips, and sort orders
- The order of documents returned by a query is not defined unless you specify a sort()
- MongoDB update/remove methods use the same query syntax

Ordering the query results

- Use the sort() method to achieve it

```
db.orders.find({“customer.customer_id”:"ANTO
N"}).sort({order_date: 1})
```

Querying - Equality

The following example retrieves from the inventory collection all documents where the type field has the value snacks:

```
db.inventory.find( { type: "snacks" } )
```

The following example selects all documents in the inventory collection where the value of the type field is either 'food' or 'snacks':

```
db.inventory.find( { type: {$in:['food',
'snacks' ] } } )
```

AND Conditions

AND conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

```
db.inventory.find( { type: 'food', price: { $lt:
9.95 } } )
```

OR Condition

In the following example, the query document selects all documents in the collection where the field qty has a value greater than ($gt) 100 or the value of the price field is less than ($lt) 9.95:

```
db.inventory.find(
    {
      $or: [ { qty: { $gt: 100 } },
{ price: { $lt: 9.95 } } ]
    }
)
```

AND as well as OR Conditions

selects all documents in the collection where the value of the type field is 'food' and either the qty has a value greater than ($gt) 100 or the value of the price field is less than ($lt) 9.95:

```
db.inventory.find(
   {
     type: 'food',
     $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ]
   }
)
```

Sample Embedded Document

```
db.embed.insert(
{
    _id: 102,
    username:'naveen1',
    contact:{
        phone : ’+919900524255',
        email : 'adith.naveen@gmail.com'
    },
    access : {
        level: 'Manager',
        group : 'Training'
    }
}
);
```

Embedded Documents

- When the field holds an embedded document, a query can either specify an exact match on the embedded document or specify a match by individual fields in the embedded document using the dot notation.

```
db.embed.find({'contact.phone':'1234'})
```

Arrays

- Consider an inventory collection that contains the following documents:

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

Query On Arrays

- The following example queries for all documents where the field ratings is an array that holds exactly three elements, 5, 8, and 9, in this order:

```
db.inventory.find( { ratings: [ 5, 8, 9 ] } )
```

Match an Array Element

```
db.inventory.find( { ratings: 5 } )
```

Query On Arrays

Match a Specific Element of an Array

```
db.inventory.find( { 'ratings.0': 5 } )
```

Specify Multiple Criteria for Array Elements

Use $elemMatch operator to specify multiple criteria on the elements of an array such that at least one array element satisfies all the specified criteria.

The following example queries for documents where the ratings array contains at least one element that is greater than ($gt) 5 and less than ($lt) 9:

```
db.inventory.find( { ratings:{$elemMatch: {$gt: 5, $lt: 9 } } } )
```

Modify Documents

- MongoDB provides the update() method to update the documents of a collection.

The method accepts as its parameters:

- an update conditions document to match the documents to update,
- an update operations document to specify the modification to perform, and
- an options document.

Modify Documents

- By default, update() updates a single document. To update multiple documents, use the multi option.

Modify Syntax

- `db.COLL_NAME.update(SELECTIOIN_CRITERIA, UPDATED_DATA)`

Example

```
{
   "_id" : ObjectId("5549d6a2166b1d451d8328c0"),
   "id" : 6789,
   "name" : "John Doe",
   "city" : "Dallas"
}
{
   "_id" : ObjectId("5549d6a2166b1d451d8328c1"),
   "id" : 1234,
   "name" : "Naveen Kumar",
   "city" : "Bengaluru"
}
```

Update Document emps

```
db.emps.update({'name':'Kumar'},{$set:
{'city':’Ney York'}})
```

By default mongodb will update only single document, to update multiple you need to set a paramter 'multi' to true.

```
db.emps.update({'name':'Kumar'},{$set:
{'city':'Las Vegas'}}, {multi:true})
```

Update multiple documents

```
db.emps.update(
   {'name':'John Doe'},
   {$set:
      {'city':'Mysuru', 'id':12121}
   },
   {multi:true}
)
```

upsert Option

- By default, if no document matches the update query, the update() method does nothing.

- However, by specifying upsert: true, the update() method either updates matching document or documents, or inserts a new document using the update specification if no matching document exists.

```
db.emps.update(
    {'name':'Hemanth'},
    {$set:
        {'city':'Mysuru', 'id':12121}
    },
    {upsert:true}
)
```

Save

Definition

```
db.collection.save()
```

Syntax

```
db.collection.save(
    <document>,
    {
      writeConcern: <document>
    }
)
```

Save

- If the document contains an _id field, then the save() method is equivalent to an update with the upsert option set to true and the query predicate on the _id field.
- In the following example, save() method performs an insert since the document passed to the method does not contain the _id field:

```
db.products.save( { item: "book", qty: 40 } )
```

Save

- In the following example, save() performs an update with upsert:true since the document contains an _id field:

```
db.products.save( { _id: 100, item: "water",
qty: 30 } )
```

Remove Documents

Remove All Documents

```
db.mycollection.remove()
```

or

```
db.mycollection.remove({})
```

Remove Documents that Match a Condition

```
db.inventory.remove( { type : "food" } )
```

Projection

- MongoDB Query Document accepts second optional parameter that is list of fields that you want to retrieve.
- To limit this you need to set list of fields with value 1 or 0. 1 is used to show the filed while 0 is used to hide the field.

```
db.COLL_NAME.find({},{KEY:1})
```

<u>Example</u>

```
db.emps.find({},{”name":1,_id:0})
(This will show name field and hide _id, by default
_id is always shown)
```

Limit

- To limit the records in MongoDB, you need to use limit() method. limit() method accepts one number type argument, which is number of documents that you want to displayed.

Syntax

- `db.COLLECTION_NAME.find().limit(NUMBER)`

Example

Displays only two records

```
db.emps.find({},{”name":1,_id:0}).limit(2)
```

Skip

- Apart from limit() method there is one more method skip() which also accepts number type argument and used to skip number of documents.

```
db.COLLECTION_NAME.find().limit(NUMBER).skip(NUMBER)
```

Following example will only display only second document.

```
db.mycol.find({},{"title":1,_id:0}).limit(1).skip(1)
```

Iterate a Cursor in the mongo Shell

The db.collection.find() method returns a cursor. To access the documents, you need to iterate the cursor.

```
var myCursor = db.inventory.find( { type: 'food' } );

while (myCursor.hasNext()) {
   print(tojson(myCursor.next()));
   //  printjson(myCursor.next());
}
```

Iterate a Cursor in the mongo Shell

- You can use the cursor method forEach() to iterate the cursor and access the documents, as in the following example:

```
var myCursor =  db.inventory.find( { type: 'food' } );

myCursor.forEach(printjson);
```

This Slide Is Intentionally Left Blank

Indexes

- 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.
- Indexes are special data structures, that store a small portion of the data set
- The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in index.

Indexes

Basic syntax of ensureIndex() method is as follows()

```
db.COLLECTION_NAME.ensureIndex({KEY:1})
```

Example (Single)

```
db.mycol.ensureIndex({"title":1})
```

Example (Multiple)

```
db.mycol.ensureIndex({"title":1,”job":-1})
```

Indexes

Note:

1. ensureIndex is depricated from MongoDB 3.x Version instead use createIndex
2. a value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order

Example (Single)

```
db.mycol.createIndex({userid:1})
```

Example (Mutiple)

```
db.mycol.createIndex({userid:1, job:-1})
```

Indexes

The created index will support queries that select on the field userid, such as the following:

```
db.records.find( { userid: 2 } )
db.records.find( { userid: { $gt: 10 } } )
```

The created index will not support

```
db.records.find( { profile_url: 2 } )
```

Compound Index

To create a compound index use an operation that resembles the following prototype:

```
db.products.createIndex( { item: 1, category: 1,
price:1})
```

Background Construction Of Index

If your collection holds a large amount of data, and your application needs to be able to access the data while building the index, consider building the index in the background

```
db.people.createIndex( { zipcode: 1}, {background:
true} )
```

Note : By default background is false

Create a Unique Index

- MongoDB allows you to specify a unique constraint on an index. These constraints prevent applications from inserting documents that have duplicate values for the inserted fields.

- MongoDB cannot create a unique index on the specified index field(s) if the collection already contains data that would violate the unique constraint for the index.

Creating Unique Index

To create a unique index, consider the following prototype:

```
db.collection.createIndex( { a: 1 }, { unique:
true } )
```

<u>Example:</u>

```
db.accounts.createIndex( { "tax-id": 1 },
{ unique: true } )
```

Unique Compound Index

You can also enforce a unique constraint on compound indexes, as in the following prototype:

```
db.collection.createIndex( { a: 1, b: 1 }, { unique:
true } )
```

Sparse Indexes

- Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value. The index skips over any document that is missing the indexed field.
- The index is “sparse” because it does not include all documents of a collection.

Example

```
db.addresses.createIndex( { ”age": 1 }, { sparse: true } )
```

NOTE:

Do not confuse sparse indexes in MongoDB with block-level indexes in other databases. Think of them as dense indexes with a specific filter.

Working on Sparse Indexes

Consider a collection scores that contains the following documents:

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : "newbie" }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
```

The collection has a sparse index on the field score:

```
db.scores.createIndex( { score: 1 } , { sparse: true } )
```

Working on Sparse Indexes

```
db.scores.find( { score: { $lt: 90 } } )
```

Because the document for the userid "newbie" does not contain the score field and thus does not meet the query criteria, the query can use the sparse index to return single record

```
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby",
"score" : 82 }
```

Working on Sparse Indexes

Consider the following query to return all documents in the scores collection, sorted by the score field:

db.scores.find().sort({ score: -1 })

Note: The above query will return all the records though it sparse index

To use the sparse index, explicitly specify the index with hint():

```
db.scores.find().sort( { score: -1 } ).hint( { score: 1 } )
```

Geospatial Indexes

Geospatial Indexes

Create / Import the data which has 2d data like location with x,y cords.

Schema Look

```
db.population.find().limit(1).pretty()
```

Create Index

```
db.population.createIndex({loc:'2d', city:1});
```

Note: GeoSpatial Indexes to be first in the list.

Contd…

Query to get all location with distance close to 72, 40

```
db.population.find({loc:{$near:[72,40]}}).limit(20)
```

Geosphere Indexes

Fields with 2dsphere indexes must hold geometry data in the form of coordinate pairs or GeoJSON data.

Sample Data

```
db.places.insert({name:'Office1', 'city':'Bengaluru', 'location':{"type":"Point", "coordinates":[50.12345, 37.44322] }});
db.places.insert({name:'Office2', 'city':'Bengaluru', 'location':{"type":"Point", "coordinates":[51.125, 37.2] }});
db.places.insert({name:'Office3', 'city':'Bengaluru', 'location':{"type":"Point", "coordinates":[52.1345, 44.412] }});
db.places.insert({name:'Office4', 'city':'Bengaluru', 'location':{"type":"Point", "coordinates":[55.1, 56.4122] }});
db.places.insert({name:'Office5', 'city':'Bengaluru', 'location':{"type":"Point", "coordinates":[67.0, 75.44322] }});
db.places.insert({name:'Office6', 'city':'Bengaluru', 'location':{"type":"Point", "coordinates":[-55.5, 55.44322] }});
db.places.insert({name:'Office7', 'city':'Bengaluru', 'location':{"type":"Point", "coordinates":[88.145, 36.42122] }});
db.places.insert({name:'Office8', 'city':'Bengaluru', 'location':{"type":"Point", "coordinates":[77.145, 77.1322] }});
db.places.insert({name:'Office9', 'city':'Bengaluru', 'location':{"type":"Point", "coordinates":[56.123, 88.44322] }});
```

Create 2dsphere Index

```
db.places.createIndex({'location':'2d
sphere'})
```

Check Created Index

```
db.places.getIndexes()
```

Query – 2dsphere

```
db.places.find({
   location:{
      $near: {
         $geometry: {
            type:'Point',
            coordinates:[50.66,50.443]
         },
         $maxDistance:2000000
      }
   }
}).pretty()
```

Note: 2000000 Number of mts
50.66 -> latitude
50.443 -> longitude

Text Indexes

Starting in MongoDB 3.2, MongoDB introduces a version 3 of the text index. Key features of the new version of the index are:

- Improved case insensitivity
- Additional delimiters for tokenization

Text Indexes

Sample Data:

```
db.sentence.insert({"words":"dog"});
db.sentence.insert({"words":"dog cat"});
db.sentence.insert({"words":"cat dog"});
db.sentence.insert({"words":"dog tree cat
running"});
db.sentence.insert({"words":"its always good to have
a dog"});
db.sentence.insert({"words":"dogs and cats play "});
db.sentence.insert({"words":"cat kills rat"});
db.sentence.insert({"words":"cat is cute"});
db.sentence.insert({"words":"dog snouser"});
```

Link: https://docs.mongodb.org/manual/core/index-text/

Text Indexes - Create

```
db.sentence.find({words:"dogs"})
```

The above query will not return any record

Create Index

```
db.sentence.createIndex({'words':'text'})
```

Once the Index is created we can do partial search and can get ranking

```
db.sentence.find({$text:{$search:'snouser.'}})
```

Text Indexes

- Lets get the ranking of the search

```
db.sentence.find({
   $text:{
      $search:'hello cat & dog'
   }
}, {score:{
         $meta:'textScore'
   }
}).sort({score:{$meta:'textScore'}})
```

Remove a Specific Index

Syntax

To remove an index, use the `db.collection.dropIndex()` method.

Example

```
db.accounts.dropIndex( { "tax-id": 1 } )
```

Modify Index

To modify an existing index, you need to drop and recreate the index.

Rebuild Indexes

If you need to rebuild indexes for a collection you can use the `db.collection.reIndex()` method to rebuild all indexes on a collection in a single operation. This operation drops all indexes, including the `_id index`, and then rebuilds all indexes.

Example

```
db.accounts.reIndex()
```

How Big is Your Index?

db.emp.stats() -- which gives the details

<u>To get Total Index Size</u>

Db.emp.totalIndexSize()

Use WiredTiger to Reduce Index Size

- You Can start **mongod** server with wiredtiger to save index space, as shown below:

```
mongod --dbpath ./data1/ --
storageEngine wiredTiger --
wiredTigerIndexPrefixCompression true
```

This Slide Is Intentionally Left Blank

Analyze Query Performance

The `cursor.explain("executionStats")` and the `db.collection.explain("executionStats")` methods provide statistics about the performance of a query. This data output can be useful in measuring if and how a query uses an index.

Evaluate Performance of Query

Consider a collection inventory with the following documents:

```
{ "_id" : 1, "item" : "f1", type: "food", quantity: 500 }
{ "_id" : 2, "item" : "f2", type: "food", quantity: 100 }
{ "_id" : 3, "item" : "p1", type: "paper", quantity: 200 }
{ "_id" : 4, "item" : "p2", type: "paper", quantity: 150 }
{ "_id" : 5, "item" : "f3", type: "food", quantity: 300 }
{ "_id" : 6, "item" : "t1", type: "toys", quantity: 500 }
{ "_id" : 7, "item" : "a1", type: "apparel", quantity: 250 }
{ "_id" : 8, "item" : "a2", type: "apparel", quantity: 400 }
{ "_id" : 9, "item" : "t2", type: "toys", quantity: 50 }
{ "_id" : 10, "item" : "f4", type: "food", quantity: 75 }
```

Query With No Index

```
db.inventory.find(
   { quantity: { $gte: 100, $lte: 200 } }
).explain("executionStats")
```

Look at the output on console

Query with Index

- Create a index on field quantity

```
db.inventory.createIndex( { quantity: 1 } )
```

Query After Creating Index

```
db.inventory.find(
   { quantity: { $gte: 100, $lte: 200 } }
).explain("executionStats")
```

Look at the output on console

Compare Performance of Indexes

- To manually compare the performance of a query using more than one index, you can use the hint() method in conjunction with the explain() method.

Consider the following query:

```
db.inv.find( { quantity: { $gte: 100, $lte: 300 }, type:
"food" } )
```

Compound Index

- For example, add the following two compound indexes. The first index orders by quantity field first, and then the type field. The second index orders by type first, and then the quantity field.

```
db.inv.createIndex( { quantity: 1, type: 1 } )
db.inv.createIndex( { type: 1, quantity: 1 } )
```

Evaluate Effect Of Both

With Both hints

```
db.inventory.find(
   { quantity: { $gte: 100, $lte: 300 }, type: "food" }
).hint({ quantity: 1, type: 1 }).explain("executionStats")
```

```
db.inventory.find(
   { quantity: { $gte: 100, $lte: 300 }, type: "food" }
).hint({ type: 1, quantity: 1 }).explain("executionStats")
```

List All Indexes

To get all indexes of the collection

```
db.people.getIndexes()
```

Get all the indexes of the DB

```
db.getCollectionNames().forEach(function(collection) {
   indexes = db[collection].getIndexes();
   print("Indexes for " + collection + ":");
   printjson(indexes);
});
```

Aggregation

- Aggregations are operations that process data records and return computed results. MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets.
- Like queries, aggregation operations in MongoDB use collections of documents as an input and return results in the form of one or more documents.

First Look @ Aggregate

Collection

```
db.orders.aggregate( [
    $match stage ——→  { $match: { status: "A" } },
    $group stage ——→  { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
] )
```

Map-Reduce

- MongoDB also provides map-reduce operations to perform aggregation.
- It has two phases:
  - a map stage that processes each document and emits one or more objects for each input document
  - reduce phase that combines the output of the map operation.
- Pros
  - Like other aggregation operations, map-reduce can specify a query condition to select the input documents as well as sort and limit the results.
- Cons
  - map-reduce is less efficient and more complex than the aggregation pipeline.

Look @ Map Reduce

```
db.orders.mapReduce(
        map      →  function() { emit( this.cust_id, this.amount ); },
        reduce   →  function(key, values) { return Array.sum( values ) },
                    {
        query    →    query: { status: "A" },
        output   →    out: "order_totals"
                    }
                )
```

Collection

Working With Aggregation

Load the data from

http://media.mongodb.org/zips.json?_ga=1.128895436.1944068379.1430128247

Data Model

Each document in the zipcodes collection has the following form:

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

Return States with Populations above 10 Million

The following aggregation operation returns all states with total population greater than 10 million:

<u>The SQL for this aggregation operation is:</u>

```
SELECT state, SUM(pop) AS totalPop
FROM zipcodes
GROUP BY state
HAVING totalPop >= (10*1000*1000)
```

Return States with Populations above 10 Million

```
db.zipcodes.aggregate( [
   { $group: { _id: "$state", totalPop: { $sum:
"$pop" } } },
   { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

Return Average City Population by State

The following aggregation operation returns the average populations for cities in each state:

```
db.zipcodes.aggregate( [
   { $group: { _id: { state: "$state", city: "$city" },
pop: { $sum: "$pop" } } },
   { $group: { _id: "$_id.state", avgCityPop: { $avg:
"$pop" } } }
] )
```

- The first $group stage groups the documents by the combination of city and state, uses the $sum expression to calculate the population for each combination, and outputs a document for each city and state combination.
- A second $group stage groups the documents in the pipeline by the _id.state field (i.e. the state field inside the _id document), uses the $avg expression to calculate the average city population (avgCityPop) for each state, and outputs a document for each state.

The following aggregation operation returns the smallest and largest cities by population for each state:

```
db.zipcodes.aggregate( [
   { $group:
      {
        _id: { state: "$state", city: "$city" },
        pop: { $sum: "$pop" }
      }
   },
   { $sort: { pop: 1 } },
   { $group:
      {
        _id : "$_id.state",
        biggestCity:  { $last: "$_id.city" },
        biggestPop:   { $last: "$pop" },
        smallestCity: { $first: "$_id.city" },
        smallestPop:  { $first: "$pop" }
      }
   }]);
```

MapReduce

Syntax:

```
db.collection.mapReduce(
   mapFunction,
   reduceFunction,
   options);
```

Using mapReduce

mapFunction

- a callback function
- has access to a single document via “this”
- should emit two properties from “this” or values derived out of “this” properties

Using mapReduce

emit(a, b)

- “a” will be used as a key representing an array of “b” values

Example:

- emit(this.category, this.sales) will create a dictionary with “category” as key and each “category” representing an array of corresponding “sales” values .

Example

Consider Garments of following

```
{ "color" : "Green", "value" : 4 }
{ "color" : "Green", "value" : 1 }
{ "color" : "Green", "value" : 4 }
{ "color" : "Blue", "value" : 4 }
{ "color" : "Yellow", "value" : 39 }
{  "color" : "Blue", "value" : 3 }
```

- emit(this.color, this.value)

would create a collection like this

```
[
{key: ”Green", values: [ 4,1,4] },
{key: ”Blue", values: [ 4,3 ] }
{key: ”Yellow", values: [ 39 ] }
]
```

Example

- The reduceFunction will receive the key and values from each document separately

Using mapReduce

reduceFunction

- – callback function
- – called for each of the key generated by the emit function
- – receives two arguments, key and an array of values

Working Example

```
map1=function(){
   emit(this.color, this.value);
}

reduce1=function(key,vals){
   for(var i=0,sum=0; i<vals.length; i++){
      sum += vals[i];
   }
   return sum;
}
```

Working Example

Here something is called named collection which is user defined

```
var res1 = db.garments.mapReduce(map1,
reduce1, {out:"something"});
db[res1.result].find();
```

SECURITY

Security

Maintaining a secure MongoDB deployment requires administrators to implement controls to ensure that users and applications have access to only the data that they require.

Enable Client Access Control

To enable access control, use command line option --auth

With access control enabled, ensure you have a user with `userAdmin` or `userAdminAnyDatabase` role in the admin database.

Steps To Configure - Security

1) Start MongoDB without access control.

```
mongod --port 27017 --dbpath /data/db1
```

2) Connect to the instance.

```
mongo --port 27017
```

3) Create the user administrator.

```
use admin
db.createUser(
  {
    user: “myUserAdmin”,
    pwd: "abc123",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  }
)
```

Steps To Configure - Security

4) Re-start the MongoDB instance with access control.

```
mongod --auth --port 27017 --dbpath /data/db1
```

5) Authenticate as the user administrator.

```
mongo --port 27017 -u "myUserAdmin" -p "abc123" --
authenticationDatabase "admin"
```

Or, in the mongo shell connected without authentication

```
use admin
db.auth("myUserAdmin", "abc123" )
```

Disable Security

- If you need to disable access control for any reason, restart the MongoDB instance without the --auth command line option

Manage User & Roles

- https://docs.mongodb.org/v3.0/core/security-built-in-roles/#database-user-roles

Create Simple Custom Role

The role document has the following fields:

Field	Type	Description
Role	String	The name of the new role.
Privileges	array	The privileges to grant the role. A privilege consists of a resource and permitted actions. For the syntax of a privilege, see the privileges array. You must include the privileges field. Use an empty array to specify no privileges.
Roles	Array	An array of roles from which this role inherits privileges. You must include the roles field. Use an empty array to specify no roles to inherit from.

Create Simple Custom Role

```
db.createRole({
    role:"testRoleRead",
    privileges:[
            {
                resource: { db: "navdb", collection: "" },
                    actions: [ "find", "update", "insert", "remove" ]
              },
            ],
    roles:[
        {role:"read", db:"inventory"},
        {role:"readWrite", db:"accounts"}
    ]
});
```

To Remove Role

```
db.dropRole("testRoleRead");
```

Some Live Examples – Role(Custom)

```
use admin
db.createRole(
   {
     role: "myClusterwideAdmin",
     privileges: [
       { resource: { cluster: true }, actions: [ "addShard" ] },
       { resource: { db: "config", collection: "" }, actions: [ "find",
"update", "insert", "remove" ] },
       { resource: { db: "users", collection: "usersCollection" }, actions:
[ "update", "insert", "remove" ] },
       { resource: { db: "", collection: "" }, actions: [ "find" ] }
     ],
     roles: [
       { role: "read", db: "admin" }
     ]
   },
   { w: "majority" , wtimeout: 5000 }
)
```

Some Live Examples – Role(Custom)

```
use reporting
db.createUser(
    {
      user: "reportsUser",
      pwd: "12345678",
      roles: [
         { role: "read", db: "reporting" },
         { role: "read", db: "products" },
         { role: "read", db: "sales" },
         { role: "readWrite", db: "accounts" }
      ]
    }
)
```

To Know User Roles

Syntax

```
db.getUser(“<<userName>>”)
```

Example

```
db.getUser("naveen")
```

Note: Make sure have mongod started with --auth

To know Role Details

Example:

```
db.getRole("userAdminAnyDatabase",
   {showPrivileges:true}
)
```

Grant Roles to Existing User

```
use reporting
db.grantRolesToUser(
  "reportsUser",
  [
    { role: "readWrite", db: "products" } ,
    { role: "readAnyDatabase", db:"admin" }
  ]
)
```

This Slide Is Intentionally Left Blank

This Slide Is Intentionally Left Blank

Java With MongoDB

```
MongoClient mc = new MongoClient();

System.out.println("Got Connection " +
mc.getConnectPoint());

System.out.println("list all DBS in Server");

for(String dbs : mc.getDatabaseNames()){
    System.out.println("DB Name is " + dbs);

    System.out.println("Collections in " + dbs );

    DB db = mc.getDB(dbs);
    for(String coll : db.getCollectionNames()){
        System.out.println("\t\t" + coll);
    }
```

Get One Record From DB

```
DBObject myojb = garments.findOne();

System.out.println("Color is " + myojb.get("color"));
System.out.println("Sales is " + myojb.get("sales"));
```

Get All Documents

```
DBCursor cur = garments.find();

while(cur.hasNext()){
    DBObject temp = cur.next();

System.out.println("Color : " + temp.get("color")
        +", Sales :" + temp.get("sales"));

}
```

Get Queried Data

```
DBCursor dbc = garments.find(
        new BasicDBObject("color", "Green")
        ).sort(new BasicDBObject("sales", 1)).limit(2);

while(dbc.hasNext()){
    DBObject temp  = dbc.next();

        System.out.println(temp.get("color") + ", " +
temp.get("sales"));
}

System.out.println("_____________________________________");
        System.out.println(JSON.serialize(dbc));
```

Save Using BasicDBObject

```
//  first way using BasicDBObject

    DBObject obj = new BasicDBObject();
    obj.put("empid", 300);
    obj.put("empname", "Krishna");
    obj.put("Desig", "Software Engineer");
    obj.put("salary", 1800);

    emp.save(obj);
    System.out.println("Record Saved Successfully....");
```

Save with Map

```
Map<String, Object> map = new HashMap<String, Object>();

map.put("empid", 301);
map.put("empname", "Chitra");
map.put("desig", "System Analyst");
map.put("Salary", 1900);

emp.save(new BasicDBObject(map));
System.out.println("Record Saved Successfully.... ");
```

Save with String

```
String jsonString =
        "{empid:302, empname:'Sowjanya', "
        + "desig:\"Team Leader\", salary:2000}";

emp.save((DBObject)JSON.parse(jsonString));

System.out.println("Record Saved Successfully.... ");
```

Store Images to MongoDB

```
// first param : db , second param is optional if you give
    // it will be named collection

    GridFS gfs = new GridFS(db, "myimages");
    String path="./images”;
    File folderpath = new File(path);
    for(File file : folderpath.listFiles()){
        InputStream is = new FileInputStream(file);
        GridFSInputFile picture = gfs.createFile(is, file.getName());

        picture.setMetaData(new BasicDBObject("description",
                "images By Naveen"));
        picture.save();
    }
    mc.close();
    System.out.println("Pictures Saved to DB... ");
```

Refer : http://api.mongodb.org/java/3.0/ for documentation

Get Images From MongoDB

```
MongoClient mc = new MongoClient();
DB db = mc.getDB(”exipic");

GridFS gfs = new GridFS(db, "myimages");

String path ="./gotimages/";

List<GridFSDBFile> list = gfs.find(new BasicDBObject());

for(GridFSDBFile file : list){
    file.writeTo(path +file.getFilename());
}

mc.close();

System.out.println("images retrieved... ");
```

LET'S TRY WITH REPLICATION

Document

LET'S TRY WITH SHARDING

Document

LET'S TRY REPLICATION & SHARDING

Windows Based OS

Document

DB Values

Document

Unix Based OS

Document

Thank you!