No-SQL Database: MongoDB

It's a document-based database. Objects are used as similar to the Javascript.

1. show dbs: to show all the databases

2. use db_name: to create new database and use it.

3. db: to check in which database you are currently.

4. db.createCollection('Collection_name').

Collection is also created just by inserting a record in a new collection.

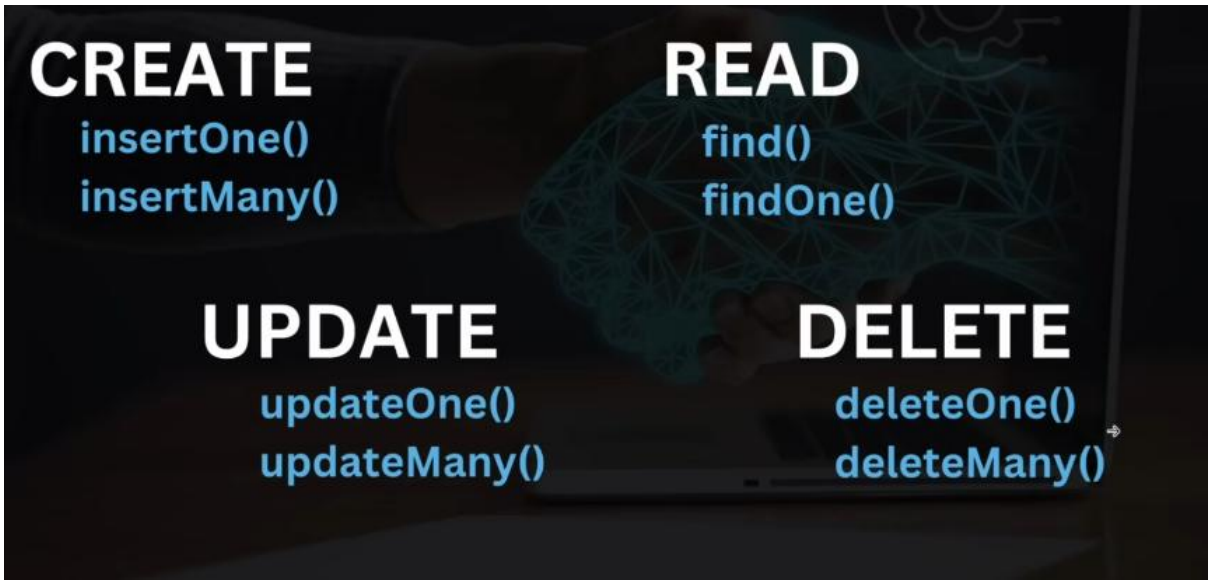
Db.user.insert({name: 'alok', class:'btech'})

5. show collections: to show all the collections.

6. db.dropDatabase(): to drop the database

7. db.collection_name.drop(): to drop the collection.

----------CRUD OPERATIONS IN MONGO----------



----------CAR DEALERSHIP CRUD OPERATIONS----------

SAMPLE DATA FOR PRACTICCE:
https://www.canva.com/design/DAGNfeKTz_Q/nJB6OKFDJxFxiAxklMrVXw/edit

```
switched to db car_dealership
car_dealership>

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

----------INSERT OPERATION----------

insertOne:

```
car_dealership> db.cars.insertOne(
... {
...   "maker": "Tata",
...   "model": "Nexon",
...   "fuel_type": "Petrol",
...   "transmission": "Automatic",
...   "engine": {
...   "type": "Turbocharged",
...   "cc": 1199,
...   "torque": "170 Nm"
...   },
...   "features": [
...   "Touchscreen",
...   "Reverse Camera",
...   "Bluetooth Connectivity"
...   ],
...   "sunroof": false,
...   "airbags": 2
...   }
... 
```

insertMany():

```
car_dealership> db.cars.insertMany(
... [
...  {
...   "maker": "Hyundai",
...   "model": "Creta",
...   "fuel_type": "Diesel",
...   "transmission": "Manual",
...   "engine": {
...   "type": "Naturally Aspirated",
...   "cc": 1493,
...   "torque": "250 Nm"
```

----------READ OPERATION----------

Find(): gives all data - depricated

```
car_dealership> db.cars.find()
[
  {
    _id: ObjectId('66b8a525d95b225bbc381167'),
    maker: 'Tata',
    model: 'Nexon',
    fuel_type: 'Petrol',
    transmission: 'Automatic',
    engine: { type: 'Turbocharged', cc: 1199, torque: '170 Nm' },
    features: [ 'Touchscreen', 'Reverse Camera', 'Bluetooth Connectivity' ],
    sunroof: false,
    airbags: 2
  },
  {
    _id: ObjectId('66b8a5b5d95b225bbc381168'),
    maker: 'Hyundai',
    model: 'Creta',
    fuel_type: 'Diesel',
    transmission: 'Manual',
    engine: { type: 'Naturally Aspirated', cc: 1493, torque: '250 Nm' },
    features: [ 'Sunroof', 'Leather Seats', 'Wireless Charging' ],
    sunroof: true,
    airbags: 6
  },
```

FindOne(): First found data from the collection.

```
car_dealership> db.cars.findOne()
{
  _id: ObjectId('66b8a525d95b225bbc381167'),
  maker: 'Tata',
  model: 'Nexon',
  fuel_type: 'Petrol',
  transmission: 'Automatic',
  engine: { type: 'Turbocharged', cc: 1199, torque: '170 Nm' },
  features: [ 'Touchscreen', 'Reverse Camera', 'Bluetooth Connectivity' ],
  sunroof: false,
  airbags: 2
}
```

Find(): showing specific columns , 1: true, 0:false

```
car_dealership> db.cars.find({},{model:1})
[
  { _id: ObjectId('66b8a525d95b225bbc381167'), model: 'Nexon' },
  { _id: ObjectId('66b8a5b5d95b225bbc381168'), model: 'Creta' },
  { _id: ObjectId('66b8a5b5d95b225bbc381169'), model: 'Baleno' },
  { _id: ObjectId('66b8a5b5d95b225bbc38116a'), model: 'XUV500' },
  { _id: ObjectId('66b8a5b5d95b225bbc38116b'), model: 'City' }
]
car_dealership>
```

Let’s say you only want Model column not _id column, then make _id as 0

```
car_dealership> db.cars.find({},{model:1,_id:0})
[
  { model: 'Nexon' },
  { model: 'Creta' },
  { model: 'Baleno' },
  { model: 'XUV500' },
  { model: 'City' }
]
```

```
car_dealership> db.cars.find({},{model:1,maker:1,_id:0})
[
  { maker: 'Tata', model: 'Nexon' },
  { maker: 'Hyundai', model: 'Creta' },
  { maker: 'Maruti Suzuki', model: 'Baleno' },
  { maker: 'Mahindra', model: 'XUV500' },
  { maker: 'Honda', model: 'City' }
]
```

Find function with condition: I want details of cars having fuel type as ‘Petrol’.

```
car_dealership> db.cars.find({fuel_type:"Petrol"})
```

Find function for nested document:

```
car_dealership> db.cars.find({"engine.type":"Turbocharged"})
[
  {
    _id: ObjectId('66b8a525d95b225bbc381167'),
    maker: 'Tata',
    model: 'Nexon',
    fuel_type: 'Petrol',
    transmission: 'Automatic',
    engine: { type: 'Turbocharged', cc: 1199, torque: '170 Nm' },
    features: [ 'Touchscreen', 'Reverse Camera', 'Bluetooth Connectivity' ],
    sunroof: false,
    airbags: 2
  },
  {
    _id: ObjectId('66b8a5b5d95b225bbc38116a'),
    maker: 'Mahindra',
    model: 'XUV500',
    fuel_type: 'Diesel',
    transmission: 'Manual',
    engine: { type: 'Turbocharged', cc: 2179, torque: '360 Nm' },
    features: [ 'All-Wheel Drive', 'Navigation System', 'Cruise Control' ],
    sunroof: true,
    airbags: 6
  }
]
```

----------UPDATING THE RECORDS----------

UpdateOne:

```
car_dealership> db.cars.updateOne(
... {model:"Nexon"},
... {$set:{color:"Red"}}
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Updating a value inside an array in a document: adding data in features column which is array.

$push:

```
car_dealership> db.cars.updateOne(
... {model:"Nexon"},
... {$push:{features:"Heated Seats"}}
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

$pull:

```
car_dealership> db.cars.updateOne( { model: "Nexon" }, { $pull: { features: "Heated Seats"
 } } )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

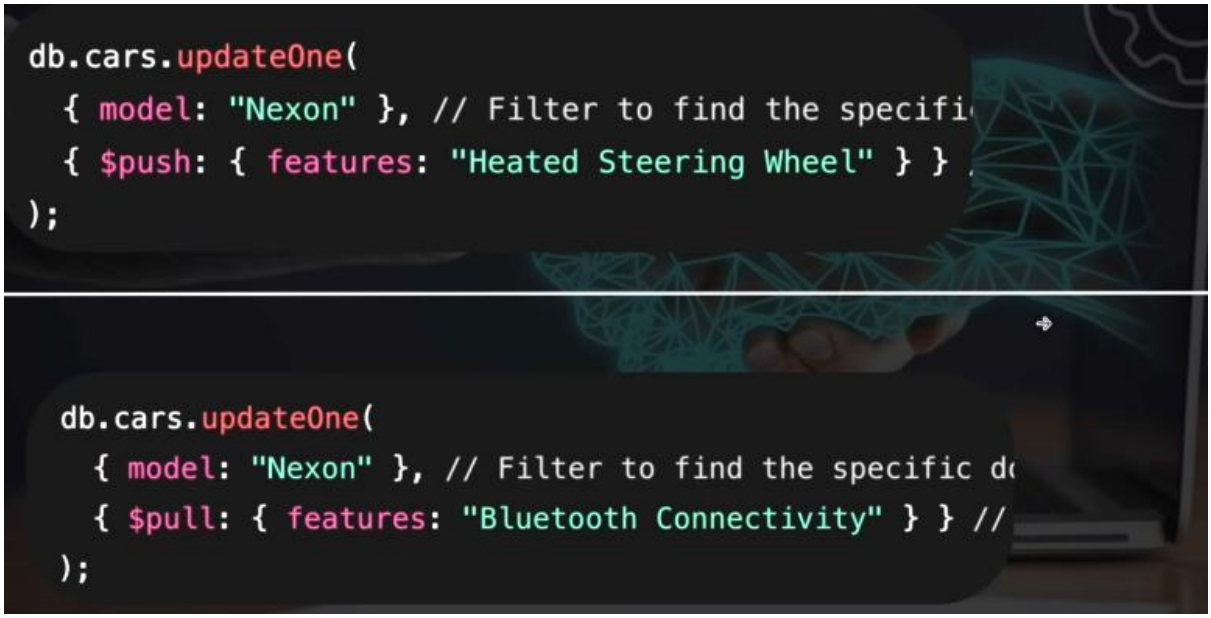
----------UPDATE MANY----------

UpdateMany will do this for every record present inside the collection, while updateOne will only do this for the first matched record only.

```
car_dealership> db.cars.updateMany( { fuel_type: "Diesel" }, { $set: { alloys: "yes" } } )

{
  acknowledged: true,
  insertedId: null,
  matchedCount: 2,
  modifiedCount: 1,
  upsertedCount: 0
}
```

```
car_dealership> db.cars.updateOne(
... {model:"Creta"},
... {$set:{"engine.torque":"270 Nm"}}
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```



----------Updating multiple values inside an array in a document----------

```
car_dealership> db.cars.updateOne(
... {model:"Nexon"},
... {$push:{features:{$each:["Wireless charging", "Voice Control"]}}}
... )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

Removing a field by $unset:

```
car_dealership> db.cars.updateOne( { model: "Nexon" }, { $unset: { color: "" } } )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

--Adding one extra column colors to all the documents in one go.

```
car_dealership> db.cars.updateMany({},{$set:{color:"Blue"}})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 4,
  modifiedCount: 4,
  upsertedCount: 0
}
```

$Upsert: If a document matching your filter exists → **update it,**
If no document matches → **insert a new document = upsert**

```
car_dealership> db.cars.updateMany(
... {model:"Venue"},
... {$set:{Maker:"Hyundai"}},
... {upsert:true}
... )
{
  acknowledged: true,
  insertedId: ObjectId('66d6dc4dc1b5e4adce90fd72'),
  matchedCount: 0,
  modifiedCount: 0,
  upsertedCount: 1
}
```

----------DELETE OPERATION----------

DeleteOne(): will delete the first matching record from the document.

```
car_dealership> db.cars.deleteOne({fuel_type:"Petrol"})
{ acknowledged: true, deletedCount: 1 }
car_dealership>
```

DeleteMany(): db.cars.deleteMany({ fuel_type: "Petrol" })

To delete all data from a collection: db.cars.deleteMany({ })

----------GROUPING IN MONGO DB----------

```
car_dealership> db.cars.aggregate([
...     {$group:{_id:"$maker"}}
... ])
[
  { _id: 'Hyundai' },
  { _id: 'Mahindra' },
  { _id: 'Maruti Suzuki' },
  { _id: 'Honda' },
  { _id: 'Tata' }
]
```

Find number of cars for each specific brand: $sum

```
car_dealership> db.cars.aggregate([
...     {$group:{
...         _id:"$maker",
...         TotalCars: {$sum:1}
...     }}
... ])
[
  { _id: 'Mahindra', TotalCars: 1 },
  { _id: 'Hyundai', TotalCars: 4 },
  { _id: 'Maruti Suzuki', TotalCars: 3 },
  { _id: 'Honda', TotalCars: 3 },
  { _id: 'Tata', TotalCars: 3 }
]
```

When you write $sum: 1, you're telling MongoDB:

"Add 1 for each document in this group."

Eg:

{ maker: "Hyundai" }

{ maker: "Hyundai" }

{ maker: "Tata" }

Hyundai → (1 + 1) = 2

Tata → (1) = 1

Find the number of cars based on different fuel type:

```
car_dealership> db.cars.aggregate([
...     {$group:{
...         _id:"$fuel_type",
...         TotalCars: {$sum:1}
...     }}
... ])
[
  { _id: 'Electric', TotalCars: 2 },
  { _id: 'Petrol', TotalCars: 6 },
  { _id: 'Diesel', TotalCars: 4 },
  { _id: 'CNG', TotalCars: 2 }
]
```

```
db.collection.aggregate([
 {
  $group: {
   _id: "$category",
       totalAmount: { $sum: "$amount" },
       averageAmount: { $avg: "$amount" },
       minAmount: { $min: "$amount" },
       maxAmount: { $max: "$amount" },
      amountsList: { $push: "$amount" },
      uniqueAmounts: { $addToSet: "$amount" }
  }
 }
])
```

<u>SQL over NoSQl – Why?</u> (Why Industries still prefer SQL ?)

Structured Data Management

- SQL databases use a well-defined schema (tables, columns, datatypes, constraints).
- Ensures consistent structure and integrity across all records.

Strong Data Integrity & ACID Compliance

- SQL databases follow ACID properties (Atomicity, Consistency, Isolation, Durability).
- Guarantees reliable transactions — no partial updates or data corruption.

Powerful Query Language (SQL)

- SQL offers a rich, standardized language for complex queries.
- Supports joins, subqueries, aggregation, grouping, sorting, and filtering easily.

Better for Complex Relationships

- Ideal for data that has interconnections (e.g., customers, orders, payments).s
- Supports JOINs and foreign keys for maintaining relational integrity.

Mature Optimization & Indexing

- SQL databases have highly optimized query engines and advanced indexing.
- Ensures faster performance even with large, structured datasets.

Standardization Across Platforms

- SQL syntax is standardized (ANSI SQL), making it portable across systems like MySQL, PostgreSQL, Oracle, etc.

Easier Data Analysis and Reporting

- SQL databases integrate seamlessly with BI tools, analytics, and reporting systems.
- Excellent for OLAP and data warehousing.