Mongo DB

Shristi Technology Labs

Contents

NoSQL

- NoSQL is used to refer to any data store that does not follow the traditional RDBMS model
- The data is non-relational
- It does not use SQL as the query language

Why NoSQL

- Offers rich query language
- Easy scalability.
- No Schema required
- Non relational
- Highly Distributable
- High performance with high availability

Why NoSQL

Schema agnostic

- A database schema is the description of all possible data and data structures in a relational database.

- ***No schema required, giving the freedom to store information without doing up-front schema design.***

Non relational

- Relations in a database establish connections between tables of data. (eg). a list of order details can be connected to a separate list of delivery details.

- ***With a NoSQL database, this information is stored as an aggregate — a single record for order with everything, including the delivery address.***

Why NoSQL

Commodity hardware

- Some databases are designed to operate best (or only) with specialized storage and processing hardware.

- ***NoSQL database can use cheap off-the-shelf servers .***
- ***These servers allow NoSQL databases to scale to handle more data.***

Highly distributable

- Distributed databases can store and process a set of information on more than one device.

- ***In NoSQL database, a cluster of servers can be used to hold a single large database.***

Type of NoSQL databases

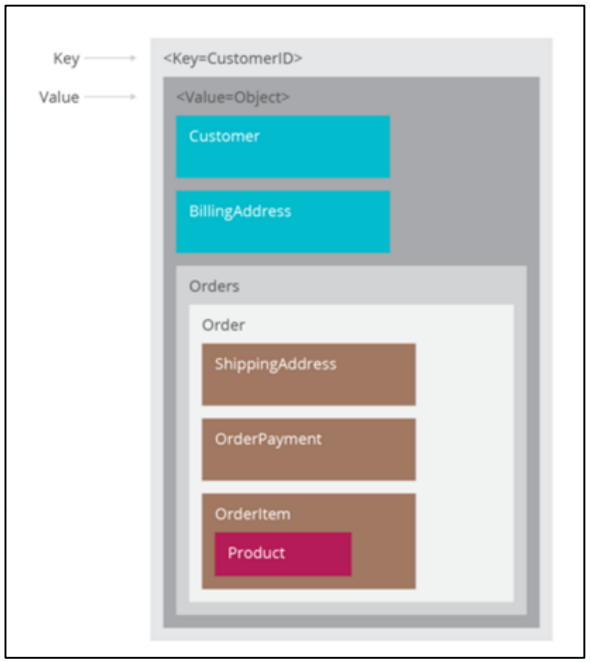
- There are various storage types available in which the content can be modelled for
  - Column-oriented
  - Document Store
  - Key Value Store
  - Graph

Key Value Storage

- A **Key-Value store allows the storage** of a value against a key
- The key must be known to retrieve the value
- **Avoid** using this if there is need to query by data, or to operate on multiple keys at the same time

Examples

- *Memcached, Redis, Riak, VoltDB*



Column Oriented Databases

- The column-oriented databases ***store data as columns***
- Used to store and process very large amounts of data distributed over many machines.
- They are well-suited for use with applications that
  - Require the ability to always write to the database
  - Geographically distributed over multiple data centers
  - Can tolerate some short-term inconsistency in replicas
  - Have the potential for truly large volumes of data, such as hundreds of terabytes

- **Avoid** using this if the systems are in early development, or having changing query patterns

Examples

- *Google's BigTable, HBase and Cassandra*

Document-oriented database

- A document store stores data as documents
- The documents act as records (or rows),
- This allows the inserting, retrieving, and manipulating of semi-structured data.
- Databases use XML, JSON with data access typically over HTTP protocol

- **Avoid** using document databases for systems that need complex transactions spanning multiple operations or queries against varying aggregate structures

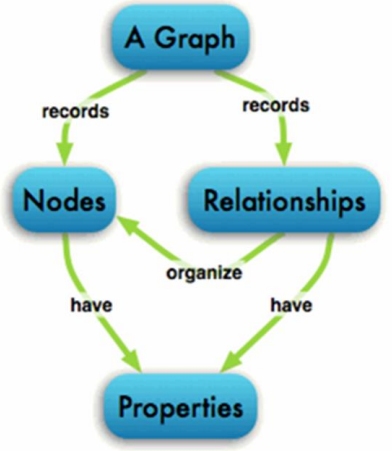
Examples

- *Couchbase, MongoDB, Lotus Notes*

```
{
  "EmployeeID": "SM1",
  "FirstName" : "Ritu",
  "LastName"  : "Kumar",
  "Age"       : 22,
  "Salary"    : 10000
}
```

Graph Databases

- Entities are also known as nodes, which have properties
- Relationships are represented as graphs.
- There can be multiple links between two nodes in a graph representing the multiple relationships that the two nodes share.



- The relationships represented may include social relationships between people, transport links between places, or network topologies between connected systems

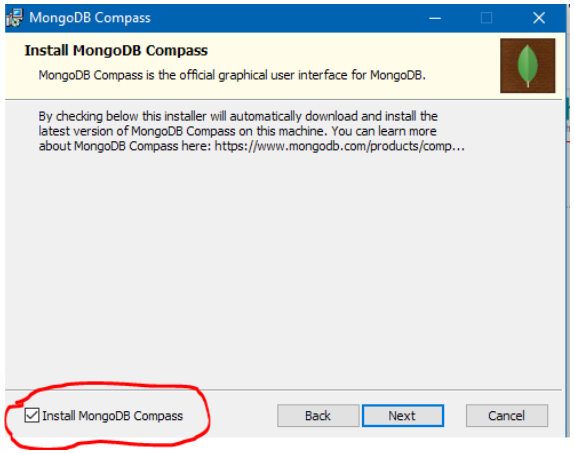
Example

- *InifiniteGraph, Neo4J, OrientDB*.

.

Installation

- Download MongoDB Community Edition from the given link https://www.mongodb.com/download-center#community



- Once downloaded install mongo
  - Select the complete setup
  - Check Mongo Compass selection
  - Click Install to start the installation
- MongoDB requires a data directory to store all data.
- The default data directory path is the absolute path **\data\db** on the drive from which MongoDB is started
- Create a folder **data/db** inside C drive

Start MongoDB

- Set the path for environment variable for mongoDB in the path
- Start the server from the command prompt by the command **mongod.exe**

```
C:\Users\SPRIYA MATHAN>mongod.exe
2018-02-24T05:58:17.201-0700 I CONTROL  [initandlisten] MongoDB starting
 : pid=10344 port=27017 dbpath=C:\data\db\ 64-bit host=SPRIYAMATHAN
```

- Or the MongoDB server can be started by running the command from the installation directory/bin of MongoDB

```
C:\Program Files\MongoDB\Server\3.6\bin>mongod.exe
2018-02-24T06:11:37.358-0700 I CONTROL  [initandlisten] MongoDB
 starting : pid=32700 port=27017 dbpath=C:\data\db\ 64-bit host
=SPRIYAMATHAN
```

- MongoDB starts listening to connections in port **27017**

Mongo Shell

- The mongo shell is an interactive JavaScript interface to MongoDB.
- Use the mongo shell to query and update data as well as perform administrative operations.
- Connect the mongo shell to the running MongoDB instance(server).
- To start the mongo shell, open a new command prompt and type the command **mongo**

```
C:\Users\SPRIYA MATHAN>mongo
MongoDB shell version v3.6.3
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.3
```

Working with Mongo

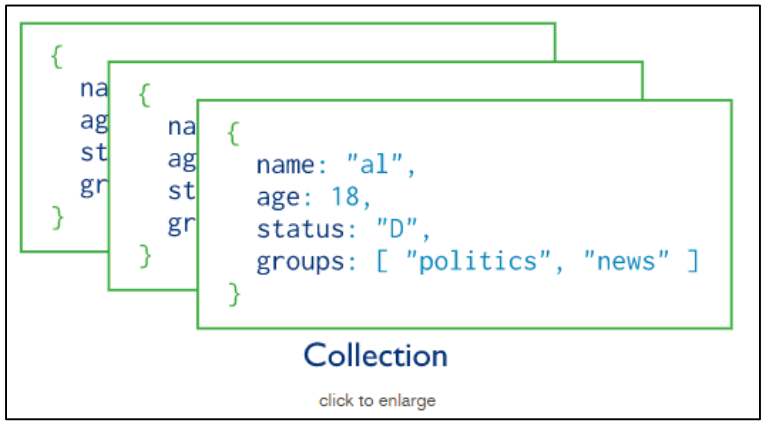
- The command to list the available databases is **show dbs**
- The command to view the current database is **db** – This will display the default database as **test**
- The command to switch databases is **use <database name>**
- The command to create a new database is **use <database-name>.**
- This will be created only if a collection is created and data is added to the collection

```
> show dbs;
admin   0.000GB
config  0.000GB
local   0.000GB
> db
test
> use local
switched to db local
> db
local
```

```
> use training
switched to db training
> db.student.insertOne({name:'Ram'})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5a91718b7cf88600fce1c98a")
}
> db
training
> show dbs;
admin     0.000GB
config    0.000GB
local     0.000GB
training  0.000GB
```

Terms in Mongo - Collection

- A collection is a group of MongoDB documents and is like an RDBMS table.
- Collections are within a single database.
- Collections do not enforce a schema.
- The documents within a collection can have different fields. All documents in a collection have a similar or related purpose.



source: mongodb doc

Terms in Mongo - Document

- A document is similar to a record of RDBMS in a MongoDB collection
- This is made of field-and-value pairs
- Documents are similar to JSON objects, but exist in the database as BSON
- BSON is a binary representation of JSON documents, but has more data types than JSON

Create Collection

- The collection can be created
  - by calling **createCollection()** method with various options, such as setting the maximum size or the documentation validation rules.
  - when data is inserted into the collection using **insertOne()** method

```
> db.createCollection("employee");
{ "ok" : 1 }
```

```
> db.hotel.insertOne({name:'The Keys'})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5a96492df033e6b29dc20fc3")
}
```

Drop a collection

- The method to drop a collection is **db.collection.drop()**
  - Returns true when successfully drops a collection.
  - Returns false when the collection to drop does not exist

```
> db.employee.drop()
true
> db.emp.drop()
false
```

Insert Documents

The documents can be inserted as one or many

- **db.collection.insertOne()**
- **db.collection.insertMany()**

Insert Documents - insertOne

- The method to insert a single document into a collection.

  db.collection.insertOne()

- If the collection does not exist, this will create the collection

- If the document does not specify an _id field, MongoDB adds the _id field with an ObjectId value to the new document.

- This returns a document that includes the newly inserted document's _id field value

```
> db.student.insertOne({
... name:'kumaran',
... id:456,
... hobbies:['sports','reading'],
... address:{
... city:'Bangalore',state:'Kar'
... },
... mobile:'987656789'
... })
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5a92783b7cf88600fce1c98e")
}
```

```
> db.student.insertOne({
... name:'Rahul',
... _id:9001,
... address:{city:'Pune'},
... mobile:987656789
... })
{ "acknowledged" : true, "insertedId" : 9001 }
```

Insert Documents - insertMany

- The method to many documents into a collection.

  db.collection.insertMany()
- Pass an array of documents to the method
- This returns a document that includes the newly inserted documents _id field values.

```
> db.student.insertMany([{name:'Lucky',id:457,hobbies:['music'],address:{city:'chennai'}},
... {name:'Mridhu',id:458,hobbies:['dance'],address:{city:'Bangalore',state:'KAR'}},
... {name:'Raju',id:459,hobbies:['reading'],address:{city:'Pune'}}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("5a941612234eccfb2f3e6e38"),
                ObjectId("5a941612234eccfb2f3e6e39"),
                ObjectId("5a941612234eccfb2f3e6e3a")
        ]
}
```

Query Documents

The documents can be queried using find()

- **db.collection.find()**

The documents can be limited for pagination or skipped while querying

Query filter documents specify the conditions that determine which records to select for read, update, and delete operations.

Query Documents - find()

- To retrieve the document use

  db.collection.find()

```
> db.student.find()
{ "_id" : ObjectId("5a91718b7cf88600fce1c98a"), "name" : "Ram" }
{ "_id" : ObjectId("5a9271917cf88600fce1c98b"), "name" : "Jack", "
ddress" : { "city" : "Bangalore", "state" : "kar" }, "mobile" : 98
{ "_id" : ObjectId("5a9272e57cf88600fce1c98c") }
{ "_id" : ObjectId("5a9273167cf88600fce1c98d"), "name" : "Hanna",
{ "_id" : ObjectId("5a92783b7cf88600fce1c98e"), "name" : "kumaran"
], "address" : { "city" : "Bangalore", "state" : "Kar" }, "mobile"
{ "_id" : ObjectId("5a941612234eccfb2f3e6e38"), "name" : "Lucky",
 { "city" : "chennai" } }
{ "_id" : ObjectId("5a941612234eccfb2f3e6e39"), "name" : "Mridhu",
: { "city" : "Bangalore", "state" : "KAR" } }
{ "_id" : ObjectId("5a941612234eccfb2f3e6e3a"), "name" : "Raju", "
: { "city" : "Pune" } }
{ "_id" : 9897, "name" : "Omkar" }
{ "_id" : 9001, "name" : "Rahul", "address" : { "city" : "Pune" },
```

Query Documents - pretty()

- To retrieve the document and show in a good format use **db.collection.find().pretty()**

```
> db.student.find().pretty();
{ "_id" : ObjectId("5a91718b7cf88600fce1c98a"), "name" : "Ram" }
{
        "_id" : ObjectId("5a9271917cf88600fce1c98b"),
        "name" : "Jack",
        "id" : 123,
        "hobbies" : [
                "sports",
                "music"
        ],
        "address" : {
                "city" : "Bangalore",
                "state" : "kar"
        },
        "mobile" : 989001990
}
```

Query Documents - limit()

- To limit the number of documents use **db.collection.find().limit()**

```
> db.student.find().limit(2).pretty()
{ "_id" : ObjectId("5a91718b7cf88600fce1c98a"), "name" : "Ram" }
{
        "_id" : ObjectId("5a9271917cf88600fce1c98b"),
        "name" : "Jack",
        "id" : 123,
        "hobbies" : [
                "sports",
                "music"
        ],
        "address" : {
                "city" : "Bangalore",
                "state" : "kar"
        },
        "mobile" : 989001990
}
```

Query Documents - skip()

- To skip the documents use
  db.collection.find().skip()

```
> db.student.find()
{ "_id" : ObjectId("5a91718b7cf88600fce1c98a"), "name" : "Ram" }
{ "_id" : ObjectId("5a9271917cf88600fce1c98b"), "name" : "Jack", "id" : 123, "hobbies" : [ "sports", "music" ], "a
ddress" : { "city" : "Bangalore", "state" : "kar" }, "mobile" : 989001990 }
{ "_id" : ObjectId("5a9272e57cf88600fce1c98c") }
{ "_id" : ObjectId("5a9273167cf88600fce1c98d"), "name" : "Hanna", "city" : "Delhi" }
{ "_id" : ObjectId("5a941612234eccfb2f3e6e39"), "name" : "Mridhula", "id" : 458, "hobbies" : [ "dance" ], "address
" : { "city" : "Udupi", "state" : "KAR" }, "address:city" : "OOty", "address:state" : "TN" }
{ "_id" : ObjectId("5a941612234eccfb2f3e6e3a"), "name" : "Raju", "id" : 459, "hobbies" : [ "reading" ], "address"
: { "city" : "Pune" }, "address:city" : "OOty", "address:state" : "TN" }
{ "_id" : 9001, "name" : "Rahul", "address" : { "city" : "Pune" }, "mobile" : 987656789 }
>
>
> db.student.find().skip(3)
{ "_id" : ObjectId("5a9273167cf88600fce1c98d"), "name" : "Hanna", "city" : "Delhi" }
{ "_id" : ObjectId("5a941612234eccfb2f3e6e39"), "name" : "Mridhula", "id" : 458, "hobbies" : [ "dance" ], "address
" : { "city" : "Udupi", "state" : "KAR" }, "address:city" : "OOty", "address:state" : "TN" }
{ "_id" : ObjectId("5a941612234eccfb2f3e6e3a"), "name" : "Raju", "id" : 459, "hobbies" : [ "reading" ], "address"
: { "city" : "Pune" }, "address:city" : "OOty", "address:state" : "TN" }
{ "_id" : 9001, "name" : "Rahul", "address" : { "city" : "Pune" }, "mobile" : 987656789 }
>
```

Query Documents - filter

- Query filter documents specify the conditions that determine which records to select for read, update, and delete operations.
- Specify conditions, using =
  - **db.find({field:value})**
- Specify conditions using query operators
  - **db.find({field:{<operator1:value>},….})**
- Specify AND conditions
  - **db.find({field:value, field:value}})**
- Specify OR conditions
  - **db.find({$or:[{field:value},{field:value}]})**
- Specify a query condition on fields in an embedded/nested document
  - **db.find('field.nestedField').- should be in quotes**

Query Documents - filter

```
> db.student.find({name:'Rahul'}).pretty()
{
        "_id" : 9001,
        "name" : "Rahul",
        "address" : {
                "city" : "Pune"
        },
        "mobile" : 987656789
}
```

Using equality

```
> db.student.find({name:{$in:['Ram','Omkar']}})
{ "_id" : ObjectId("5a91718b7cf88600fce1c98a"), "name" : "Ram" }
{ "_id" : 9897, "name" : "Omkar" }
```

Using query operator

Query on Nested Documents

```
> db.student.find({"address.city":'Pune'}).pretty()
{
        "_id" : ObjectId("5a941612234eccfb2f3e6e3a"),
        "name" : "Raju",
        "id" : 459,
        "hobbies" : [
                "reading"
        ],
        "address" : {
                "city" : "Pune"
        }
}
```

Simple Query

```
> db.student.find({"address.city":{$in:['Bangalore','Pune']}}).pretty()
{
        "_id" : ObjectId("5a9271917cf88600fce1c98b"),
        "name" : "Jack",
        "id" : 123,
        "hobbies" : [
                "sports",
                "music"
        ],
        "address" : {
                "city" : "Bangalore",
                "state" : "kar"
        },
        "mobile" : 989001990
}
```

Complex query

Query Documents - AND

Using AND

```
> db.student.find({name:'Mridhu',id:458}).pretty()
{
        "_id" : ObjectId("5a941612234eccfb2f3e6e39"),
        "name" : "Mridhu",
        "id" : 458,
        "hobbies" : [
                "dance"
        ],
        "address" : {
                "city" : "Bangalore",
                "state" : "KAR"
        }
}
```

```
> db.student.find({name:'Mridhu',"address.city":'Bangalore'}).pretty()
{
        "_id" : ObjectId("5a941612234eccfb2f3e6e39"),
        "name" : "Mridhu",
        "id" : 458,
        "hobbies" : [
                "dance"
        ],
        "address" : {
                "city" : "Bangalore",
                "state" : "KAR"
        }
}
```

Query Documents - $or

- performs a logical OR operation on an array of two or more <expressions>
- selects the documents that satisfy at least one of the <expressions>

```
{ $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }
```

```
> db.student.find({$or:[{name:'Mridhu'},{city:'Delhi'}]}).pretty()
{
        "_id" : ObjectId("5a9273167cf88600fce1c98d"),
        "name" : "Hanna",
        "city" : "Delhi"
}
{
        "_id" : ObjectId("5a941612234eccfb2f3e6e39"),
        "name" : "Mridhu",
        "id" : 458,
        "hobbies" : [
                "dance"
        ],
        "address" : {
                "city" : "Bangalore",
                "state" : "KAR"
        }
}
```

Query Documents - $and

- performs a logical AND operation on an array of two or more expressions (e.g. <expression1>, <expression2>, etc.)
- selects the documents that satisfy all the expressions in the array

```
> db.hotels.find({$and:[{name:'Keys'},{city:'BANGALORE'}]}).pretty()
{
        "_id" : ObjectId("5af50f95c34d6e405c0aabe0"),
        "name" : "Keys",
        "cuisine" : "SI",
        "city" : "BANGALORE",
        "rating" : 5,
        "type" : [
                "Veg",
                "Non-Veg"
        ]
}
```

Query Documents - $nor

- performs a logical NOR operation on an array of one or more query expression
- selects the documents that fail all the query expressions in the array

```
> db.hotels.find({$nor:[{name:'Keys'},{city:'Bangalore'}]}).pretty()
{ "_id" : ObjectId("5a9e19eafcf3784efc7632aa"), "city" : "Pune" }
{
        "_id" : ObjectId("5af2cc600180820838669 0a8"),
        "name" : "A2B",
        "cuisine" : "SI",
        "city" : "Chennai",
        "rating" : 5,
        "type" : [
                "Veg",
                "Non-Veg"
        ]
}
```

Query Documents - $not

- performs a logical NOT operation on the specified <operator-expression>
- selects the documents that do not match the <operator-expression>

```
{ field: { $not: { <operator-expression> } } }
```

```
> db.hotels.find({rating:{$not:{$gt:7}}}).pretty()
{
        "_id" : ObjectId("5a9d6f4b3018d95214e1cceb"),
        "name" : "Keys",
        "cuisine" : "Indian",
        "city" : "Delhi",
        "rating" : 7,
        "type" : [
                "Veg",
                "Non-Veg"
        ]
```

Query an Array - for all elements

```
> db.student.find({hobbies:['music']}).pretty()
{
        "_id" : ObjectId("5a941612234eccfb2f3e6e38"),
        "name" : "Lucky",
        "id" : 457,
        "hobbies" : [
                "music"
        ],
        "address" : {
                "city" : "chennai"
        }
}
>
```

Query for all the elements in the array

```
> db.student.find({$or:[{name:'Mridhu'},{hobbies:['music']}]}).pretty()
{
        "_id" : ObjectId("5a941612234eccfb2f3e6e38"),
        "name" : "Lucky",
        "id" : 457,
        "hobbies" : [
                "music"
        ],
        "address" : {
                "city" : "chennai"
        }
}
{
        "_id" : ObjectId("5a941612234eccfb2f3e6e39"),
        "name" : "Mridhu",
        "id" : 458,
        "hobbies" : [
                "dance"
        ],
        "address" : {
                "city" : "Bangalore",
                "state" : "KAR"
        }
}
```

Query an Array – atleast one element

```
> db.student.find({hobbies:'music'}).pretty()
{
        "_id" : ObjectId("5a9271917cf88600fce1c98b"),
        "name" : "Jack",
        "id" : 123,
        "hobbies" : [
                "sports",
                "music"
        ],
        "address" : {
                "city" : "Bangalore",
                "state" : "kar"
        },
        "mobile" : 989001990
}
{
        "_id" : ObjectId("5a941612234eccfb2f3e6e38"),
        "name" : "Lucky",
        "id" : 457,
        "hobbies" : [
                "music"
        ],
        "address" : {
                "city" : "chennai"
        }
}
```

Query for atleast one element in the array

Query Documents - Sort()

```
> db.employee.find().sort({'city':+1}).pretty()
{
        "_id" : ObjectId("5af1248d107a6e0e8360ece0"),
        "name" : "Raju",
        "empid" : 20,
        "salary" : 9000.3,
        "hobbies" : [
                "sports",
                "music"
        ],
        "address" : {
                "city" : "Bangalore",
                "state" : "KAR",
                "zip" : 989898
        }
}
{
        "_id" : ObjectId("5af1231f107a6e0e8360ecdf"),
        "name" : "Ram",
        "city" : "Bangalore"
}
{ "_id" : 23, "name" : "Kavitha", "city" : "Chennai" }
{ "_id" : 21, "name" : "Kiran", "city" : "Pune" }
{ "_id" : 22, "name" : "Rohan", "city" : "Pune" }
```

Sort by city

Update Documents

The documents can be updated as one or many

- **db.collection.updateOne(<filter>, <update>, <options>)**
- **db.collection.updateMany(<filter>, <update>, <options>)**
- **db.collection.replaceOne(<filter>, <replacement>, <options>)**

Update Documents - single

- To update one document - **updateOne()**
- uses the $set operator to update the value
- If property not available the field will be added
- If the criteria does not match, nothing will change(
- If the criteria does not match, still to create new document use **upsert**

```
> db.student.updateOne(
... {id:457},
... {
... $set:{"address.city":'Mysore'}
... })
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
>
```

Update Documents - upsert

- If set to true, creates a new document when no document matches the criteria.
- The default value is false, which does not insert a new document when no match is found.

```
> db.student.update({id:451},{$set:{'address.state':'KAR'}},{upsert:true})
WriteResult({
        "nMatched" : 0,
        "nUpserted" : 1,
        "nModified" : 0,
        "_id" : ObjectId("5bc8aea1559f3423cf2d33cf")
})
```

Update Documents - multiple

- To update many documents - **updateMany()**
- uses the $set operator to update the value
- Check for **id>457** and change the address

```
> db.student.updateMany(
... { id:{$gt:457} } ,
... { $set:
... {"address.city":'Bangalore',"address.state":'KAR'}
... })
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
```

- To update all the documents,

```
> db.student.updateMany({},{$set:{'college':'ABC'}})
{ "acknowledged" : true, "matchedCount" : 5, "modifiedCount" : 5 }
```

Update array - $push

- To update many documents - **updateMany()**

```
> db.student.updateMany({},{$push:{hobbies:{$each:['music','sports']}}})
```

Add or remove a field

- To add a field in a document - use the $set operator

```
> db.student.updateOne(
... {id:459},
... { $set:{mobile:9876789081}
... })
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

new field mobile added

- To remove a field in a document - use the $unset operator

```
> db.student.updateOne(
... {id:458},
... {$unset:{hobbies:['dance']}
... })
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Replace a document - replaceOne()

- To replace the entire content of a document except for the _id field, pass an entirely new document as the second argument to **db.collection.replaceOne()**
- Cant change the _id field as it is immutable
- Creates a new document if the properties don’t match and removes the properties

```
> db.student.replaceOne(
... {name:'Omkaran'},
... {
... name:'Omkar Vishal',
... address:{
... city:'chennai',state:'TN'
... },
... mobile:987689989,
... hobbies:['cricket']
... })
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Delete Documents

The methods to delete one or many documents is

- **db.collection.deleteOne()**
- **db.collection.deleteMany()**

```
> db.student.deleteOne({id:456});
{ "acknowledged" : true, "deletedCount" : 1 }
```

```
> db.student.deleteMany({"address.city":'chennai'})
{ "acknowledged" : true, "deletedCount" : 2 }
```

Operators – Query Operators

Comparison Query Operators

- $eq, $gt, $gte, $in, $lt, $lte, $ne, $nin

Logical Query Operators

- $and, $not, $nor, $or

Element Query Operators

- $exists,$type

Element Operators - $exist

Syntax:

```
{ field: { $exists: <boolean> } }
```

- If true, the query matches the documents that contain the field, including documents where the field value is null.
- If false, the query returns only the documents that do not contain the field.

```
> db.student.find({id:{$exists:true, $nin:[458,459]}}).pretty()
{
        "_id" : ObjectId("5a96bbedf033e6b29dc20fc7"),
        "name" : "Lucky",
        "id" : 457,
        "hobbies" : [
                "music"
        ],
```

```
> db.student.find({city: {$exists:false}}).pretty()
{
        "_id" : ObjectId("5a96bbedf033e6b29dc20fc7")
        "name" : "Lucky",
```

Operators – Update Operators

Field Update Operators

- $currentDate, $inc, $min, $max, $mul, $set, $unset, $rename

Array Update Operators

- $pop,$push,$pull,$each,$slice,$[],$sort

Bitwise Update Operators

$inc

- This operator increments a field by a specified value
- Accepts positive and negative values.

```
> db.orders.find({_id:113})
{ "_id" : 113, "name" : "chair", "quantity" : 800, "price" : 100 }
> db.orders.update({_id:113},{$inc:{quantity:+100,price:-50}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.orders.find({_id:113})
{ "_id" : 113, "name" : "chair", "quantity" : 900, "price" : 50 }
```

- If the field does not exist, creates the field, sets the field to the specified value.
- Will give error if the field has null value.

```
> db.orders.update({_id:113},{$inc:{stock:+300}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.orders.find({_id:113})
{ "_id" : 113, "name" : "chair", "quantity" : 900, "price" : 50, "stock
" : 300 }
```

$min and $max

$min

- updates the value of the field to a specified value if the specified value is less than the current value of the field.

```
> db.orders.find({_id:113})
{ "_id" : 113, "name" : "chair", "quantity" : 210, "price" : 600 }
> db.orders.update({_id:113},{$min:{quantity:100}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.orders.find({_id:113})
{ "_id" : 113, "name" : "chair", "quantity" : 100, "price" : 600 }
```

$max

- updates the value of the field to a specified value if the specified value is greater than the current value of the field.

```
> db.orders.update({_id:113},{$max:{price:900}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.orders.find({_id:113})
{ "_id" : 113, "name" : "chair", "quantity" : 100, "price" : 900 }
```

$rename

- To rename a field
- The new name must differ from the existing field name
- If the name already exists, then that will be removed and thos will be added

```
{$rename: { <field1>: <newName1>, <field2>: <newName2>, ... } }
```

```
> db.orders.update({_id:113},{$rename:{quantity:'stock'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.orders.find({_id:113})
{ "_id" : 113, "name" : "chair", "price" : 900, "stock" : 100 }
```

```
> db.orders.update({_id:113},{$rename:{stock:'price'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.orders.find({_id:113})
{ "_id" : 113, "name" : "chair", "price" : 100 }
```

$ and $[]

$

- The positional $ operator identifies an element in an array to update without explicitly specifying the position of the element in the array.

```
> db.student.updateOne({id:457,hobbies:'music'},
{$set:{'hobbies.$':'dance'}})
{ "acknowledged" : true, "matchedCount" : 1, "mo
difiedCount" : 1 }
```

$[]

- all positional operator $[] indicates that the update operator should modify all elements in the specified array field.

```
> db.student.update({},{$set:{'hobbies.$[]'
:['dance','music','sports']}})
WriteResult({ "nMatched" : 1, "nUpserted" :
 0, "nModified" : 1 })
```

Data Models

- Normalized Data models
- Embedded data models

Embedded Data Model

- Embed related data in a single structure or document
- generally known as denormalized models
- Easy to update related data in a single atomic write operation.
- Provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation.

Use embedded model:

- When there is "***contains"*** relationships between entities
- To represent one-to-many relationships between entities.

Embedded data model (denormalized)

source: mongodb manual

Normalized Data Model

- Describes relationships using references between documents.
- References provides more flexibility than embedding.
- May require more round trips to the server.

use normalized data models:

- When embedding results in duplication of data
- To represent more complex many-to-many relationships.
- To model large hierarchical data sets.

Normalized Data model

source: mongodb manual

SchemaValidation

- Adding schema while creating a collection

```
> db.createCollection("employee",{
... validator:{
... $jsonschema:{
... bsontype:"object",
... required:['name','salary','year','dept','address.city','address.street'],
... properties:{
... name:{
... bsontype:'string',
... description:'must be a string and required'
... },
... gender:{
... bsontype:'string',
... description:'must be a string and not required'
... },
... year:{
... bsontype:'int',
... minimum:2000
... ,maximum:2050,
... exclusiveMaximum:false,
... description:'must be between 2000 and 2050'
... },
... dept:{
... enum:['Admin','Mktg','Production'],
... description:'must be within enum values'
... },
... 'address.city':{
... bsonType:'string',
... description:'must be a string and required'
... },
... 'address.street':{
... bsonType:'string',
... description:'must be a string and required'
... }
```

Aggregation

- Aggregation operations process data records and return computed results.
- Aggregation operations group values from multiple documents together and can perform a variety of operations on the grouped data to return a single result.
- Three ways to perform aggregation:
  - the aggregation pipeline,
  - the map-reduce function,
  - single purpose aggregation methods.

Syntax

db.collection.aggregate(AGGREGATE_OPERATION)

Aggregate Expressions

Few Aggregate Expressions are listed

- **$sum, $avg, $min, $max, $first, $last**

Example

$sum:

```
> db.mobile.aggregate([{$group:{_id:"$brand",price:{$sum:"$price"}}}])
```

$min:

```
> db.mobile.aggregate([{$group:{_id:"$brand",price:{$min:"$price"}}}])
```

$max:

```
> db.mobile.aggregate([{$group:{_id:"$brand",price:{$max:"$price"}}}])
```

Example using $group

```
> db.orders.find().pretty()
{
        "_id" : 113,
        "name" : "chair",
        "quantity" : 900,
        "price" : 50,
        "stock" : 300,
        "category" : "furniture"
}
{
        "_id" : 114,
        "name" : "laptop",
        "quantity" : 2000,
        "price" : 11800,
        "category" : "electronics"
}
```

```
> db.orders.aggregate([{$group:{_id:'$category',total_quantity:{$sum:'$quantity'}}}])
{ "_id" : "sports", "total_quantity" : 1690 }
{ "_id" : "electronics", "total_quantity" : 5200 }
{ "_id" : "leather goods", "total_quantity" : 600 }
{ "_id" : "gadgets", "total_quantity" : 770 }
{ "_id" : "furniture", "total_quantity" : 2660 }
```

Aggregation pipeline

- The aggregation pipeline consists of stages.
- Each stage transforms the documents as they pass through the pipeline.
- The stages may or may not one output document for every input document; e.g., some stages may generate new documents or filter out documents.
- Pipeline stages can appear multiple times in the pipeline..

$match, $sort

- **$match -** filters the documents to pass only those documents that match the specified condition(s) to the next pipeline stage.

```
> db.orders.aggregate([{$match:{price:{$gt:2500}}},
{$group:{_id:'$category',quant_total:{$sum:'$quanti
ty'}}}])
```

- **$sort -** Sorts all the documents returns them to the pipeline in sorted order.

```
> db.orders.aggregate([{$match:{price:{$gt:2500}}},
{$group:{_id:'$category',quant_total:{$sum:'$quanti
ty'}}},{$sort:{quant_total:1}}])
```

$limit, $skip,

```
> db.orders.aggregate([{$group:{_id:'$category'
,quant_total:{$sum:'$quantity'}}},{$limit:5}])
```

```
> db.orders.aggregate([{$group:{_id:'$category'
,quant_total:{$sum:'$quantity'}}},{$limit:5},{$
skip:2}])
```

$project

- Similar to select

```
> db.orders.aggregate([{$project:{name:1,quantity:1}}])
```

```
> db.orders.aggregate([{$project:{_id:0,name:1,quantity:1}}])
```

Id will be omitted

```
> db.orders.aggregate([{$project:{name:1,quant
ity:1,category:1}},{$match:{quantity:{$gt:250}
}},{$group:{_id:'$category',quant_total:{$sum:
'$quantity'}}}])
```

$unwind()

- Deconstructs an array field from the input documents and creates a new document for each element.
- Each output document becomes the input document with the value of the array field replaced by the element.

```
> db.hotels.aggregate([{$unwind:'$cuisine'}])
```

```
> db.hotels.aggregate([{$match:{rating:{$gte:7}}},
{$unwind:'$cuisine'},{$sort:{name:1}}])
```

Single Purpose Aggregation Operations

Operations on single document

db.collection.count() – to get the total number of documents

db.collection.distinct() – to get the distinct values

```
> db.mobile.count()
12
> db.mobile.distinct("brand")
[ "Samsung", "Sony", "iPhone" ]
> db.mobile.distinct("name")
[ "A123", "B123", "C123" ]
> db.mobile.count({brand:'Samsung'})
4
```

Indexing

- Supports the efficient execution of queries in MongoDB.
- Helps MongoDB to find documents that match the query criteria without performing a collection scan
- Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form.

Indexing

- The index stores the value of a specific field or set of fields, ordered by the value of the field.
- MongoDB creates a unique index on the **_id** field during the creation of a collection.
- To create an index in the Mongo Shell, use **db.collection.createIndex()**

Types

- Single Index
- Compound Index(32 fields)
- MultiKey Index (arrays)
- Text indexes
- 2D Indexes
- Geospatial Index

Methods

To drop a index, pass the name of the index to the

- **db.collection.dropIndex()**

To get the name of the index, run

- **db.collection.getIndexes()**

Summary

- Overview of Nosql
- Types of NoSQL databases
- Introduction to MongoDB (version – 3.6)
- Querying in MongoDB
- Indexing in MongoDB
- Integrating Mongo with a java application

Thank you