NOSQL DATABASE

MONGODB DAY 3

Lecture 3 Agenda

- **Objective** :
  - Database **Performance Techniques**: Using **Indexes**
  - **Delete** Operations
  - **Drop** Database
  - Handle **Dynamic Data** Using **Variables** and **For-Each Loops**
  - **Backup** and **Restore** Databases
  - Validate **Data Using Schema Validation**

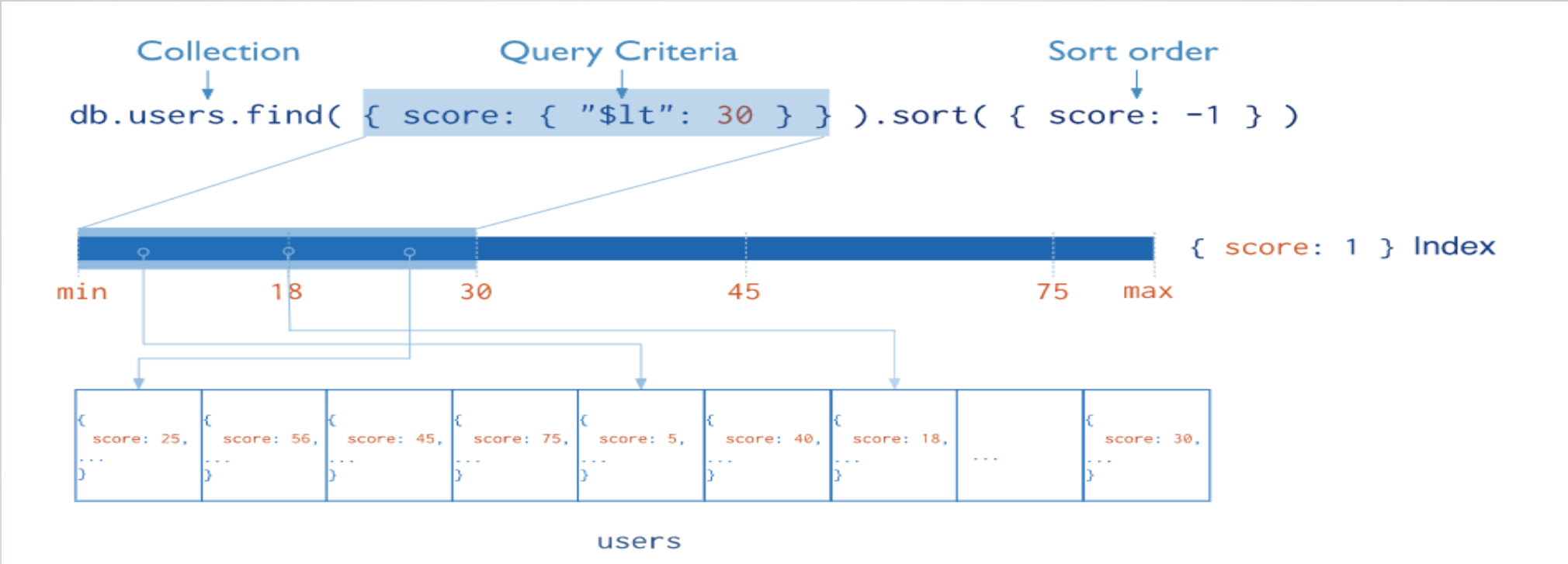
Lecture 3 Agenda

- Mongo Indexes
- Delete Documents
- Drop Database
- Using Variable , forEach
- Backup & Restore
- Schema Validation

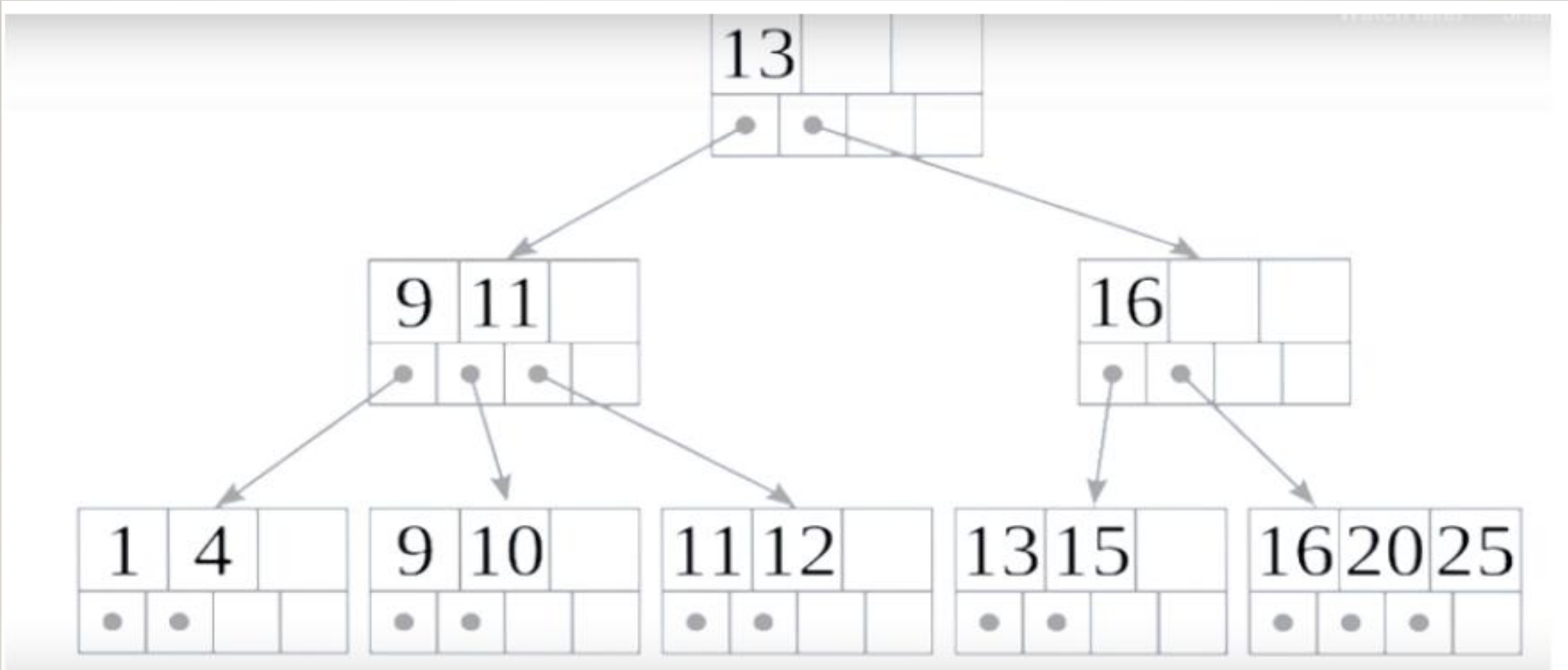
MongoDB Index

- Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a ***collection scan***, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the ***index*** to ***limit the number of documents it must inspect.***
- Indexes are special **data structures**. that store a small portion of the collection's 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. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.
- Although indexes **improve query performance**, **adding an index has negative performance impact for write operations**. For collections with a high write-to-read ratio, indexes are expensive because ***each insert must also update any indexes.***
- The following diagram illustrates a query that selects and orders the matching documents using an index:

MongoDB Index



MongoDB Index



Index Creation

- Winning Plan Stage :
  - Collection Scan
  - Index Scan
  - db.products.find({item:"laptop"}).explain()
  - db.products.explain().find({item:"laptop"})

Index Creation

- **To create Index : Simple Field**
- db.products.createIndex(

  { item: 1} ,

  { name: "query for inventory" })
- **To create Index : Compound Field**
- db.products.createIndex(

  { item: 1, quantity: 1 } ,

  { name: "query for inventory" })

Index Types

- Single Field
- Compound Index
- To find Collection Index
  - db.collection.getIndexes()
- To Drop Index use:

db.collection.dropIndex("index Name")

db.runCommand({dbstats:1})

https://www.mongodb.com/docs/manual/indexes/

Compound Index- Cases

- db.employee.insertMany([{_id:1 , fName:"malak",lName:"mohamed"},{_id:2,fName:"mazen",lName:"mohamed"}])
- db.employee.createIndex({**fName:1,lName:1**},{name:"IX_Employee_Name"})
- db.employee.find({**fName**:"malak",**lName**:"mohamed"}).explain() // **IXSCAN**
- db.employee.find({**lName**:"mohamed",**fName**:"malak"}).explain() // **IXSCAN**

Compound Index- Cases

- db.employee.find({**fName**:"malak"}).explain() // **IXSCAN**

- db.employee.find({**lName**:"mohamed"}).explain() // **COLLSCAN**
- The index is ordered by **fName first**, then by **lName**.
- This means the index is sorted ***primarily*** by fName, and within the same fName, it is **further** sorted by lName.
- because they ***do not match the prefix of the compound index.***

TTL Index (Time-to-Live)

- TTL indexes are special **single-field** indexes that MongoDB can use to ***automatically*** remove documents from a collection after a **certain amount of time.**

```
db.eventlog.createIndex(
  { "lastModifiedDate": 1 },
  { expireAfterSeconds: 3600 })
```

- **Important notes:**
- **Field type requirement**
  - The field (lastModifiedDate) **must be of type Date or an array of Date values**.
  - If it's a string or number, TTL will not work.
- **TTL index limitation**
  - You can only use a **single field** in a TTL index (not compound).
- **Expiration process**
  - MongoDB's background task checks TTL indexes **once every 60 seconds**.
  - So documents may live up to ~60 seconds longer than expireAfterSeconds.

Delete Document

- db.collection.deleteOne({})
- db.collection.deleteMany({})
- db.collection.deleteMany({}) //Delete All Documents , Collection exists

- **Different between :**
  - db.collection.deleteOne({})
  - db.collection. findOneAndDelete({})

Drop Collection ,Drop Database

- db.collection.drop({}) ////Delete All Documents , Collection NOT exists
- db.**dropDatabase**({})

Using Variable

- var myNames=[

{name:"ahmed",age:10},

{name:"ali",age:20},

{name:"eman",age:15}

]

- db.inventory.insertOne(myNames)
- db.inventory.insertMany(myNames)

Using Variable

```
var data = {};
print(typeof data) //object

if (Array.isArray(data))
print("data is Array")
else if (typeof data === 'object')
print("data is object")
```

```
data = [ ]
print(typeof data) // [ ] Array

data = "ahemd"
print(typeof data) // string

data = 123
print(typeof data) // number
```

Using forEach

- db.orders.find({}).forEach(function (n) {print("name is :" + n.name)})

Default Database creation path

- C:\Program Files\MongoDB\Server\Version\bin\mongod/.cfg

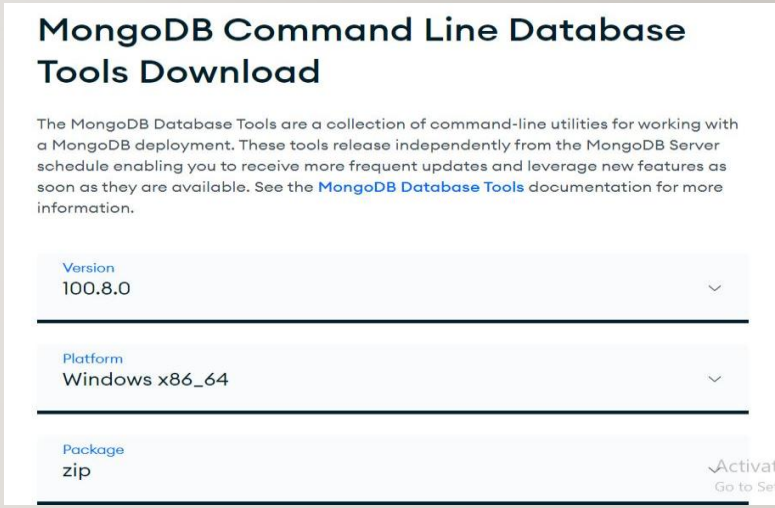
storage:

dbPath: C:\Program Files\MongoDB\Server\Version\data

db.serverCmdLineOpts()

Mongo Backup & Restore Full DB

- First make sure from
  - Installed **MongoDB Database tools**
  - bin folder **mongodump.exe , mongorestore.exe**
  - Make sure from **Environment Path**
- **Open Windows CMD**



mongodump --db databaseName --out "to Save backup File path"

mongodump --db DentalCenter --out "F:\Mohamed\NO_SQL_MongoDB\DB"

Mongo Backup & Restore Full DB

- Open Windows CMD

mongorestore --db ***databaseNameYouNeed(NEW)*** --dir " Saved backup File path“

mongorestore --db DentalCenter --dir "F:\Mohamed\NO_SQL_MongoDB\DB\DentalCenter"

Mongo Backup & Restore Collection

- **Open Windows CMD**

mongodump --db databaseName –collection “collectionName” --out “to Save backup File path“

mongodump --db DentalCenter –collection "clinic" --out "F:\Mohamed\NO_SQL_MongoDB\DB"

Mongo Backup & Restore Collection

- Open Windows CMD

mongorestore --db ***databaseNameYouNeed(NEW)*** --dir " Saved backup File path“

mongorestore --db DentalCenter --dir "F:\Mohamed\NO_SQL_MongoDB\DB\DentalCenter\clinic.bson"

Schema Validation:
lets you create **validation rules** for your fields, such as allowed **data types and value ranges**.

```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         title: "Student Required Input",
         required: [ "name", "age", "code" ],
      }}})
```

Schema Validation:

```
db.createCollection("students", {
   validator: {
      $jsonSchema: {
         bsonType: "object",
         title: "Student Object Validation",
         required: [ "address", "major",
"name", "year" ],
         properties: {
            name: {
               bsonType: "string",
               description: "'name' must be a string and is required"
            },
            year: {
               bsonType: "int",
               minimum: 2017,
               maximum: 3017,
               description: "'year' must be an integer in [ 2017, 3017 ] and is required"
            },
            gpa: {
               bsonType: [ "double" , "int" ],
               description: "'gpa' must be a double if the field exists"
            }
         }
      }
   }
} )
```

Schema Validation:
Modify existing Collection

```
db.runCommand( {
   collMod: "students",
   validator: { $jsonSchema: {
      bsonType: "object",
      required: [ "username", "password" ],
      properties: {
         username: {    bsonType: "string",
            description: "username must be a string and is required"},
         password: {            bsonType: "string",              minLength: 6,
               description: "must be a string of at least 6 characters, and is required"        }     }  } }  } )
```

Schema Validation:
remove existing validation

```
db.runCommand({
  collMod: "students",
  validator: {}
})
```

Schema Validation:

Enforce the **same schema** consistently across **all related collections at once.**

```
var schema = {
  $jsonSchema: {
    bsonType: "object",
    required: ["name", "email"],
    properties: {
      name: { bsonType: "string" },
      email: { bsonType: "string" }
    }
  }
};
```

```
["employees", "staff", "students"].forEach(col => {
  db.runCommand({
    collMod: col,
    validator: schema
  });
});
```

Schema Validation:
Conditional Schema using: oneOf

```
db.runCommand({
collMod: "employees",
validator: {
"$jsonSchema": {
"bsonType": "object",
"required": ["status"],
"properties": {
"status": {
"bsonType": "string",
"enum": ["Active", "Inactive"]},
"department": {
"bsonType": "string"
}},
"oneOf": [
{"properties": {
"status": { "enum": ["Active"] }
},"required": ["department"]
},{
"properties": {
"status": { "enum": ["Inactive"] } } } ] } }});
```

Schema Validation:
Block all Fields Using additionalProperties :false

```
db.createCollection("staff", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "age"],
      properties: {
        _id:{},          // allow any type for _id
        name: { bsonType: "string" },
        age: { bsonType: "int" }
      },     additionalProperties: false    }  }})
```

Schema Validation:
Block Specific Fields Using Reject Specific fields

```
db.createCollection("people", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      not: {
        required: ["address"]    // blockedField
      },
   properties: { //Optional, but if provided, value must follow the specified data type
        name: { bsonType: "string" },
        age: { bsonType: "int" }      }    }  }})
```

FORBIDDEN FIELDS COMPARISON (CODE1,2)

Feature / Behavior	Code 1: Block All Extra Fields(additionalProperties: false)	Code 2: Block Specific Fields(not: { required: ["field"] })
Purpose	Forbid any fields not explicitly listed	Forbid **specific** field(s) from being present
Field control	You must list all allowed fields in properties	Only list the blocked fields using not.required
Default _id issue	Must explicitly allow _id in properties	No issue with _id, since it's not forbidden
New field addition (future-proofing)	Requires schema update to allow new fields	Automatically allows new fields unless forbidden
Example use case	Sensitive documents with a fixed structure	Want to prevent specific keys like password, admin, debug
Syntax complexity	Medium (write all allowed fields)	Low (just a not.required clause)

Schema Validation: examples

Some examples - Schema Validation

THANK YOU

Any Question