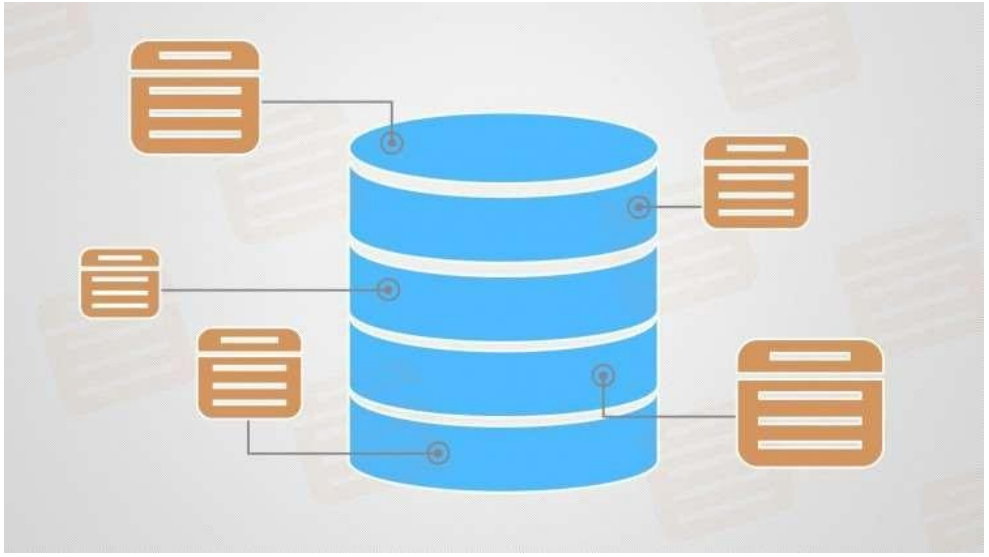
CODETANTRA

What is Data?

- ✓ Data is a collection of a small unit of information.
- ✓ It can be used in a variety of forms like text, numbers, bytes, etc.
- ✓ it can be stored in pieces of paper or electronic memory, etc.

What is Database?

- ✓ A database is an organized collection of data, so that it can be easily accessed and managed.
- ✓ we can organize data into tables, rows, columns, and index it to make it easier to find relevant information.



There are two most common types of databases.

1. SQL Databases
2. NOSQL Databases

Relational Databases/SQL Databases:

- ✓ The data will be stored in tables and these tables have fixed structure.
- ✓ Example: Employee(eno, ename, esal, eaddr)
- ✓ The data stored in the tables has relationships like one to one, one to many, many to one, etc.
- ✓ To retrieve data from relational databases, we have to write join queries which collects data from different tables.
- ✓ Example DB: Oracle, mySQL, etc

Relational Databases/SQL Databases:

NoSQL Databases

- ✓ 'NoSQL' stands for Not Only SQL.
- ✓ Non-relational database for handling large, unstructured or semi-structured data.
- ✓ Focuses on scalability, flexibility, and high performance.

Key Characteristics:

- ✓ Schema-less structure — data can evolve easily.
- ✓ Horizontally scalable and fault-tolerant.
- ✓ Fast data access optimized for specific data models.

Types of NoSQL Databases:

- Key-Value Store: Redis, DynamoDB
- Document Store: MongoDB, CouchDB
- Column-Family Store: Cassandra, HBase
- Graph Database: Neo4j, ArangoDB

Example:

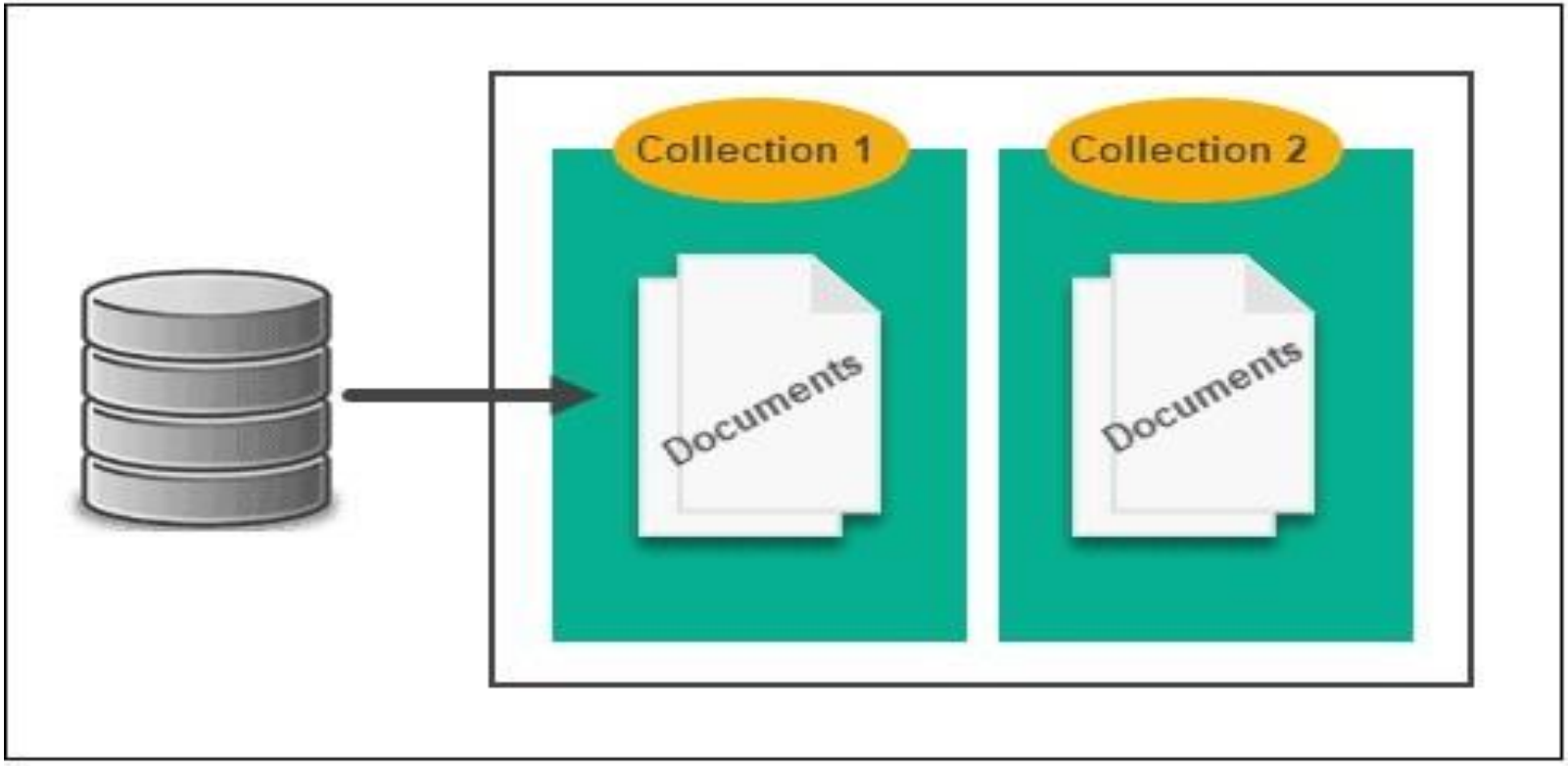
MongoDB stores data in BSON (Binary JSON) format, allowing nested documents and arrays.

Difference Between SQL and NoSQL

Feature	SQL Databases	NoSQL Databases
Data Model	Relational (tables with rows & columns)	Non-relational (documents, key-value, graph, column-based)
Schema	Fixed, predefined schema	Dynamic, flexible schema
Scalability	Vertically scalable (scale-up)	Horizontally scalable (scale-out)
Query Language	Uses SQL (Structured Query Language)	Uses varied APIs or query methods
Data Storage	Stores structured data	Stores structured, semi-structured, or unstructured data
Transactions	Follows ACID properties	Often follows BASE properties
Examples	MySQL, PostgreSQL, Oracle, SQL Server	MongoDB, Cassandra, Redis, CouchDB
Best For	Complex queries, consistency, relational data	High scalability, flexibility, large unstructured data

SQL → Structured, consistent
NoSQL → Flexible, scalable, ideal for distributed apps

What is Document Data Base



Document Databases:

- ✓ Data will be stored in separate documents and each document is independent of others.
- ✓ It doesn’t have a fixed structure/ schema.
- ✓ Example DB: MongoDB, CouchDB, etc.

What is Mongo DB?

- ✓ Mongo DB is an open source, document oriented database that stores data in form of documents (key and value pairs).

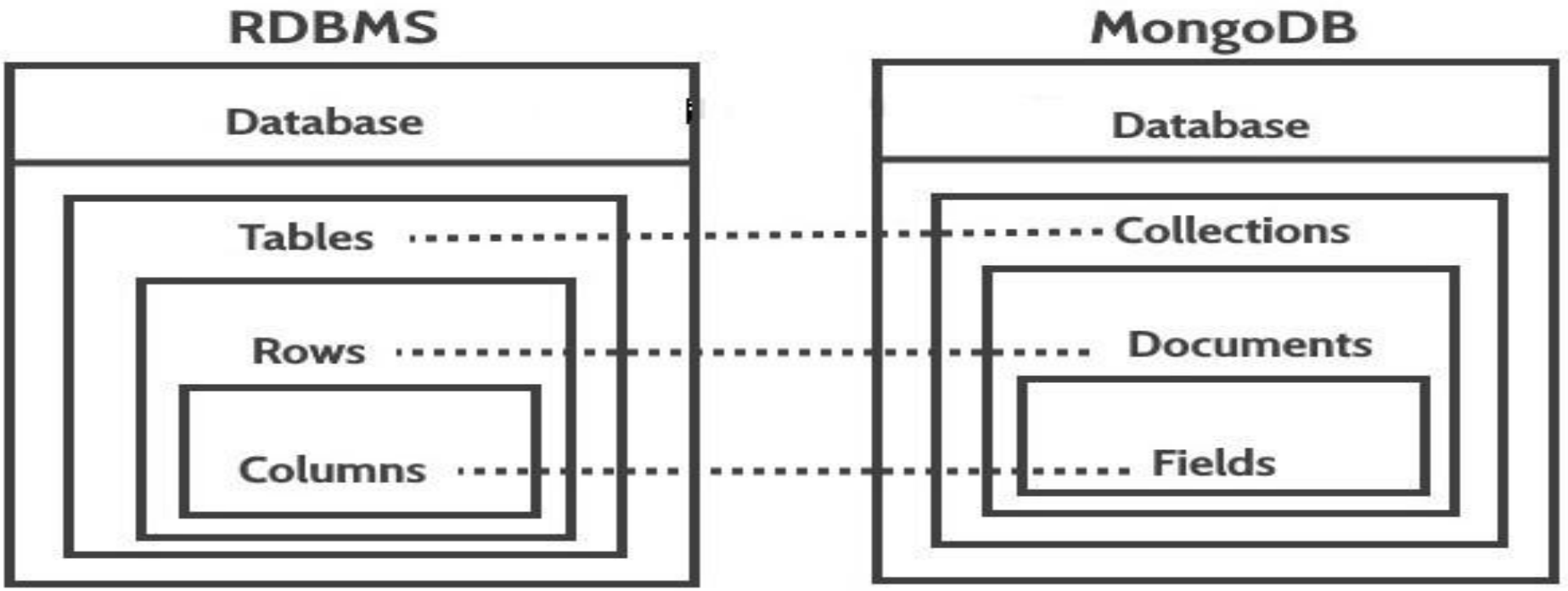
- ✓ MongoDB was founded in 2007 by Dwight Merriman, Eliot Horowitz and Kevin Ryan – the team behind DoubleClick.

Key Characteristics of Mongo DB Database

- ✓ All information related to a document will be stored in a single place.
- ✓ To retrieve data, it is not required to perform a join operation and hence retrieval is very fast.
- ✓ Documents are independent of each other and no schema. Hence we can store unstructured data like, videos, audio files etc.

- ✓ We can perform operations like editing existing document, deleting document & inserting new documents very easy.
- ✓ Data store/retrieval data is in the form of document (JSON) which can be understandable by any programming language without any conversion.
- ✓ We can store very huge amount of data & hence scalability is more.
- ✓ Note: performance & flexibility are biggest assert in MongoDB

Mapping relational database to Mongo DB



- ✓ **Collections** in MongoDB is equivalent to the tables in RDBMS.
- ✓ **Documents** in MongoDB is equivalent to the rows in RDBMS.
- ✓ **Fields** in MongoDB is equivalent to the columns in RDBMS.
- ✓ Fields (key and value pairs) are stored in document, documents are stored in collection and collections are stored in database.
- ✓ Data represent in the MongoDB as JSON format.

What is a field?

- ✓ Fields is a key and value pairs
- ✓ Syntax:

  Key : value (or) field_name: value
- ✓ Example:

  name: “Ravi”

  age: 25

What is Document?

- ✓ The document is the unit of storing data in a MongoDB database.
- ✓ document use JSON (JavaScript Object Notation) style for storing data.
- ✓ Example

Data types in Mongo DB

Data Types	Description	Example
string	May be an empty string or a combination of characters.	“name”: ”Ravi”
integer	Sequence of digits.	“age”:18
boolean	Logical values True or False.	“pass”:true
double	A type of floating point number.	“marks”: 45.78
null	Not zero, not empty.	“mobno”:null

Data Types	Description	Example
array	A list of values.	“skills”:[“c”,”python”]
object	An entity which can be used in programming. May be a value, variable, function, or data structure.	{“name”:”amar”, “age”:24}
Object Id	Whenever we create a new document in the collection MongoDB automatically creates a unique object id for that document(if the document does not have it). There is an _id field in MongoDB for each document.	“_id”:ObjectId(....)

How to find MongoDB version

- ✓ db.version() is get the current working version

  Syntax:

```
db.version()
```

```
4.2.19
```

How to find all databases

Show all available databases

syntax:

```
show dbs
or
db.adminCommand("listDatabases")
or
db.getMongo().getDBNames()
```

Create a Database

- ✓ use database_name is used to create database.
- ✓ The command will create a new database if it doesn't exist, otherwise it will return the existing database.
- ✓ Default current database is a test database

Syntax

```
use database_name
```

Example

If we want to use a database with name **<mydb>**

```
>use mydb
switched to db mydb
```

How to Check current selected Database

We can get current database is

```
>db
mydb
```

or

```
>db.getName()
mydb
```

How to Drop a Database

- ✓ The dropDatabase command is used to drop a database.
- ✓ It also deletes the associated data files.
- ✓ It operates on the current database.
- ✓ Syntax

```
>db.dropDatabase()
```

- ✓ This will delete the selected database.
- ✓ If you have not selected any database, then it will delete default 'test' database.

Example

```
>db.dropDatabase()
{ "dropped" : "mydb", "ok" : 1 }
```

dropped → which database should be removed. “ok”: 1 → Successfully removed.

Create a Collection

✓ db.createCollection() is used to create collection.

Syntax:

db.createCollection(name, options)

Parameter	Type	Description
Name	String	Name of the collection to be created
Options	Document	(Optional) Specify options about memory size and max

- ✓ Options parameter is optional, so we need to specify only the name of the collection.

Field	Type	Description
capped	Boolean	(Optional) If true, enables a capped collection. Capped collection is a 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.
size	number	(Optional) Specifies a maximum size in bytes for a capped collection. 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 a collections without options parameter

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

“ok”:1 → collection created successfully.

Creating Collection with option Parameter

```
> db.createCollection("mycollection1",{capped:true})
{
        "ok" : 0,
        "errmsg" : "the 'size' field is required when 'capped' is true",
        "code" : 72,
        "codeName" : "InvalidOptions"
}

> db.createCollection("mycollection1", {capped: true, size: 65764,
    max:1000})
{ "ok" : 1 }
```

How to find all Collections

- ✓ Show all available collections in current database.

syntax:

show collections or

show tables or

db.getCollectionNames()

Drop a Collection

- ✓ **db.collection.drop()** is used to drop a collection from the database.

Syntax

```
db.collection_name.drop()
```

Example

Now drop the collection with the name **mycollections**.

```
> db.mycollections.drop()
true
```

The result is “true” then successfully drop the collection.

Creating the Collection in Mongo DB on the fly

- ✓ In MongoDB is that we need not to create collection before we insert document in it.
- ✓ With a single command you can insert a document in the collection and the MongoDB creates that collection on the fly.

Syntax:

```
db.collection_name.insert({key:value, key:value…})
```

Example:

```
> use mydb
switched to db mydb

> db
Mydb

> db.mycollection.insert({
... name:"Raju",
... age:24,
... })
WriteResult({ "nInserted" : 1 })
>
```

“nInserted”:1 → one document is inserted into collection.

Insert Documents

Create or Insert

- ✓ Create or insert operations add new documents to a collection.
- ✓ We need not create collection before we insert the document.
- ✓ In a single command we can insert a document in collection and the MongoDB create the collection on fly.

Insert a Single document into collection

Syntax

```
>db.collection_name.insertOne({key:value, key:value…})
```

Example

Create a document and insert into collection

```
> var doc1 = {
... name: "ravi",
... age: 20,
... rollno:30
... }
> db.mycol.insertOne(doc1)
{
        "acknowledged" : true,
        "insertedId" :
    ObjectId("623d4a3cae87cd8d399b1f2f")
}
>
```

"acknowledged" : true

→ document should be inserted successfully.

"insertedId" : ObjectId("623d4a3cae87cd8d399b1f2f")

→Default object id (or)

Insert the document directly into collection

```
> db.mycol.insertOne({name:"raja", age:21})
{
        "acknowledged" : true,
        "insertedId" :
    ObjectId("623d4ca6ae87cd8d399b1f30")
}
>
```

Document with an Array and insert into collection

```
> var doc2 = {
... name:"Raju",
... age:21,
... subjects: ["eng","mat","sci"]
... }

> db.mycol.insertOne(doc2)
{
        "acknowledged" : true,
        "insertedId" : ObjectId("623d5263ae87cd8d399b1f32")
}
>
```

Create a embedded document and insert into collection

```
> var doc2 = {
... name: "Rajesh",
... age: 21,
... marks: { sub1:45, sub2:56, sub3:45}
... }

> db.mycol.insertOne(doc2)
{
        "acknowledged" : true,
        "insertedId" : ObjectId("623d4e46ae87cd8d399b1f31")
}
>
```

Insert a multiple documents

- ✓ Insert a multiple documents into collection.

Syntax:

```
>db.collection_name.insertMany([
{key:value, key:value…},
{key:value, key:value…}
])
```

Example

```
> var doc3 = [
{ name:"ramesh“, age:21, scores:{sub1:45,sub2:56,sub3:54} },
{name:"ravali“, age:22, subjects:["eng1","mats","sci"]}
]

> db.mycol.insertMany(doc3)
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("623d5779ae87cd8d399b1f33"),
                ObjectId("623d5779ae87cd8d399b1f34")
        ]
}
```

(or)

```
> db.mycol.insertMany([{name:"sanath",
   age:23},{name:"amith", age:24}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("623d58bcae87cd8d399b1f35"),
                ObjectId("623d58bcae87cd8d399b1f36")
        ]
}
>
```

Single document by using insert command

- ✓ Insert command is used insert a single document into collection.

Syntax:

```
>db.collection_name.insert({ key1:val1,
key2:val2,…})
```

Example:

```
>   db.mycol.insert({name:"ramith",age:34}) WriteResult({
"nInserted" : 1 })
```

```
{ "nInserted" : 1 }
```

→ one record/document insert into collection.

Multiple document by using insert command

- ✓ Insert command is used to insert multiple documents into collection.

Syntax:

```
>db.collection_name.insert([
{key1:val1, key2:val2,…},
{key1:val1,key2:val2,…},…
])
```

Example

```
> db.mycol.insert([{name:"babu",age:24},{name:"giri",age:31}])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
>
```

"nInserted" : 2 → two record/document insert into collection.

Indexing in MongoDB

• Indexing improves the performance of search queries in MongoDB by reducing the amount of data scanned.

• By default, MongoDB creates an index on the '_id' field for every collection.

• You can create additional indexes on frequently queried fields to optimize performance.

Syntax:

```
db.collection.createIndex({ field_name: 1 })  // 1 for ascending, -1 for descending
```

Example:

```
db.students.createIndex({ age: 1 })
```

View existing indexes:

```
db.students.getIndexes()
```

Drop an index:

```
db.students.dropIndex({ age: 1 })
```

Query Document

find() method

- ✓ The find() method to retrieve all the documents from a collection.
- ✓ Querying all document in the JSON format.

Syntax

```
>db.collection_name.find({})
```

Example

```
>db.mycol.find()
```

Example

```
> db.mycol.find()
{ "_id" : ObjectId("623d4a3cae87cd8d399b1f2f"), "name"
    : "ravi", "age" : 20, "rollno" : 30 }
{ "_id" : ObjectId("623d4ca6ae87cd8d399b1f30"), "name" :
    "raja", "age" : 21 }
{ "_id" : ObjectId("623d4e46ae87cd8d399b1f31"), "name" : "Rajesh",
    "age" : 21, “marks" : { "sub1" : 45, "sub2" : 56, "sub3" : 45 } }
{ "_id" : ObjectId("623d5263ae87cd8d399b1f32"), "name" : "Raju", "age" :
    21, "subjects" : [ "eng", "mat", "sci" ] }
{ "_id" : ObjectId("623d5779ae87cd8d399b1f33"), "name" : "ramesh",
    "age" : 21, “scores" : { "sub1" : 45, "sub2" : 56, "sub3" : 54 } }
```

- ✓ To improve the readability, we can format the output in JSON format with this command.

Syntax:

```
db.students.find().forEach(printjson)
        or
db.students.find().pretty()
```

Example:

```
db.mycol.find.forEach(printjson)
        or
db.mycol.find().pretty()
```

```
> db.mycol.find().pretty()
{
        "_id" : ObjectId("623d4a3cae87cd8d399b1f2f"), "name" : "ravi",
        "age" : 20,
        "rollno" : 30
}
{
        "_id" : ObjectId("623d4ca6ae87cd8d399b1f30"), "name" : "raja",
        "age" : 21
}
```

Count documents

✓ count() is used to count the documents in collection.

Syntax

```
>db.collection_name.count()
```

Example:

```
> db.mycol.count()

  5

> db.mycol.count({age:{$gt:20}})
  4
```

findOne() method

- ✓ **findOne()** method, that returns only one document.

Syntax

```
>db.collection_name.findOne()
```

Example

```
>  db.mycol.findOne({})
{
        "_id" : ObjectId("623d4a3cae87cd8d399b1f2f"), "name" : "ravi",
        "age" : 20,
        "rollno" : 30
}
>
```

Update documents

Update() method

- ✓ The **update()** method updates the values in the existing document in the collections.

Syntax:

```
>db.collection_name.update({selection_criteria}
    , {$set:{updated_data}})
```

Example:

```
>db.mycol.update({name:"Raju"},
{$set:{name:"amith"}})
```

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

"nMatched" : 1

→ it matches one document.

"nModified" : 1

→it modify one document.

"nUpserted" : 0

→it upsertd is zero documents.

updateOne() method

- ✓ This method updates a single document which matches the given filter

Syntax

```
>db.collection_name.updateOne(<filter>, <update>)
```

Example

```
>db.mycol.updateOne({name:"Raju"},{$set:{age:23}})
```

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

"acknowledged" : true

→ successfully updated.

"matchedCount" : 1

→ matches the one document.

"modifiedCount" : 1

→modified one document.

updateMany() method

- ✓ The updateMany() method updates all the documents that matches the given filter.

Syntax

```
>db.collection_name.updateMany(<filter>,
    <update>)
```

Example

```
>db.mycol.updateMany({age:{$gt:18}},
{$set:{active:true}})
```

```
{ "acknowledged" : true, "matchedCount" : 6, "modifiedCount" : 6 }
```

"acknowledged" : true

→ successfully updated.

"matchedCount" : 6

→ matches the 6 document.

"modifiedCount" : 6

→modified 6 document.

Delete a Document

deleteOne() method

- ✓ The deleteOne() method allows you to delete a single document from a collection.

Syntax

```
>db.collection_name.deleteOne(filter, option)
```

Example

```
>   db.mycol.deleteOne({name:"ravi"})
```

{ "acknowledged" : true, "deletedCount" : 1 }

"acknowledged" : true

→ successfully deleted.

"deletedCount" : 1

→ one document is deleted.

deleteMany() method

- ✓ The deleteMany() method allows you to remove all documents that match a condition from a collection.

syntax

>db.collection.deleteMany(filter, option)

Example

> db.mycol.deleteMany({age:{$gt:21}})

{ "acknowledged" : true, "deletedCount" : 1 }

Remove() method

- ✓ The remove() method is used for removing the documents from a collection in MongoDB.

Syntax

```
>db.collection_name.remove(delete_criteria, justone)
```

- ✓ deletion criteria – (Optional) deletion criteria according to documents will be removed.
- ✓ justOne – (Optional) if set to true or 1, then remove only one document.

Example

```
> db.mycol.remove({age:{$gt:21}}) WriteResult({
  "nRemoved" : 2 })
```

"nRemoved“ → removed two documents.

Schema Design & Data Modeling in MongoDB

- MongoDB uses a flexible schema model — documents can have varying structures.
- Schema design depends on how data will be accessed and used in queries.
- **Two main approaches to modeling relationships:**
  1. Embedded Documents – Store related data in a single document.

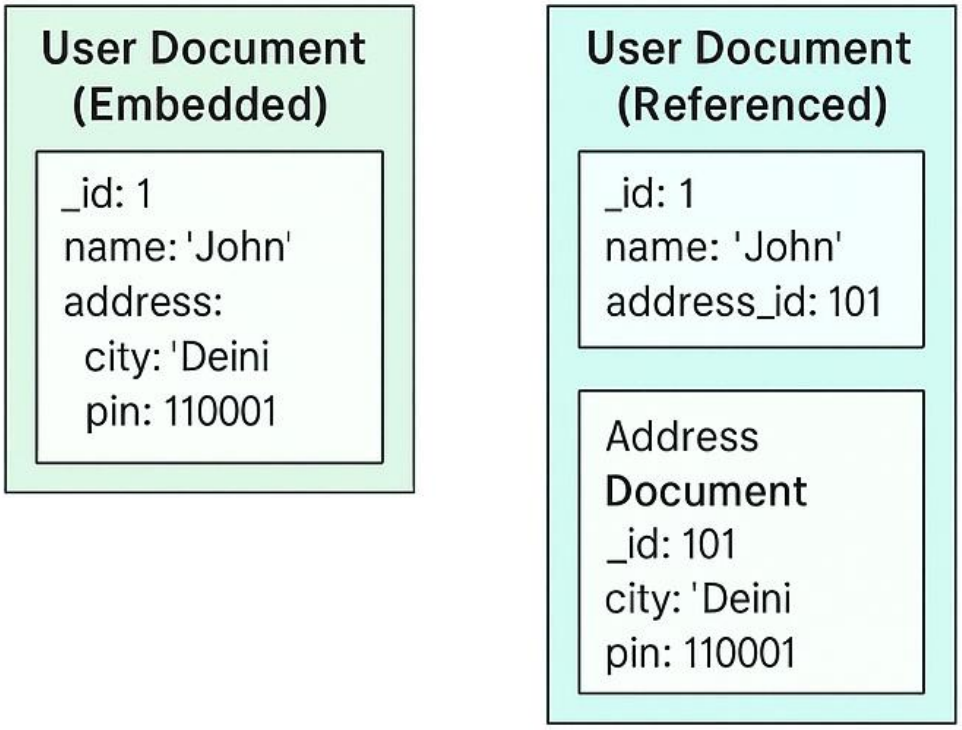
     Example: A user document containing an embedded array of address objects.

  2. Referenced Documents – Store related data in different collections and reference by _id.

     Example: A separate 'users' and 'orders' collection with user_id as a reference.

- MongoDB also supports schema validation using JSON Schema validators in collection creation.

- MongoDB is schema-less, but collections can follow a structurd schema for consistency
- Schema design defines how data is organized, related, and stored within documents
- A good design improves query performance and scalability
- Relationships can be handled using;
  - Embedded Documents ⟶ for tightly coupled data
  - Referenced Documents ⟶ for loosely coupled data




```
// Defining MongoDB schema using Mongoose
const mongoose = require('nongoose');
const userSchema = new mongosse.Schema(
  name: String,
  email: String,
  address: {
    city: Delhi, {°: pujra· 10'ta· pin: 110001} ;
```

Embedded vs Referenced Data Model

Embedded Document Example:

```
{
  _id: 1,
  name: 'John',
  address: { city: 'Delhi', pin: 110001 }
}
```

Referenced Document Example:

```
{ _id: 1, name: 'John', address_id: 101 }
{ _id: 101, city: 'Delhi', pin: 110001 }
```

Embedded → Fast reads, fewer joins
Referenced → Less duplication, better for large or changing data.

MongoDB Schema Example (Mongoose)

```
// Mongoose Schema Example
const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({
  name: String,
  email: String,
  address: { city: String, pin: Number }
});

const User = mongoose.model('User', userSchema);

const newUser = new User({
  name: 'John',
  email: 'john@example.com',
  address: { city: 'Delhi', pin: 110001 }
});
newUser.save();
```

Key Takeaways

- MongoDB supports flexible and scalable schema design.
- Embedded models simplify design and improve read performance.
- Referenced models minimize duplication and handle complex relationships.
- Good schema design improves query efficiency and maintainability.


thank you