MongoDB

MongoDB is an open source NoSQL database management program. **NoSQL (Not only SQL)** is used as an alternative to traditional relational databases. NoSQL databases are quite useful for working with large sets of distributed data. MongoDB is a tool that can manage document-oriented information, store or retrieve information.

MongoDB is used for high-volume data storage, helping organizations store large amounts of data while still performing rapidly. Organizations also use MongoDB for its ad-hoc queries, indexing, load balancing, aggregation, server-side JavaScript execution and other features.

Structured Query Language (SQL) is a standardized programming language that is used to manage relational databases. SQL normalizes data as schemas and tables, and every table has a fixed structure.

Instead of using tables and rows as in relational databases, as a NoSQL database, the MongoDB architecture is made up of collections and documents.

Collections are equivalent of SQL tables, contain document sets. Documents are made up of key-value pairs -- MongoDB's basic unit of data.

Difference between SQL and NoSQL

SQL	**NoSQL**
RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)	Non-relational or distributed database system.
These databases have fixed or static or predefined schema	They have a dynamic schema
These databases are not suited for hierarchical data storage.	These databases are best suited for hierarchical data storage.
These databases are best suited for complex queries	These databases are not so good for complex queries
Vertically Scalable	Horizontally scalable
Examples: MySQL, PostgreSQL, Oracle, MS-SQL Server, etc	**Examples:** MongoDB, GraphQL, HBase, Neo4j, Cassandra, etc

MongoDB vs. RDBMS: What are the differences?

A relational database management system (RDBMS) is a collection of programs and capabilities that let IT teams and others create, update, administer and otherwise interact with a relational database. RDBMS store data in the form of tables and rows. RDBMS most commonly uses SQL.

One of the main differences between MongoDB and RDBMS is that RDBMS is a relational database while MongoDB is nonrelational. Likewise, while most RDBMS systems use SQL to manage stored data, MongoDB uses BSON for data storage -- a type of NoSQL database.

While RDBMS uses tables and rows, MongoDB uses documents and collections. In RDBMS a table -- the equivalent to a MongoDB collection -- stores data as columns and rows. Likewise, a row in RDBMS is the equivalent of a MongoDB document but stores data as structured data items in a table. A column denotes sets of data values, which is the equivalent to a field in MongoDB.

RDBMS	**MongoDB**
Database	**Database**
Table	**Collection** - stores data as columns and rows.
Row	**Document** - stores data as structured data items
Column	**Field** - denotes sets of data values
Primary Key	Primary Key (Default key _id provided by MongoDB itself)

MongoDB is also better suited for hierarchical storage.

Why is MongoDB used?

An organization might want to use MongoDB for the following:

1. **Storage**. MongoDB can store large structured and unstructured data volumes and is scalable vertically and horizontally. Indexes are used to improve search performance. Searches are also done by field, range and expression queries.

2. **Data integration.** This integrates data for applications, including for hybrid and multi-cloud applications.

3. **Complex data structures descriptions.** Document databases enable the embedding of documents to describe nested structures (a structure within a structure) and can tolerate variations in data.

4. **Load balancing.** MongoDB can be used to run over multiple servers.

Features of MongoDB

Features of MongoDB include the following:

1. **Replication.** A replica set is two or more MongoDB instances used to provide high availability. Replica sets are made of primary and secondary servers. The primary MongoDB server performs all the read and write operations, while the secondary replica keeps a copy of the data. If a primary replica fails, the secondary replica is then used.

2. **Scalability.** MongoDB supports vertical and horizontal scaling. Vertical scaling works by adding more power to an existing machine, while horizontal scaling works by adding more machines to a user's resources.

3. **Load balancing.** MongoDB handles load balancing without the need for a separate, dedicated load balancer, through either vertical or horizontal scaling.

4. **Schema-less.** MongoDB is a schema-less database, which means the database can manage data without the need for a blueprint.
(*Schema-less databases are a type of NoSQL database that do not require a predefined schema to store data. Instead, they allow data to be stored in flexible and dynamic formats, such as JSON documents, key-value pairs, graphs, or columns.)

5. **Document**. Data in MongoDB is stored in documents with key-value pairs instead of rows and columns, which makes the data more flexible when compared to SQL databases.

**** <u>Scaling</u>**

Scaling alters the size of a system. In the scaling process, we either compress or expand the system to meet the expected needs. The scaling operation can be achieved by adding resources to meet the smaller expectation in the current system, by adding a new system to the existing one, or both.

Scaling can be categorized into 2 types:

1. **<u>Vertical Scaling:</u>**

When new resources are added to the existing system to meet the expectation, it is known as vertical scaling.

Consider a rack of servers and resources that comprises the existing system. Vertical scaling is based on the idea of adding more power(CPU, RAM) to existing systems, basically adding more resources.

Vertical scaling is not only easy but also cheaper than Horizontal Scaling. It also requires less time to be fixed.

2. **<u>Horizontal Scaling:</u>**

When new server racks are added to the existing system to meet the higher expectation, it is known as horizontal scaling.

Now when the existing system fails to meet the expected needs, and the expected needs cannot be met by just adding resources, we need to add completely new servers. This is considered horizontal scaling. Horizontal scaling is based on the idea of adding more machines to our pool of resources. Horizontal scaling is difficult and also costlier than Vertical Scaling. It also requires more time to be fixed.

Advantages of MongoDB

MongoDB offers several potential benefits:

1. **Schema-less.** Like other NoSQL databases, MongoDB doesn't require predefined schemas. It stores any type of data. This gives users the flexibility to create any number of fields in a document, making it easier to scale MongoDB databases compared to relational databases.
2. **Document-oriented.** One of the advantages of using documents is that these objects map to native data types in several programming languages., Having embedded documents also reduces the need for database joins, which can lower costs.
3. **Scalability.** A core function of MongoDB is its horizontal scalability, which makes it a useful database for companies running big data applications.
4. **Third-party support.** MongoDB supports several storage engines and provides pluggable storage engine APIs that let third parties develop their own storage engines for MongoDB.

Disadvantages of MongoDB

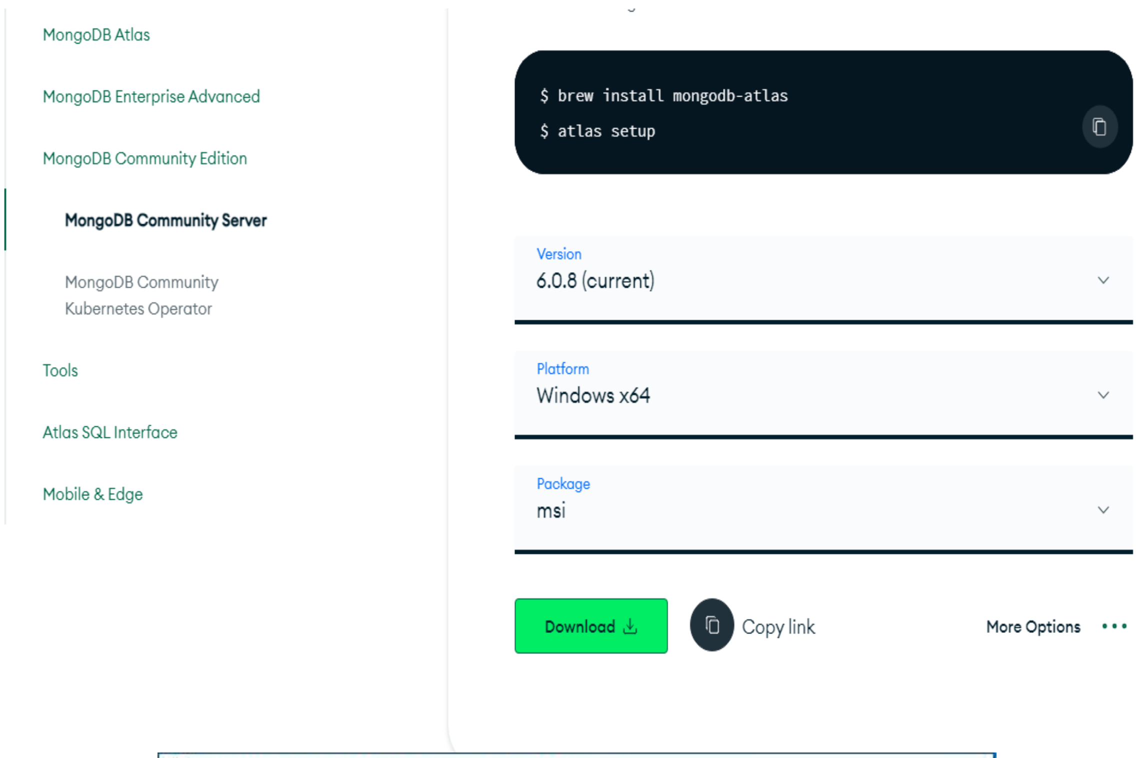
Though there are some valuable benefits to MongoDB, there are some downsides to it as well.

1. **Continuity.** With its automatic failover strategy, a user sets up just one master node in a MongoDB cluster. If the master fails, another node will automatically convert to the new master. This switch promises continuity, but it isn't instantaneous -- it can take up to a minute. By comparison, the Cassandra NoSQL database supports multiple master nodes. If one master goes down, another is standing by, creating a highly available database infrastructure.
2. **Data consistency.** MongoDB doesn't provide full referential integrity through the use of foreign-key constraints, which could affect data consistency.
3. **Security**. In addition, user authentication isn't enabled by default in MongoDB databases. However, malicious hackers have targeted large numbers of unsecured MongoDB systems in attacks, which led to the addition of a default setting that blocks networked connections to databases if they haven't been configured by a database administrator.

How to install and setup MongoDB

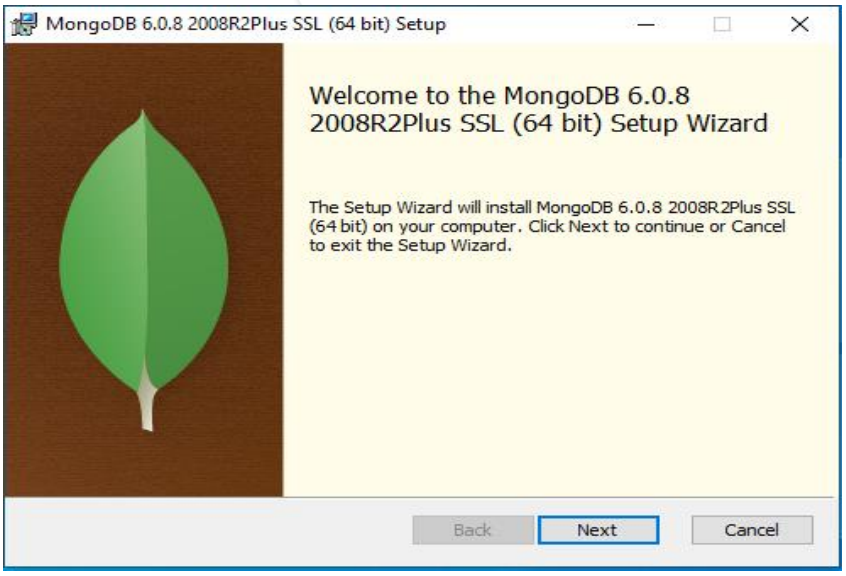
Step 1:

- ✓ Go to https://www.mongodb.com/
- ✓ Under the **Products** tab
  - o Community Edition
    - ▪ Community Server
    - ▪ Select Package
    - ▪ Package (msi)
    - ▪ Download
- ✓ Run and install



```
$ brew install mongodb-atlas
$ atlas setup
```

Version: 6.0.8 (current)
Platform: Windows x64
Package: msi


MongoDB 6.0.8 2008R2Plus SSL (64 bit) Setup

Welcome to the MongoDB 6.0.8 2008R2Plus SSL (64 bit) Setup Wizard

The Setup Wizard will install MongoDB 6.0.8 2008R2Plus SSL (64 bit) on your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Step 2:

Mongo Shell

The mongo shell is an interactive JavaScript interface to MongoDB. You can use the mongo shell to query and update data as well as perform administrative operations.

To download

- ✓ Go to https://www.mongodb.com/try/download/shell
- ✓ Download
- ✓ Extract files and from **bin** folder **Copy exe** and **dll** files to "**C:\Program Files\MongoDB\Server\6.0\bin**"

Step 3:

Set path property in environment variable

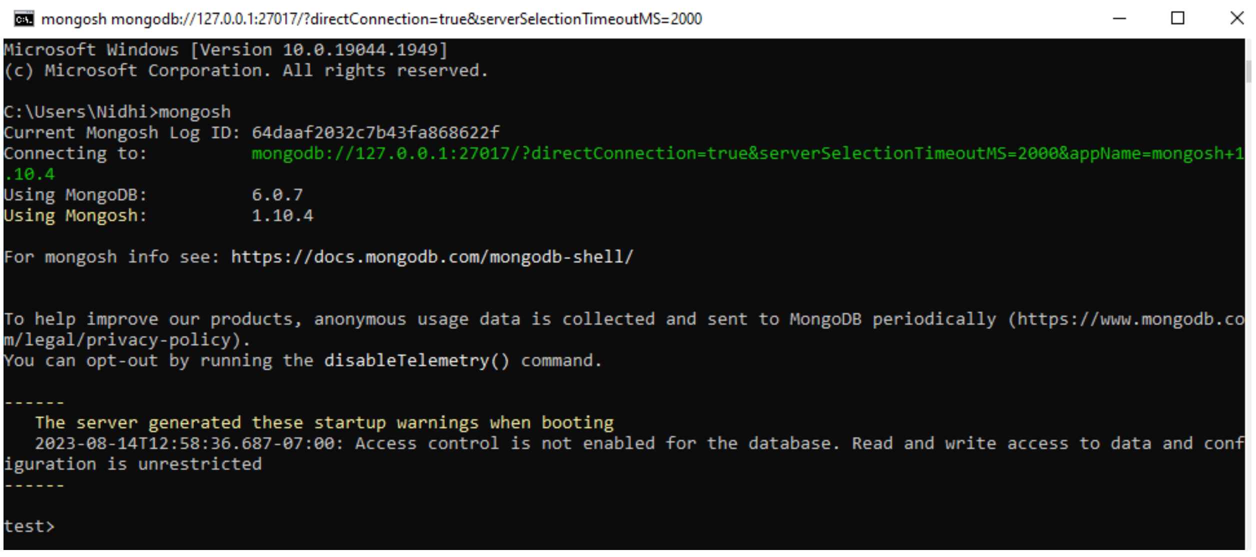
- ✓ Select "System Variable" > Path
- ✓ Click on "Edit"
- ✓ Add "C:\Program Files\MongoDB\Server\6.0\bin"

Step 4:

To check > open Command Prompt

Type **mongosh**

It will shown as below screenshot.



```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
Microsoft Windows [Version 10.0.19044.1949]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Nidhi>mongosh
Current Mongosh Log ID: 64daaf2032c7b43fa868622f
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1
.10.4
Using MongoDB:          6.0.7
Using Mongosh:          1.10.4

For mongosh info see: https://docs.mongodb.com/mongodb-shell/


To help improve our products, anonymous usage data is collected and sent to MongoDB periodically (https://www.mongodb.co
m/legal/privacy-policy).
You can opt-out by running the disableTelemetry() command.

------
   The server generated these startup warnings when booting
   2023-08-14T12:58:36.687-07:00: Access control is not enabled for the database. Read and write access to data and conf
iguration is unrestricted
------

test>
```

- **Default new database**
  - **Test>**

- **To list all the available DBs write below command**
  1. **test>** show dbs
     - admin   40.00 KiB
     - config 72.00 KiB
     - local   76.00 KiB

- **db**
  1. To show current active database.
- **show collections**
  1. Display all available collections in existing database
- **To create/use database**
  1. **test>** use mydb
     - **mydb>**

Insert Data

- **To create collection with only one document**
  1. **mydb**>db.student.insertOne({name:"Test",rollno:41})
     - **Un above example student is collection(table).**
     - **Two fields(column) are "name" and "rollno"**
     - **1st document(row) data are "Test" and "41" for "name" and "rollno" respectively.**

- **To create collection with more than one documents**
  1. **mydb**>db.student.insertMany([{name:"P1",age:20},{name:"P2",age:24}])

Output:

```
 { _id: ObjectId("64dfd8ffc925fb0136d77817"), name: 'P1', age: 20 },
 { _id: ObjectId("64dfd8ffc925fb0136d77818"), name: 'P2', age: 24 }
```

- **We can also store non-uniform document fields as shown in below example.**
  1. **mydb**>db.student.insertMany([
     {name:"P1",age:28,status:"Active"},
     {name:"P2",age:24},
     {name:"P3",age:27,status:"Active",city:"Ahmedabad"}
     ])

So here object contains 3,2 4 fields respectively.

Find Data

- **Find document/Read document**
  1. Mydb>db.student.find()
     - Display all documents of **student** collection

Syntax

```
db.student.find(query,projection)
```

  2. **mydb**>db.student.find({name:"P1"})
     - As an output it will give all the documents with name P1 and also display all the fields as shown below.

Output :

```
[ { _id: ObjectId("64de65b454282c021d807835"), name: 'P1', age: 20 } ]
```

  3. **mydb**> db.student.find({name:"P1"},{_id:0,age:false })
     - As an output it will give all the documents with name P1 and displays only Name field as shown below

```
Output: [ { name: 'P1' } ]
```

Note: To display field write "true" or 1 and to not display field write "false" or 0

Methods:

1. **findOne**() :findOne() method returns only one document that satisfies the criteria entered. If the criteria entered matches for more than one document, the method returns only one document according to natural ordering, which reflects the order in which the documents are stored in the database.
2. **Limit() :**The limit() method limits the number of records or documents that you want. It basically defines the max limit of records/documents that you want. Or in other words, this method uses on cursor to specify the maximum number of documents/ records the cursor will return.
   - **Restrict number of documents to be displayed/read.**
     1. Suppose, P1 name exists in 4 documents and we want to display limited number of documents then we can use **limit** function.
     2. Suppose, we have written 1 in limit then it will display only 1st document of 4 documents.
        - db.student.find({name:"P1"}).limit(1)
          Or
        - db.student.findOne({name:"P1"})

Displays only first document of collection

3. skip() -Call skip() method on a cursor to control where MongoDB begins returning results. This approach may be useful in implementing paginated results.

- **Skip number of documents**
   1. Suppose we have 3 documents with name “P1”. Below command will skip 1st document and give only 2nd document as an output

      - db.student.find({name:"P1"}).limit(1).skip(1)

```
Output:
[ { _id: ObjectId("64de6dbb54282c021d807837"), name: 'P1',
age: 23 } ]
```

Update Database

- **Method: updateOne() and updateMany()**

<ins>**Syntax**</ins>

db.collection.updateOne(filter,document,options)
Updates only one document where filter is matched

db.collection.updateMany(filter,document,options)
Updates only all documents where filter is matched

Parameters:

1. **filter**: The selection criteria for the update, same as find() method.
2. **document**: A document or pipeline that contains modifications to apply.
3. **options**: Optional. May contains options for update behavior. It includes upsert, writeConcern, collation, etc.

- **db.student.updateOne(**

  { name:"P1" },

  {$set:{name:"P4"}}

)

  - Updates only one document with name P4 where name is P1
- **db.student.updateOne({name:"P1",age:23},{$set:{name:"P4"}})**
  - Updates only one document with name P4 where name is P1 and age is 23

- **db.student.updateOne({name:"P1",age:23},{$set:{name:"P4",age:11}})**
  - Updates only one document with name P4 and age 11 where name is P1 and age is 23
- **db.student.updateOne({name:"P1",age:23},{$set:{name:"P4",age: 11}})**

- Updates only one document with name P4 and age 11 where name is P1 and age is 23

➢ **db.student.updateMany({name:"P1"},{$set:{name:"P4"}})**

- Updates all documents with name P4 and where name is P1

Upsert

In MongoDB, upsert is an option that is used for update operation e.g. update(), findAndModify(), etc. Or in other words, upsert is a combination of update and insert (update + insert = upsert).

If the value of this option is set to true and the document or documents found that match the specified query, then the update operation will update the matched document or documents.

Or if the value of this option is set to true and no document or documents matches the specified document, then this option inserts a new document in the collection and this new document have the fields that indicate in the operation.

By default, the value of the upsert option is false. If the value of upsert in a sharded collection is true then you have to include the full shard key in the filter.

```
Syntax: upsert: <boolean>
```

The value of upsert option is either true or false.

```
db.student.updateOne (
                           {age:45},
                            {$set:{name:"PQR"}},
                            {upsert:true}
                         )
```

Delete Database

- **Method: deleteOne() and deleteMany()**

We can delete documents by using themethods deleteOne() or deleteMany().
These methods accept a query object. The matching documents will bedeleted.

deleteOne()

The deleteOne() method will delete the first document that matches the query provided.

- **db.student.deleteOne({name:”P1”})**
  - This deletes only 1st document where name is “P1”

deleteMany()

The deleteMany() method will delete all documents that match the query provided.

- **db.student.deleteMany({})**
  - This deletes all documents of collection

- **db.student.deleteMany({name:”P1”})**
  - This deletes all the documents where name is “P1”

Logical Operator

MongoDB supports logical query operators. These operators are used for filtering the data and getting precise results based on the given conditions. The following table contains the comparison query operators:

Operator	Description
$and	It is used to join query clauses with a logical AND and return all documents that match the given conditions of both clauses.
$or	It is used to join query clauses with a logical OR and return all documents that match the given conditions of either clause.
$not	It is used to invert the effect of the query expressions and return documents that does not match the query expression.
$nor	It is used to join query clauses with a logical NOR and return all documents that fail to match both clauses.

- ✓ **$and**
  - **To fetch documents with more than one conditions and all the conditions must be satisfied.**
    - db.student.find({$and:[{name:"P1"},{age:28}]})
      or
      db.student.find({name:"P1",age:28})

This fetches documents which satisfies both the conditions.

- ✓ **$or**
  - **To fetch documents with more than one conditions and one of the conditions must be satisfied.**
    - db.student.find({$or:[{name:"P1"},{age:28}]})

This fetches documents which satisfies one of the conditions.

- ✓ **$not**
  - It performs a logical NOT operation on the specified <operator-expression> and selects the documents that do not match the <operator-expression>. This includes documents that do not contain the field.

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

db.people.find({ age: { $not: { $lt: 20 } } })

- ✓ **$nor**
  - The $nor is a logical query operator that allows the user to perform a logical NOR operation on an array of one or more query expressions. This operator is also used to select or retrieve documents that do not match all of the given expressions in the array. The user can use this operator in methods like find(), update(), etc., as per their requirements.
    - **db.people.find({$nor: [{name: "P1"}]})**
    - **db.people.find({$nor: [{name: "P1"},{age:20}]})**
      - **It will show documents which do not have name "P1" or age "20"**
      - **To display only name field.**
        - **db.people.find({ age: { $in: [20, 21] } },{name:1,_id:0})**

Comparison Operators

MongoDB comparison operators can be used to compare values in a document. The following table contains the common comparison operators.

Operator	**Description**

$eq	Matches values that are equal to the given value.
	Syntax: { field: { $eq: value } }
	db.people.find({age:{$eq:20}})

$gt	Matches if values are greater than the given value.
	Syntax: { field: { $gt: value } }
	db.people.find({age:{$gt:20}})

$lt	Matches if values are less than the given value.
	Syntax: { field: { $lt: value } }
	db.people.find({age:{$lt:20}})

$gte	Matches if values are greater or equal to the given value.
	Syntax: { field: { $gte: value } }
	db.people.find({age:{$gte:20}})

$lte	Matches if values are less or equal to the given value.
	Syntax: { field: { $lte: value } }
	db.people.find({age:{$lte:20}})

$in	Matches any of the values in an array.
	Syntax: { field: { $nin: [<value1>, <value2>, ... <valueN>] } }
	db.people.find({age:{$in:[45,70]}})

$ne	Matches values that are not equal to the given value.
	Syntax: { field: { $ne: value } }
	db.people.find({age:{$ne:20}})

$nin	Matches none of the values specified in an array.
	Syntax: { field: { $nin: [<value1>, <value2>, ... <valueN>] } }
	db.people.find({age:{$nin:[45,70]}})

- ✓ **Update only one document with branch "CSE" and age "21" where age is equal to 5.**
  - db.people.updateOne({age:{$eq:5}},{$set:{branch:"CSE",age: 21}})
- ✓ **To display all documents where age is greater than 25**
  - db.people.find({age:{$gt:25}})
- ✓ **To display all documents where age is greater than 25 and less than 50**
  - db.people.find({age:{$gt:25,$lt:50}})
  - db.people.find({$and:[{age:{$gt:25}},{age:{$lt:50}}]})
- ✓ **To display exact match of the documents.**
  - db.people.find({age:{$eq:70}})
- ✓ **To display all documents other than age is 70**
  - db.people.find({age:{$ne:70}})
- ✓ **To display Matches any of the values in an array.**
  - db.people.find({age:{$in:[45,70]}})
- ✓ **To display other than Matches any of the values in an array.**
  - db.people.find({age:{$nin:[45,70]}})

Field Update operators

Name	Description
$currentDate	Sets the value of a field to current date, either as a Date or a Timestamp.
$inc	Increments the value of the field by the specified amount.
$mul	Multiplies the value of the field by the specified amount.
$rename	Renames a field.
$set	Sets the value of a field in a document.
$unset	Removes the specified field from a document.

✓ **$inc**

- **{ $inc: { <field1>: <amount1>, <field2>: <amount2>, ... } }**

  Examples:
  - db.table1.updateMany({},{$inc:{age:10}});
    - Increase age field values by 10 for all documents
  - db.table1.updateOne({},{$inc:{age:15}});
    - Increase age field values by 15 of 1st document
  - db.table1.updateOne({},{$inc:{age:-15}});
    - increase age field values by (-15) of 1st document

✓ **$mul**

Multiply the value of a field by a number. To specify a $mul expression, use the following prototype:

{ $mul: { <field1>: <number1>, ... }
}

The field to update must contain a numeric value.

- db.table1.updateOne({},{$mul:{age:15}});
  - Multiply age field by 15 of 1st document

- db.table1.updateMany({name:’N1’},{$mul:{age:0.5}});
  - Multiply age field by 0.5 for all documents where name is “N1”
- db.table1.updateOne({},{$mul:{age:2}});
  - Multiply age field by 0.5 for all documents

- ✓ **$unset**

  The $unset operator deletes a particular field. Consider the following.

  syntax:{ $unset: { <field1>: "", ... } }

  Example:

  db.people.updateOne({age:{$eq:21}},{$unset:{branch:"CSE",age: 21}})

- ✓ **$rename**

  The $rename operator updates the name of a field and has the following form:

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

  ```
  db.people.updateMany({},{$rename:{'name':'uname'}})
  db.people.updateMany({},{$rename:{'name':'Uname',’branch’:’Branch’}})
  ```

- ✓ **createCollection()**
  - db.createCollection(‘student’): to create collection

- ✓ **drop()**
  - db.student.drop(): Drop /delete the collection

- ✓ **Count()**
  - db.collection.count(query)

Returns the count of documents that would match a find() query. The db.collection.count() method does not perform the find() operation but instead counts and returns the number of results that match a query.

The db.collection.count() method is equivalent to the db.collection.find(query).count() construct.

db.people.count({uname:"N1"})

It will give an aswer with below warning.

DeprecationWarning: Collection.count() is deprecated. Use countDocuments or estimatedDocumentCount.

or

db.people.find({uname:"N1"}).count()

- ✓ **sort()**

  Specifies the order in which the query returns matching documents. You must apply sort() to the cursor before retrieving any documents from the database.

  The sort parameter contains field and value pairs, in the following form:

  { field: value }

  Example:

  To display all documents in descending order of age.

  db.student1.find().sort({age:-1})

Ascending: 1 and Descending: -1

MongoDB Cursor

When find() method is used to find a document present in a given collection, then this method returns a pointer which will point to a document of a collection, now pointer is known as cursor.

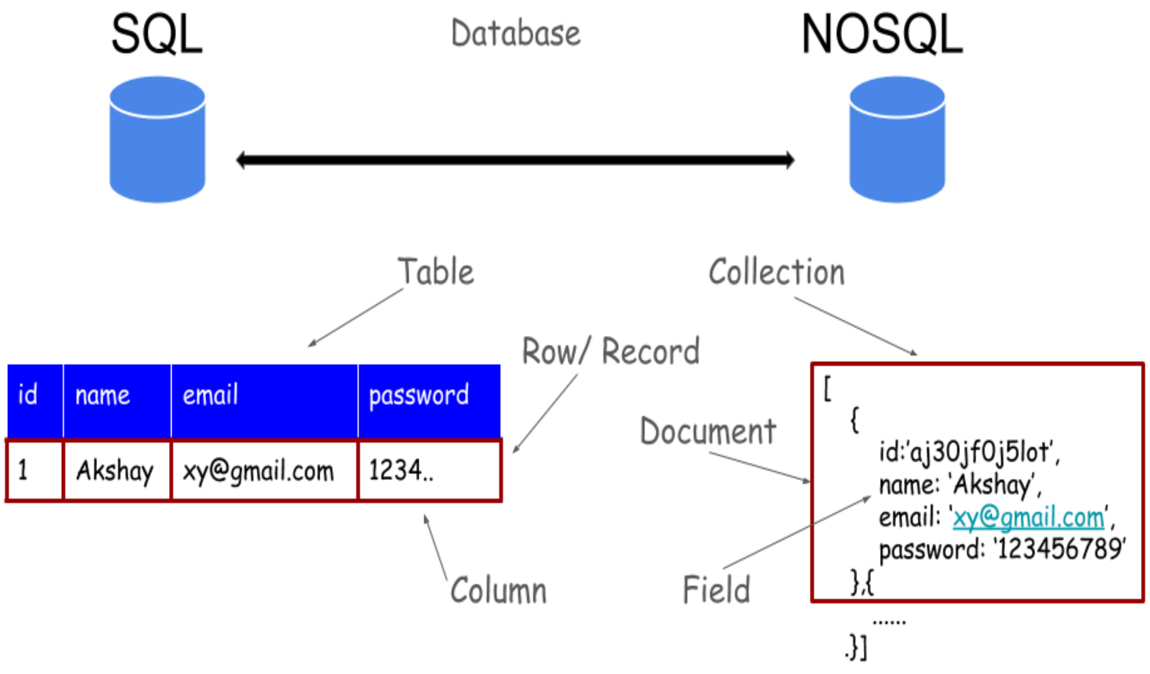
Using this cursor(pointer) also, we can access the document. By default, cursor iterate automatically, but we can also iterate it manually.

For Example:

```
> let rec=db.KPP.find({age:30})

> rec
```

SQL to MongoDB Mapping



<table>
<tr><th>SQL</th><th>MongoDB</th></tr>
<tr><th colspan="2">Create and Alter commands</th></tr>
<tr><td>CREATE TABLE KPP (
  id MEDIUMINT NOT NULL
    AUTO_INCREMENT,
  user_id Varchar(20),
  age Number,
  status char(1),
  PRIMARY KEY (id)
)</td><td>db.createCollection ("KPP”)</td></tr>
<tr><td>ALTER TABLE KPP ADD join_date DATETIME</td><td>db.KPP.updateMany(
  { },
  { $set: { join_date: new Date() } }
)</td></tr>
<tr><td>ALTER TABLE KPP DROP COLUMN join_date</td><td>db.KPP.updateMany(
  { },
  { $unset: { "join_date": "" } }
)</td></tr>
<tr><td>DROP TABLE KPP</td><td>db.KPP.drop ()</td></tr>
</table>

Insert Statement	

INSERT INTO KPP (user_id, age, status)
VALUES ("mongo", 45, "A")
	db.KPP.insertOne(
	{ user_id: "mongo", age: 18, status: "A" })

Select Command	
SELECT *FROM KPP	db.KPP.find()
SELECT id, user_id, status FROM KPP	db.KPP.find({ }, { user_id: 1, status: 1 })
SELECT user_id, status FROM KPP	db.KPP.find({ }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM KPP WHERE status = "B"	db.KPP.find({ status: "A" })
SELECT user_id, status FROM KPP WHERE status = "A"	db.KPP.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })
SELECT * FROM KPP WHERE status != "A"	db.KPP.find({ status: { $ne: "A" } })

SELECT *
FROM KPP
WHERE status = "A"
AND age = 50
	db.KPP.find(
	{ status: "A",
	age: 50 }
	)

SELECT *
FROM KPP
WHERE status = "A"
OR age = 50
	db.KPP.find(
	{ $or: [{ status: "A" } , { age: 50 }] }
	)

SELECT *
FROM KPP
WHERE age > 25
	db.KPP.find(
	{ age: { $gt: 25 } }
	)

SELECT *
FROM KPP
WHERE age < 25
	Db.KPP.find(
	{ age: { $lt: 25 } }
	)

SELECT *
FROM KPP
WHERE age > 25
AND age <= 50
	db.KPP.find(
	{ age: { $gt: 25, $lte: 50 } }
	)

SELECT *
FROM KPP
WHERE user_id like "%bc%"
	db.KPP.find({ user_id: /bc/ })
	-or-
	
	db.KPP.find({ user_id: { $regex: /bc/ } })

SELECT *
FROM KPP
WHERE user_id like "bc%"
	db.KPP.find({ user_id: /^bc/ })
	-or-
	db.KPP.find({ user_id: { $regex: /^bc/ } })

SELECT * from KPP where NAME=”abc” and AGE:20	db.student.find({name:"abc",age:20})

SELECT *
FROM KPP
WHERE status = "A"
ORDER BY user_id ASC
	db. KPP. find({ status: "A" }). sort({ user_id: 1 })

SELECT *
FROM KPP
WHERE status = "A"
ORDER BY user_id DESC
	db. KPP. find({ status: "A" }). sort({ user_id: -1 })

SELECT COUNT(*)	db. KPP. count()

FROM KPP	or db. KPP. find(). count()

SELECT COUNT(user_id)
FROM KPP
	db. KPP.count({ user_id: { $exists: true } })
	or
	db. KPP.find({ user_id: { $exists: true } }
	).count()

SELECT COUNT(*)
FROM KPP
WHERE age > 30
	db. KPP.count({ age: { $gt: 30 } })
	or
	db. KPP.find({ age: { $gt: 30 } }).count()

SELECT *
FROM KPP
LIMIT 1
	db. KPP.findOne()
	or
	db. KPP.find(). limit(1)

SELECT *
FROM KPP
LIMIT 5
SKIP 10
	db. KPP.find(). limit(5). skip(10)

EXPLAIN SELECT * FROM KPP WHERE status = "A"	db. KPP. find({ status: "A" }).explain()
Update Statements	

UPDATE KPP SET status = "C"
WHERE age > 25
	db.KPP.updateMany({ age: { $gt: 25 } }, { $set: {
	status: "C" } })

UPDATE KPP SET age = age + 3
WHERE status = "A"
	db.KPP.updateMany({ status: "A" } , { $inc: {
	age: 3 }

Delete Statements	
DELETE FROM KPP WHERE status = "D"	db.KPP.deleteMany({ status: "D" })
DELETE FROM KPP	db.KPP.deleteMany({ })

Task: Create a collection named student having fields name, age, standard, percentage. Insert 5 to 10 random records in table.

1)find name of all students having age>5
2)update the standard for all students by 1.
3)arrange all the records in descending order of age.
4)show the name of student who is the oldest student among all students.
5)delete the record of the student if standard is 12.

```
db.student.insertOne({name:"abc",age:13,standard:6,perc:80})
db.student.insertOne({name:"def",age:15,standard:8,perc:90})
db.student.insertOne({name:"ghi",age:10,standard:3,perc:75})
db.student.insertOne({name:"pqr",age:5,standard:1,perc:89})
db.student.insertOne({name:"xyz",age:17,standard:12,perc:97})

db.student.find({age:{$gt:5}})


db.student.find().sort({age:-1})


db.student.find().sort({age:-1}).limit(1)


db.student.updateMany({},{$inc:{standard:1}})


db.student.deleteOne({standard:12})
```

Regex

$regex : Provides regular expression capabilities for pattern matching strings in queries.

To use $regex, use one of the following syntax:

```
{ <field>: { $regex: /pattern/} }
```

case 1: Find all names where "shah" occurs as a substring or as a separate word.

```
db.KPP.find({name:{$regex:/shah/}})
```

or

```
db.KPP.find({name:{$regex:"shah"}})
```

case 2: To have a case-insensitive matching we can append /i indentifier after RE

```
db.KPP.find({name:{$regex:/shah/i}})
```

case 3: To match a string beginning with "shah" only.

```
db.KPP.find({name:{$regex:/^shah/}})
```

case 4: To end with "shah" only.

```
db.KPP.find({name:{$regex:/shah$/}})
```

case 5: To end with digit only.

```
db.KPP.find({name:{$regex:/[0-9]$/}})
```

case 6: To start with digit only.

```
db.KPP.find({name:{$regex:/^[0-9]/}})
```

or

```
db.KPP.find({name:{$regex:/^\d/}})
```

case 7: To accept only digits, nothing else. Not even blank.

```
db.KPP.find({name:{$regex:/^[0-9]+$/}})
```

case 8: To accept only with empty string also.

```
db.KPP.find({name:{$regex:/^[0-9]*$/}})
```

case 9: To match having a name of 3-10 letters only.

```
db.KPP.find({name:{$regex:/^[A-Za-z]{3,10}$/}})
```

If we include \w instead of this [A-Za-z], then it may allow digits & underscore also.

Note: all patterns can be matched with string only. If there is one field age and we have inserted all int values then RegEx can not be compared with it.