

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: <u>MySQL</u> , <u>PostgreSQL</u> , Oracle, MS-SQL Server, etc	Examples: <u>MongoDB</u> , <u>GraphQL</u> , <u>HBase</u> , <u>Neo4j</u> , <u>Cassandra</u> , 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 <code>_id</code> 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:

- ❖ **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.
- ❖ **Data integration.** This integrates data for applications, including for hybrid and multi-cloud applications.
- ❖ **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.
- ❖ **Load balancing.** MongoDB can be used to run over multiple servers.

Features of MongoDB

Features of MongoDB include the following:

- ❖ **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.

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

- ❖ **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.

Above Scalability feature of mongoDB is explained in detail.

Scaling

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:

- **Vertical Scaling:**

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. Now when the existing system fails to meet the expected needs, and the expected needs can be met by just adding resources, this is considered vertical scaling. 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.

- **Horizontal Scaling:**

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:

- ✓ **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.
- ✓ **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.
- ✓ **Scalability.** A core function of MongoDB is its horizontal scalability, which makes it a useful database for companies running big data applications. In addition, sharding lets the database distribute data across a cluster of machines. MongoDB also supports the creation of zones of data based on a shard key.
- ✓ **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.

- ✓ **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.
- ✓ **Data consistency.** MongoDB doesn't provide full referential integrity through the use of foreign-key constraints, which could affect data consistency.
- ✓ **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
 - |-Community Edition
 - |-Download Community
 - |-Download Mongo dB Community server (8.0.9)
 - Select Version, Platform, Package(msi- **Microsoft Software Installer**) and Download it

Version	8.0.9 (current)	▼
Platform	Windows x64	▼
Package	msi	▼

Download ⬇

 Copy link

More Options ...

- ✓ Run and install downloaded version

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> (version - 2.5.2)

Version	2.5.2	▼
Platform	Windows x64 (10+)	▼
Package	zip	▼

[Download](#)   Copy link [More Options](#) 

- ✓ Download zip
- ✓ Extract files and from **bin** folder **Copy exe** and **dll** files to “C:\Program Files\MongoDB\Server\8.0\bin”

Step 3:

Set path property in environment variable (Under **User variables temporary** if you don't have administrative rights)

- ✓ Select “System Variable” > Path
- ✓ Click on “Edit”
- ✓ Add “C:\Program Files\MongoDB\Server\8.0\bin”

Step 4: Verify MongoDB Installation

Check MongoDB Server Version

To confirm MongoDB is installed and accessible via command line:

Open **Command Prompt**, type:

```
mongod --version
```

This will display the version of the **MongoDB server** (mongod) installed.

Start the MongoDB Shell

To start the MongoDB Shell (mongosh):

In **Command Prompt**, type:

```
mongosh
```

This connects to the **default MongoDB instance** running on: **localhost:27017**

What is mongosh?

The mongosh command starts the **MongoDB Shell**, a command-line interface for interacting with your MongoDB server. It allows you to:

- Perform administrative tasks
- Create and manage databases and collections
- Insert, query, and update documents

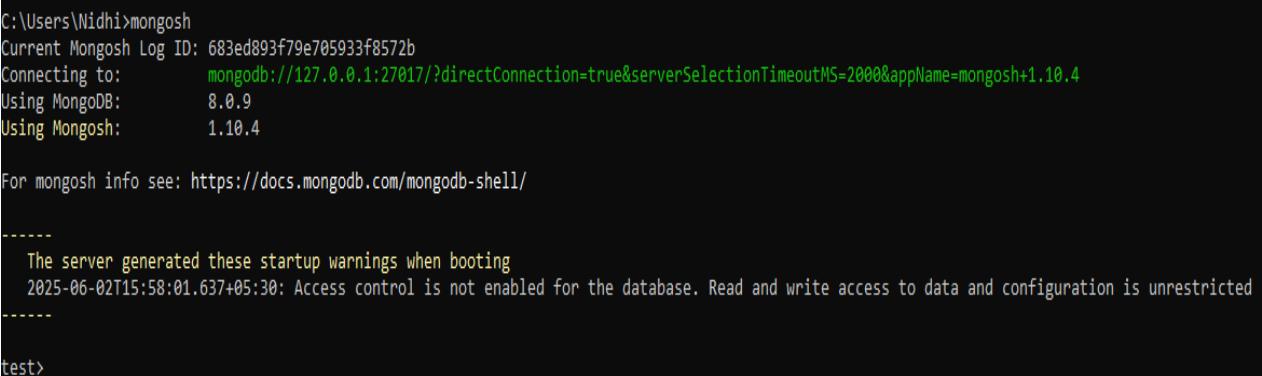
- Run MongoDB commands and scripts

Connect to a Specific MongoDB Instance:

mongosh "mongodb://hostname:port"

mongosh "mongodb://192.168.1.100:27018"

It will shown as below screenshot.



```
C:\Users\Widhi>mongosh
Current Mongosh Log ID: 683ed893f79e705933f8572b
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1.10.4
Using MongoDB:      8.0.9
Using Mongosh:       1.10.4

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

-----
The server generated these startup warnings when booting
2025-06-02T15:58:01.637+05:30: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
-----

test>
```

- ✓ **Default database**
 - **Test>**

What is MongoDB Compass?

MongoDB Compass is a graphical user interface (GUI) tool provided by MongoDB for interacting with MongoDB databases. It is designed to provide a user-friendly way to visualize, manipulate, and manage your data without needing to use the command line.

To download

- ✓ Go to <https://www.mongodb.com/try/download/shell>
- ✓ And download **MongoDB Compass Download (GUI) (1.46.5 stable)**

Commands

Show Databases:	show dbs
Switch to a Database:	use databaseName
Create a Database (Switching to a non-existent database creates it):	use newDatabase
Drop current database:	db.dropDatabase()
Show Collections:	show collections
Create a Collection:	db.createCollection("collectionName")
Drop a Collection:	db.collectionName.drop()
Insert a Document:	db.collectionName.insertOne({ key: "value" })
Insert Documents:	db.collectionName.insertMany([{ key: "value" }, { key: "value" }])
Find All Documents in a Collection:	db.collectionName.find()
Find Documents with a Query:	db.collectionName.find({ key: "value" })
Find Documents' specific field with a Query:	db.collectionName.find({ key: "value" }, {key :true})
Update a Document:	db.collectionName.update({ key: "value" }, { \$set: { key: "new_value" } })
Update a Document and if document not available then insert a document:	db.collectionName.update({ key: "value" }, { \$set: { key: "new_value" } }, {upsert:true})
Delete a first Document:	db.collectionName.deleteOne({})
Delete all Documents:	db.collectionName.deleteMany({})
Delete a Document with a Query:	db.collectionName.deleteOne({key: "value"})
Delete Documents with a Query:	db.collectionName.deleteMany({key: "value"})
Show Users:	show users

✓ To list all the available DBs write below command

- **test> show dbs**
 - admin 40.00 KiB
 - config 72.00 KiB
 - local 76.00 KiB

✓ To show current active database.

- **test>db**

✓ Display all available collections in existing database

- **test>show collections**
- ✓ To create/use database
- **test> use mydb**
 - **mydb>**

db.createCollection()

MongoDB db.createCollection(name, options) is used to create collection.

Basic syntax of createCollection() command is as follows –

db.createCollection(name, options)

In the command, name is name of collection to be created. The options parameter allows you to specify various configuration options when creating a new collection.

Example:

db.createCollection("student")

The drop() Method

For Collection

MongoDB's db.collection.drop() is used to drop a collection from the database. Basic syntax of drop() command is as follows –

db.COLLECTION_NAME.drop()

Example

First, check the available collections into your database mydb.

```
>use mydb
switched to db mydb
>show collections
mycollection
user
student
```

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

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

Again check the list of collections into database.

```
>show collections
user
student
```

For Database

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

```
{ ok: 1, dropped: 'mydb' }
```

This will drop a database named “mydb”

renameCollection()

Call the **db.collection.renameCollection()** method on a collection object, to rename a collection. Specify the new name of the collection as an argument.

Example:

```
db.student.renameCollection("students")
```

This renames “**student**” collection to “**students**”.

Insert documents

✓ insertOne()

To create collection with only one document

- **mydb>db.student.insertOne({name:"Test",rollno:23})**
 - In above example student is collection(table).
 - Two fields(columns) are “name” and “rollno”
 - 1st document(row) data are “Test” and “23” for fields “name” and “rollno” respectively.

✓ insertMany()

To create collection with more than one documents

- **mydb>db.student.insertMany([{name:"N1",age:20},{name:"N2",age:24}])**

Output:

```
{ _id: ObjectId("64dfd8ffc925fb0136d77817"), name: 'N1', age: 20 },  
{ _id: ObjectId("64dfd8ffc925fb0136d77818"), name: 'N2', age: 24 }
```

✓ We can also store non-uniform document fields as shown in below example.

- **mydb>db.student.insertMany([
 {name:"N1",age:20,status:"Active"},
 {name:"N2",age:24},
 {name:"N3",age:27,status:"Active",city:"Ahmedabad"}
])**

So, here documents contain 3,2 4 fields respectively.

Find documents

✓ find()

Find document/Read document

- Mydb>db.student.find()
 - Display all documents of **student** collection

Syntax

db.student.find (query/filter/condition, projection)

Query/filter/condition : Optional. Specifies selection filter using query operators. To return all documents in a collection, omit this parameter or pass an empty document

Projection :Optional. Specifies the fields to return in the documents that match the query filter. To return all fields in the matching documents, omit this parameter.

Example: **only query**

o mydb> **db.student.find({name:"N1"})**

- As an output it will give all the documents with name N1 and also display all the fields as shown below.

Output :

```
[ { _id: ObjectId("64de65b454282c021d807835"), name: 'N1', age: 20 } ]
```

Example: **query and projection**

• mydb> **db.student.find({name:"N1"},{_id:0,age:false })**

- As an output it will give all the documents with name N1 and displays only **name** field as shown below

Output:

```
[ { name: 'N1' } ]
```

Note: To display field write “true” or 1

To not display field write “false” or 0

If you use 1 (or true) for a field in the projection document, it means that field will be included in the output.

If you use 0 (or false) for a field, it means that field will not be included in the output.

Here's how it works:

Including Specific Fields:

If you specify a field with 1 or true, only that field and the _id field (by default) will be included in the result.

Example: { name: 1, age: 1 } will include only the name, age, and _id fields in the output.

Excluding Specific Fields:

If you specify a field with 0 or false, that field will be excluded from the result, and all other fields will be included (except `_id`, which is included by default unless explicitly excluded).
Example: { name: 0, age: 0 } will exclude the name and age fields and include all other fields.

Excluding the `_id` Field:

You can explicitly exclude the `_id` field by setting it to 0 or false.

Example: { `_id`: 0, name: 1 } will include only the name field and exclude the `_id` field.

Mutually Exclusive Inclusion and Exclusion:

You cannot mix inclusion (1 or true) and exclusion (0 or false) in the same projection document, except for the `_id` field.

Example: { name: 1, age: 0 } is invalid, but { name: 1, `_id`: 0 } is valid.

✓ `findOne()`

- ✓ `findOne()` method returns only a document that satisfies the criteria entered or first document of the collection.
- ✓ 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.
 - **`Db.student.findOne()`** - Since no filter or query condition is provided, MongoDB will return the **first document** it finds in the collection, according to the natural order (which is usually the order in which documents were inserted).
 - **`Db.student.findOne({name:"N1"})`** - MongoDB searches the collection for a document that matches the specified filter criteria, {name: "N1"}. If such a document exists, it returns the first match it finds.

✓ `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.
- For example, N1 name exists in 4 documents and we want to display limited number of documents then we can use **limit** method
- Suppose, we have written 1 in limit then it will display only 1st document of 4 documents.
 - **`Db.student.find({name:"N1"}).limit(1)`**
Or we can use `findOne()` to display only 1st document
 - **`Db.student.findOne({name:"N1"})`**

Displays only first document of collection

✓ `skip()`

- Call the `skip()` method on a cursor to control where MongoDB begins returning results. This approach may be useful in implementing paginated results.
- Suppose, we have 3 documents with name "N1". Below command will skip 1st document and give only 2nd document as an output.

- **`db.student.find({name:"N1"}).limit(1).skip(1)`**

Output:

```
[ { _id: ObjectId("64de6dbb54282c021d807837"), name: 'N1', age: 23 } ]
```

Update documents

✓ `updateOne()`

```
db.collection.updateOne(filter,document,options)
```

Updates only one document where filter is matched

✓ `updateMany()`

```
db.collection.updateMany(filter, document, options)
```

Updates 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** etc.

```
db.student.updateOne(
  {name:"N1"},
  {$set:
    {
      name:"N4"
    }
  })
```

Updates only one document with **name N4** where **name is N1**

- `db.student.updateOne({name:"N1",age:23},{ $set:{name:"N4"}})`
 - Updates only one document with name N4 where name is N1 and age is 23
- `db.student.updateOne({name:"N1",age:23},{ $set:{name:"N4",age:11}})`
 - Updates only one document with name N4 and age 11 where name is N1 and age is 23
- `db.student.updateMany({name:"N1"},{$set:{name:"N4"}})`
 - Updates all documents with name N4 and where name is N1

✓ **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**).

Syntax: upsert: <boolean>

- ✓ The value of upsert option is either true or false. By default it is **false**.

`collection.update_one(filter, update, upsert)`

Key Points:

- **Filter:** Specifies which document to look for.
- **Update:** Specifies the update operation to perform.
- **Upsert:** If set to `true`, MongoDB will insert a new document if no matching document is found.

Upsert operations are efficient for ensuring a document with specific criteria exists, either by updating an existing document or inserting a new one if none exists.

- ✓ 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.

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

In above example it will find document with age 45 in collection. If condition matched then update the name else insert new document with mentioned fields.

Note: Update will only update the values if the document is already exist with mentioned filter value.

If document does not exist in collection with mentioned filter value and we want to add document with the mentioned fields then we have to add 3rd parameter `upsert:true`.

Delete Documents

deleteOne() and deleteMany()

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

✓ deleteOne()

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

- **db.student.deleteOne({name:"N1"})**
 - This deletes only 1st document where name is "N1"

✓ 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:"N1"})**
 - This deletes all the documents where name is "N1"

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 has the following parameter:
- ✓ The db.collection.count() method is equivalent to the db.collection.find(query).count() construct.

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

It will give an answer with below warning.

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

Or

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

As an output it will give 3 if uname "N1" exists in 3 documents

Or

db.people.countDocuments({uname:"N1"});

As an output it will give 3 if uname "N1" exists in 3 documents

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:

```
sort({ field: value })
```

Example:

To display all documents in descending order of age.

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

Ascending: 1 and Descending: -1

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: { \$in: [<value1>, <value2>, ... <valueN>] } } db.people.find({ age: { \$in: [25, 50] } })
\$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({ qty: { \$nin: [5, 15] } })

- ✓ **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}})`
- ✓ **Display all documents where age is greater than 25**
 - `db.people.find({age:{ $gt:25}})`
- ✓ **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 } }]})`
- ✓ **Display all documents where age is equal to 70**
 - `db.people.find({age:{ $eq:70}})`
- ✓ **Display all documents where age is not equal to 70**
 - `db.people.find({age:{ $ne:70}})`
- ✓ **Display all the documents where people are aged 45 or 70**
 - `db.people.find({age:{ $in:[45,70]}})`
- ✓ **Display all the documents where people are not aged 45 or 70.**
 - `db.people.find({age:{ $nin:[45,70]}})`

Logical operators

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:"N1"},{age:28}]})`
or
`db.student.find({name:"N1",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:"N1"},{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.student.find({ age: { \$not: { \$lt: 20 } } })

The \$not operator in MongoDB is used to negate the query criteria. It can be combined with other operators to match documents where the specified condition is **not true**. Here are some examples to illustrate the use of the \$not operator:

1. Basic Example with \$not and \$eq (Equal)

- **Query:** Find all documents in the student collection where the age is **not equal** to 18.
- **Code:**
`db.student.find({ age: { $not: { $eq: 18 } } })`
- **Explanation:** This query matches all documents where the age field is not equal to 18.

2. Example with \$not and \$lt (Less Than)

- **Query:** Find all documents in the student collection where the age is **not less** than 20.
- **Code:**
`db.student.find({ age: { $not: { $lt: 20 } } })`
- **Explanation:** This query returns all documents where the age field is 20 or more.

3. Example with \$not and \$regex

- **Query:** Find all documents in the student collection where the name does **not** start with the letter "J".
- **Code:**
`db.student.find({ name: { $not: { $regex: "^J" } } })`
- **Explanation:** This query matches all documents where the name field does not start with the letter "J". The ^ in the regex pattern indicates the start of the string.

4. Example with \$not and \$in

- **Query:** Find all documents in the student collection where the grade is **not** in the set ["A", "B"].
- **Code:**
`db.student.find({ grade: { $not: { $in: ["A", "B"] } } })`
- **Explanation:** This query matches all documents where the grade field is neither "A" nor "B".

✓ \$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.student.find({$nor: [{name: "N1"}]})`
 - `db.student.find({$nor: [{name: "N1"},{age:20}]})`
 - It will show documents which do not have name "N1" or age "20"

Field Update operators

Name	Description
<u>\$currentDate</u>	Sets the value of a field to current date, either as a Date or a Timestamp.
<u>\$inc</u>	Increments the value of the field by the specified amount.
<u>\$mul</u>	Multiplies the value of the field by the specified amount.
<u>\$rename</u>	Renames a field.
<u>\$set</u>	Sets the value of a field in a document.
<u>\$unset</u>	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'}})
```

✓ \$currentDate

The \$currentDate operator in MongoDB is used in update operations to set the value of a field to the current date or timestamp. It is often used to automatically update fields with the current date and time whenever a document is modified.

Example 1: Setting a Field to the Current Date

- **Scenario:** You have a users collection, and you want to update the lastModified field to the current date whenever a user's document is updated.

Query:

```
db.users.update(
  { name: "ABC" },
  { $currentDate: { lastModified: { $type: "date" } } }
)
```

- **Explanation:**
 - This query finds the document where the name is "ABC" and sets the lastModified field to the current date (Date type).
 - The \$type: "date" option is optional. If you don't specify it, MongoDB defaults to using a Date type.

Example 2: Setting a Field to the Current Timestamp

- **Scenario:** You have an orders collection, and you want to set the lastUpdated field to the current timestamp whenever an order is updated.

Query:

```
db.orders.update(
  { orderId: 12345 },
  { $currentDate: { lastUpdated: { $type: "timestamp" } } }
)
```

- **Explanation:**
 - This query finds the document where the orderId is 12345 and sets the lastUpdated field to the current timestamp (Timestamp type).
 - The Timestamp type is different from Date in that it is a special internal MongoDB type that includes an incrementing ordinal value in addition to the time.

When you see a Timestamp with { t: 1723223315, i: 1 }, it represents two components:

▪ t (Time Component):

Description: The t represents the number of seconds since the Unix epoch (January 1, 1970, 00:00:00 UTC).

Example Value: In Timestamp({ t: 1723223315, i: 1 }), the value 1723223315 corresponds to the number of seconds.

- **i (Increment Component):**

Description: The i represents an incrementing ordinal value that differentiates multiple operations occurring within the same second. Example Value: In Timestamp({ t: 1723223315, i: 1 }), the value 1 indicates that this is the first operation within that particular second. If another operation occurred in the same second, its i value might be 2

Example 3: Using \$currentDate with Other Update Operators

- **Scenario:** You want to increment a field and update the lastModified field in a single update operation.

Query:

```
db.inventory.update(
  { item: "apple" },
  {
    $inc: { quantity: 10 },
    $currentDate: { lastModified: true } // equivalent to { $type: "date" }
  }
)
```

- **Explanation:**
 - This query finds the document where the item is "apple", increments the quantity by 10, and sets the lastModified field to the current date.

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, this 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.lju.find({age:30})
> rec
```

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 “test” occurs as a substring or as a separate word.

```
db.lju.find({name:{ $regex:/test/}})
```

or

```
db.lju.find({name:{ $regex:"test"}})
```

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

```
db.lju.find({name:{ $regex:/test/i}})
```

This will return documents if name is stored in collection as “Test”, “tEST”, “teST”, “Testing” etc.

case 3: To match a string beginning with “test” only.

```
db.lju.find({name:{ $regex:/^test/}})
```

case 4: To end with “test” only.

```
db.lju.find({name:{ $regex:/test$/}})
```

case 5: To end with digit only.

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

case 6: To start with digit only.

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

or

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

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

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

case 8: To accept only with empty string also.

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

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

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

Letters only is mentioned in question so we have added ^ and \$.

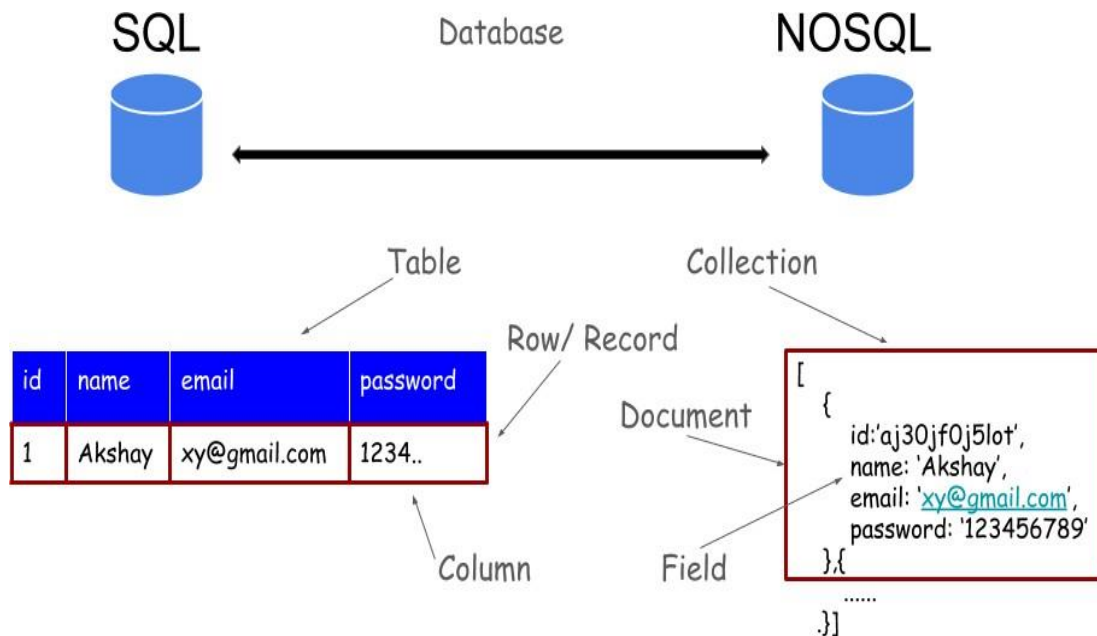
Note:

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

If it is asked in question that it **allowed only letters** then use [A-Za-z]

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.

SQL to MongoDB Mapping



SQL	MongoDB
<pre>CREATE TABLE lju (id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(20),age Number, status char(1), PRIMARY KEY (id))</pre>	<pre>db.createCollection ("lju")</pre> <p>creates collection no need to define schema</p>
<pre>ALTER TABLE lju ADD join_date DATETIME</pre>	<pre>db.lju.updateMany({}, { \$set: { join_date: new Date() } })</pre> <p>Note: new Date().toLocaleDateString()-The toLocaleDateString() method returns the date (not the</p>

	time) of a date object as a string, using locale conventions.
ALTER TABLE lju DROP COLUMN join_date	db.lju.updateMany({ }, { \$unset: { "join_date": "" } })
DROP TABLE lju	db.lju.drop()
INSERT INTO lju (name, age, status) VALUES ("Test", 45, "A")	db.lju.insertOne({ name: "Test", age: 18, status: "A" })
SELECT *FROM lju	db.lju.find()
SELECT id, name FROM lju	db.lju.find({ }, { name:1 })
SELECT name, status FROM lju	db.lju.find({ }, { name:1,status:1,_id: 0 })
SELECT * FROM lju WHERE status ="A"	db.lju.find({ status: "A" })
SELECT name FROM lju WHERE status = "A"	db.lju.find({ status: "A" }, { name: 1,_id: 0 })
SELECT * FROM lju WHERE status != "A"	db.lju.find({ status: { \$ne: "A" } })
SELECT * FROM lju WHERE status = "A" AND age = 50	db.lju.find({status:"A",age:50}) or db.lju.find({ \$and: [{ status: "A" }, { age: 50 }] })
SELECT * FROM lju WHERE status = "A" OR age = 30	db.lju.find({ \$or: [{ status: "A" }, { age: 30 }] })
SELECT * FROM lju WHERE age > 25	db.lju.find({ age: { \$gt: 25 } })
SELECT * FROM lju WHERE age <= 25	db.lju.find({ age: { \$lte: 25 } })

SELECT * FROM lju WHERE age > 25 AND age <= 50	db.lju.find({ age: { \$gt: 25, \$lte: 50 } })
SELECT * FROM lju WHERE name like "%abc%"	db.lju.find({ name: { \$regex: /abc/ } })
SELECT * FROM lju WHERE name like "abc%"	db.lju.find({ name: /^bc/ }) or db.lju.find({ name: { \$regex: /^abc/ } })
SELECT * from lju where name="abc" and age=20	db.lju.find({name:"abc",age:20})
SELECT * FROM lju WHERE status = "A" ORDER BY age ASC	db.lju.find({ status: "A" }).sort({ age: 1 })
SELECT * FROM lju WHERE status = "A" ORDER BY age DESC	db.lju.find({ status: "A" }). sort({ age: -1 })
SELECT COUNT(*) FROM lju	db.lju. count() or db. lju. find(). count()
SELECT COUNT(*) FROM lju WHERE age > 30	db.lju.count({ age: { \$gt: 30 } }) or db.lju.find({ age: { \$gt: 30 } }).count()
SELECT * FROM lju LIMIT 1	db.lju.findOne() or db.lju.find().limit(1)
SELECT * FROM lju LIMIT 2 OFFSET 3;	db.lju.find().limit(2).skip(3)
EXPLAIN SELECT * FROM lju WHERE status = "A"	db.lju. find({ status: "A" }).explain()
UPDATE lju SET status = "C" WHERE age > 25	db.lju.updateMany({ age: { \$gt: 25 } }, { \$set: { status: "C" } })
UPDATE lju SET age = age + 3 WHERE status = "A"	db.lju.updateMany({ status: "A" }, { \$inc: { age: 3 } }
DELETE FROM lju WHERE status = "D"	db.lju.deleteMany({ status: "D" })
DELETE FROM lju	db.lju.deleteMany({ })

SELECT * FROM lju WHERE name NOT IN ('aaa','abc','bbb');	db.lju.find({ name: { \$nin: ['aaa','abc','bbb'] } })
SELECT name FROM lju WHERE age IN (20,23,33);	db.lju.find({ age: { \$in: [20, 23, 33] } }, {name:1,_id:0})

Examples

Example1

Create a collection named student having fields name, age, standard, percentage. Insert 5 to 10 random documents in collection.

- 1) Find name of all students having age>5
- 2) Increase 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.insertMany([{"name":"abc",age:13,standard:6,perc:80}, {"name":"def",age:15,standard:8,perc:90}, {"name":"ghi",age:10,standard:3,perc:75}, {"name":"pqr",age:5,standard:1,perc:89}, {"name":"xyz",age:17,standard:12,perc:97}])
```

- 1) db.student.find({age:{\$gt:5}},{name:1,_id:0})
- 2) db.student.updateMany({},{\$inc:{standard:1}})
- 3) db.student.find().sort({age:-1})
- 4) db.student.find({}, {name:1,_id:0}).sort({age:-1}).limit(1)
- 5) db.student.deleteOne({standard:12})

Example2

Perform the tasks as asked below.

Create Collection “employees” with following data

```
[{_id: 1,name: "Eric",age: 30,position: "Full Stack Developer",salary: 60000},
{_id: 2,name: "Erica",age: 35,position: "Intern",salary: 8000},
{_id: 3,name: "Erica",age: 40,position: "UX/UI Designer",salary: 56000},
{_id: 4,name: "Eric",age: 37,position: "Team Leader",salary: 85000},
{_id: 5,name: "Eliza",age: 25,position: "Software Developer",salary: 45000},
{_id: 6,name: "Trian",age: 29,position: "Data Scientist",salary: 75000},
{_id: 7,name: "Elizan",age: 25,position: "Full Stack Developer",salary: 49000}]
```

- 1) Find All Documents
- 2) Find Documents by Position “Full Stack Developer”:
- 3) Retrieve name of employees whose age is greater than or equal to 25 and less than or equal to 40.
- 4) Retrieve name of the employee with the highest salary.

- 5) Retrieve employees with a salary greater than 50000.
- 6) Count the number of employees who have salary greater than 50000
- 7) Retrieve employees who are either " **Software Developer**" or "**Full Stack Developer**" and are below 30 years.
- 8) Increase the salary of an employee who has salary less than 50000 by 10%.
- 9) Delete all employees who are older than 50.
- 10) Give a 5% salary raise to all "**Data Scientist**"
- 11) Find documents where name like "%an"
- 12) Find documents where name like "Eri--" (Case Insensitive)
- 13) Find documents where name like "%ric%"
- 14) Find documents where name contains only 4 or 5 letters.
- 15) Find documents where name must end with digit

Answers:

```
db.employees.insertMany([{_id: 1,name: "Eric",age: 30,position: "Full Stack Developer",salary: 60000},
{_id: 2,name: "Erica",age: 35,position: "Intern",salary: 8000},
{_id: 3,name: "Erical",age: 40,position: "UX/UI Designer",salary: 56000},
{_id: 4,name: "treric7",age: 37,position: "Team Leader",salary: 85000},
{_id: 5,name: "Eliza",age: 25,position: "Software Developer",salary: 45000},
{_id: 6,name: "Trian",age: 29,position: "Data Scientist",salary: 75000},
{_id: 7,name: "Elizan",age: 25,position: "Full Stack Developer",salary: 49000}])
```

- 1) db.employees.find()
- 2) db.employees.find({position:"Full Stack Developer"})
- 3) db.employees.find({ age: { \$gte: 30, \$lte: 40 } },{name:1,_id:0})
- 4) db.employees.find({}, {name:1, _id:0}).sort({ salary: -1 }).limit(1)
- 5) db.employees.find({ salary: { \$gt: 50000 } })
- 6) db.employees.find({salary:{\$gt:50000}}).count()
- 7) db.employees.find({ \$and: [{ \$or: [{ position: "Software Developer" }, { position: "Full Stack Developer" }] }, { age: { \$lt: 30 } }] })
- 8) db.employees.updateOne({salary:{\$lt:50000}},{ \$mul: { salary: 1.1 } })
- 9) db.employees.deleteMany({ age: { \$gt: 50 } })
- 10) db.employees.updateMany({ position: "Data Scientist" }, { \$mul: { salary: 1.05 } })
- 11) db.employees.find({name:{\$regex:/an\$/}})
- 12) db.employees.find({name:{\$regex:/^eri[A-z]{2}\$/i}})
- 13) db.employees.find({name:{\$regex:/ric/i}})

14) db.employees.find({name:{\$regex:/^[A-Za-z]{4,5}\$/i}})

15) db.employees.find({name:{\$regex:/[0-9]\$/}})

Example3

Insert 10 documents with random data with fields _id,brand,price,cat as shown below.

```
db.product.insertMany([
  {_id:1,brand:"samsung",price:29000,cat:"mobile"},
  {_id:2,brand:"nokia",price:5000,cat:"mobile"},
  {_id:3,brand:"vivo",price:16000,cat:"mobile"},
  {_id:4,brand:"samsung",price:60000,cat:"tv"},
  {_id:5,brand:"samsung",price:40000,cat:"washing machine"},
  {_id:6,brand:"ifb",price:45000,cat:"wasing machine"},
  {_id:7,brand:"apple",price:120000,cat:"mobile"},
  {_id:8,brand:"oppo",price:20000,cat:"mobile"},
  {_id:9,brand:"sony",price:80000,cat:"tv"},
  {_id:10,brand:"vivo",price:31000,cat:"mobile"},
])
```

- 1) Display price and brand of product which are of mobile cat.**
- 2) Increase price of each Samsung products by 1000.**
- 3) Update all vivo product by adding field quantity and add random value**
- 4) Display price of products which are of vivo or oppo brand.**
- 5) Display brand and cat of products which are less than 80000 and greater than or equal to 30000.**

Answers:

- 1) db.product.find({cat:"mobile"},{cat:0,_id:0})
- 2) db.product.updateMany({brand:"samsung"},{\$inc:{price:1000}})
- 3) db.product.updateMany({brand:"vivo"},{\$set:{quantity:5}})
- 4) db.product.find({\$or:[{brand:"vivo"},{brand:"oppo"}]},{price:1,_id:0})
- 5) db.product.find({price:{\$lt:80000,\$gte:30000}},{price:0,_id:0})

Example4

Consider following student collection:

```
[  
  {_id:123433,name: "SSS",age:22},  
  {_id:123434,name: "YYY",age:2},  
  {_id:123435,name: "PPP",age:32},  
]
```

Do as directed:

- (1) Update name="JJJ" and age=40, where age=20 occurs. Insert new document, if record is not found.
- (2) To retrieve age and name fields of documents having names "YYY" & "SSS". Don't project _id field.

Answers:

- 1) db.info.updateMany({age:20},{ \$set:{name:'JJJ',age:40}}, {upsert:true})
- 2) db.info.find({ \$or: [{ name: 'YYY' }, { name: 'SSS' }] }, { _id: 0 })