MONGO DB

Pallavi Mahalatkar

INTRODUCTION

MongoDB is a leading open-source NoSQL database known for its high performance, flexibility, and ease of scalability. Unlike traditional relational databases that use tables and rows, MongoDB stores data in dynamic, JSON-like documents, which allows for more versatile data models and structures. This document-oriented storage model makes MongoDB ideal for handling large volumes of unstructured or semi-structured data, such as data generated by modern web and mobile applications.

The database was developed by MongoDB Inc., originally released in 2009, and has since grown to become one of the most widely used NoSQL databases in the world. Its ability to store data in a format that closely resembles objects in application code makes it very popular among developers, simplifying data integration and manipulation.

One of the standout features of MongoDB is its flexible schema. Unlike relational databases that require a fixed schema, MongoDB allows for schema evolution, meaning the structure of the documents can change over time. This flexibility can significantly speed up development cycles, as developers can modify the data model without major disruptions.

Security is a critical aspect of MongoDB, which offers features like authentication, authorization, encryption, and auditing to protect sensitive data. It also supports role-based access control (RBAC) to fine-tune user permissions and ensure secure access to the database.

In summary, MongoDB's combination of a flexible data model, powerful querying and aggregation capabilities, robust scalability, and comprehensive security features make it a compelling choice for modern application development. Its ability to efficiently handle diverse and evolving data types positions it as a versatile solution for a wide range of use cases, from simple web applications to complex, data-intensive systems.

Why Mongo DB ?

MongoDB is a leading open-source NoSQL database known for its high performance, flexibility, and ease of scalability. Unlike traditional relational databases that use tables and rows, MongoDB stores data in dynamic, JSON-like documents, which allows for more versatile data models and structures. This document-oriented storage model makes MongoDB ideal for handling large volumes of unstructured or semi-structured data, such as data generated by modern web and mobile applications.

Where mongo DB is used ?

- Handling Large Volumes of Data
- **Scalability:**
- Flexible Schema Design
- High Performance
- Real-Time Analytics
- Developer Productivity
- Cloud Integration
- Big Data Applications
- Content Management Systems
- **Internet of Things (IoT)**
- E-Commerce
- Mobile Applications

In summary, MongoDB is used in scenarios where data flexibility, scalability, high performance, and developer productivity are critical. Its versatile nature allows it to be applied across various industries and use cases, making it a popular choice for modern application development.

How to Download Mongo DB ?

Downloading and installing MongoDB involves several steps, depending on your operating system. Below are the instructions for downloading and installing MongoDB on Windows, macOS, and Linux.

For Windows:

1. **Visit the MongoDB Download Center:**
   - Go to the MongoDB Download Center.
2. **Select the MongoDB Version:**
   - Choose the "Community" server.
   - Select your version, platform (Windows), and package (MSI).
3. **Download the Installer:**
   - Click the "Download" button to download the MSI installer.
4. **Run the Installer:**
   - Double-click the downloaded MSI file to run the installer.
   - Follow the prompts in the setup wizard. You can use the default settings, but make sure to select the option to install MongoDB as a service.
5. **Configure MongoDB:**
   - During the installation, you might be prompted to choose a setup type. Select "Complete."
   - Optionally, install MongoDB Compass, a GUI for MongoDB, if prompted.
6. **Verify the Installation:**
   - Open Command Prompt and run mongod --version to verify that MongoDB is installed correctly.

Common Steps After Installation:

1. **Check MongoDB Status:**
   - For Windows: Use the Services application to check if MongoDB service is running.
   - For macOS and Linux: Run sudo systemctl status mongod to check the status of MongoDB.
2. **Access MongoDB Shell:**
   - Open a new terminal or command prompt window and run:
   - This command starts the MongoDB shell, where you can interact with your database.
3. **Configuration and Data Directory:**
   - MongoDB stores data in /data/db by default. Ensure this directory exists and is writable. You can configure different paths in the mongod.conf file.

By following these steps, you can successfully download, install, and start using MongoDB on your system.

THIS PICTORIAL EXAMPLES MAKE US UNDERSTAND SQL

SQL is a standard language for accessing and manipulating databases. It is used for querying, updating, and managing data stored in relational database management systems (RDBMS). SQL is widely used because it provides a systematic way to create, retrieve and update

Key Components of SQL:

1. **DDL (Data Definition Language):**

   Commands that define the structure of the database.

   - CREATE: Creates new tables, databases, indexes, etc.
   - ALTER: Modifies existing database objects.
   - DROP: Deletes database objects.

2. **DML (Data Manipulation Language):**

   Commands that manipulate data in existing tables.

   - SELECT: Retrieves data from one or more tables.
   - INSERT: Adds new rows to a table.
   - UPDATE: Modifies existing data within a table.
   - DELETE: Removes rows from a table.

3. **DCL (Data Control Language):**

   Commands that control access to data.

   - GRANT: Gives user access privileges to the database.
   - REVOKE: Removes user access privileges.

4. **TCL (Transaction Control Language):**

   Commands that manage transactions.

   - COMMIT: Saves all changes made during the current transaction.
   - ROLLBACK: Undoes changes made during the current transaction.
   - SAVEPOINT: Sets a point within a transaction to which you can later roll back.

Basic SQL Examples

1. Creating a Table :

```
CREATE TABLE employees (
   id INT PRIMARY KEY,
   name VARCHAR(100),
   position VARCHAR(50),
   salary DECIMAL(10, 2),
   hire_date DATE
);
```

2. Inserting Data into a Table :

```
INSERT INTO employees (id, name, position, salary, hire_date)
VALUES (1, 'John Doe', 'Software Engineer', 60000.00, '2021-03-15');
```

3. Querying Data from a Table:

```
SELECT name, position, salary
FROM employees
WHERE salary > 50000;
```

4. Updating Data in a Table

```
sql
Copy code
UPDATE employees
SET salary = 65000.00
WHERE id = 1;
```

5. Deleting Data from a Table

```
DELETE FROM employees
WHERE id = 1;
```

6. Adding a Column to an Existing Table

```
ALTER TABLE employees
ADD COLUMN department VARCHAR(50);
```

7. Dropping a Table

```
DROP TABLE employees;
```

Summary

SQL is a powerful language used for managing and manipulating relational databases. By understanding and using SQL commands, you can effectively interact with and manage data stored in a database.

1. SQL Schema and Table Structure

Schema Diagram:

```
diff
+----------------------+
|      DATABASE        |
+----------------------+
|       employees      |
+----------------------+
```

Table Structure:

```
-------------------------------------------------------+
|                       employees                       |
+-----------+-------------+-----------+--------+-----------+
| id (INT)  | name (VARCHAR)| position  | salary | hire_date|
| PRIMARY KEY |  (100)    | (50)      | DECIMAL|   (DATE)  |
+-----------+-------------+-----------+--------+-----------+
|     1     | John Doe    | Software  | 60000  | 2021-03-15|
|           |             | Engineer  |        |           |
+-----------+-------------+-----------+--------+-----------+
```

2. SQL Query Flow

SELECT Query Flow:

mermaid

```
graph TD;
    A[Client] -->|Sends SQL Query| B[Database Server];
    B -->|Processes Query| C[Database Engine];
    C -->|Retrieves Data| D[Database Storage];
    D --> C;
    C -->|Returns Data| B;
    B -->|Sends Result| A;
```

3. SQL CRUD Operations

CRUD Operations:

```
----------------------+
|       CREATE       |
+----------------------+
| CREATE TABLE        |
| INSERT INTO         |
+----------------------+

+----------------------+
|       READ         |
+----------------------+
| SELECT FROM         |
+----------------------+

+----------------------+
|       UPDATE       |
+----------------------+
| UPDATE SET          |
+----------------------+

+----------------------+
|        DELETE        |
+----------------------+
| DELETE FROM          |
+----------------------+
```

4. Entity-Relationship Diagram (ERD)

Example ERD:

```
+-----------------------+            +----------------------+
|       Employee        |            |      Department      |
+-----------------------+            +----------------------+
| id (PK)               |<---------->| id (PK)              |
| name                  |            | name                 |
| position              |            +----------------------+
| salary                |
| hire_date             |
| department_id (FK)    |
+-----------------------+
```

5. Data Manipulation Example

Example Table Data:

```
diff
employees
+----+----------+----------------+--------+------------+
| id |   name   |    position    | salary | hire_date  |
+----+----------+----------------+--------+------------+
| 1  | John Doe | Software Eng.  | 60000  | 2021-03-15 |
| 2  | Jane Smith | Data Analyst | 55000  | 2022-01-10 |
+----+----------+----------------+--------+------------+
```

Example Query:

SELECT name, position, salary FROM employees WHERE salary > 50000;

Query Result:

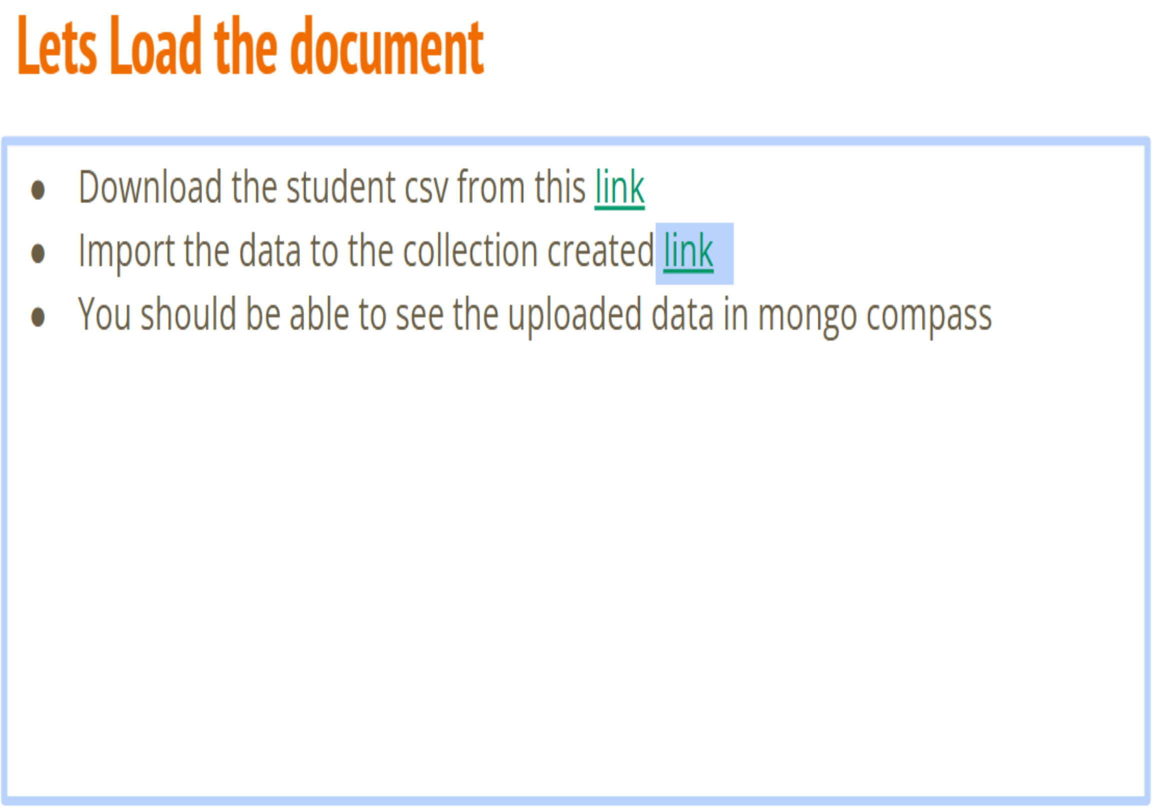
```
diff
+----------+----------------+--------+
|   name   |    position    | salary |
+----------+----------------+--------+
| John Doe | Software Eng.  | 60000  |
| Jane Smith | Data Analyst | 55000  |
+----------+----------------+--------+
```

This some of the examples and key points will help us in the coding and by the above key points

Helps us to use the correct form of query while coding https://drive.google.com/file/d/1L3FhU59Tw-a-o-

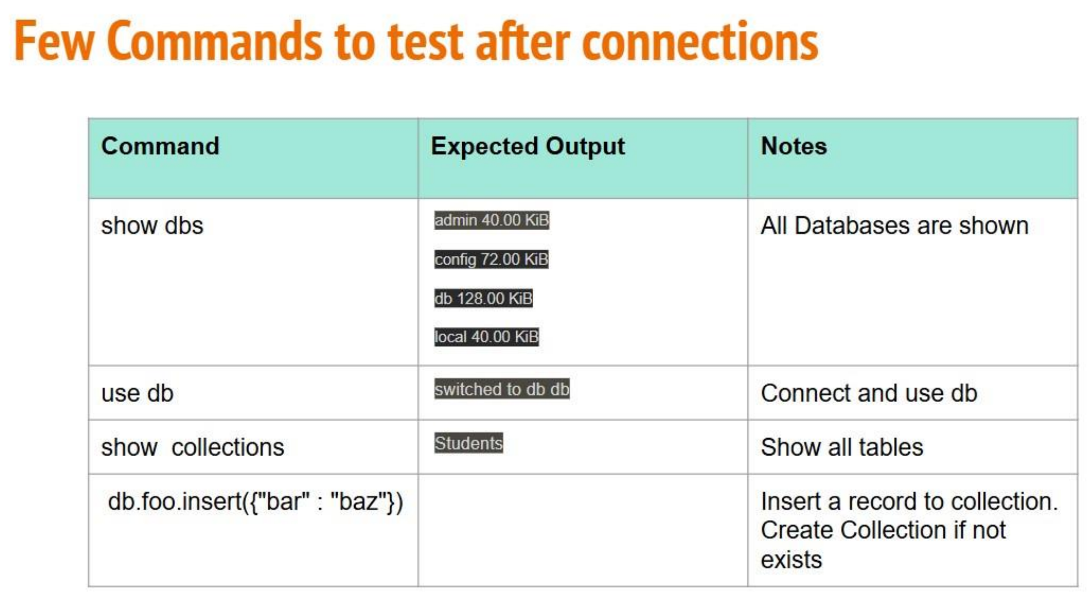
- link

By using the above links create a database in mongo db compass and save the csv file with data base name db and collection name student.



Lets Load the document

- Download the student csv from this link
- Import the data to the collection created link
- You should be able to see the uploaded data in mongo compass



Few Commands to test after connections

Command	Expected Output	Notes
show dbs	admin 40.00 KiB config 72.00 KiB db 128.00 KiB local 40.00 KiB	All Databases are shown
use db	switched to db db	Connect and use db
show collections	Students	Show all tables
db.foo.insert({"bar" : "baz"})		Insert a record to collection. Create Collection if not exists

Explanation :

1.db>show dbs

This command lists all the databases present on the MongoDb server.

Output :

- admin 40.00 KiB
- config 108.00 KiB
- db 56.00 KiB
- local 72.00 KiB

Each line shows the name of a database and its size .for example,

The ‘db’ database.

2.db>use db:

Output:

already on db db

This message indicates that the shell is already using the DB database. If it wasn't, the output would have been switched to db db

3 . **db> show collections**:

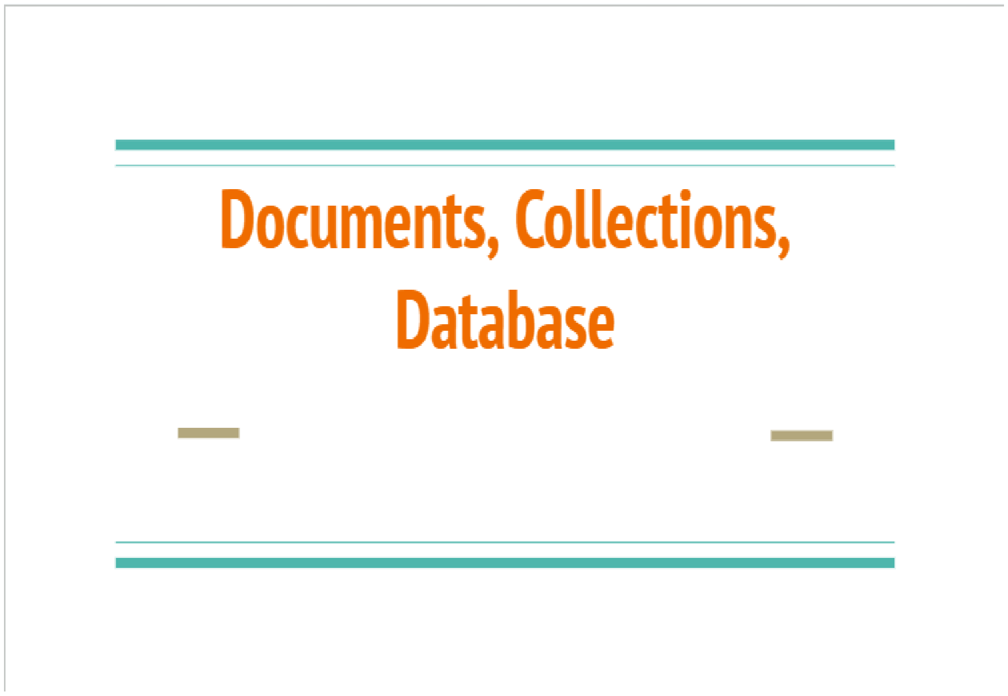
- This command lists all collections in the current database (db).
- **Output**: student
- This indicates that there is one collection named student in the db database.

4 . **db> db.foo.insert({"bar" : "baz"})**:

- This command inserts a document with the content {"bar" : "baz"} into the foo collection of the db database.
- **Deprecation Warning**:
  - Deprecation Warning: Collection. insert() is deprecated. Use insert One, insert Many, or bulk Write.
  - This warning informs you that the insert() method is deprecated and suggests using insert One, insert Many, or bulk Write instead.

Output:

- acknowledged: true
  - This indicates that the insert operation was acknowledged by the database.
- insertedIds: { '0': ObjectId('6658acbf6fd215de8acdcdf6') }
  - This shows the IDs of the inserted documents. Since only one document was inserted, there's only one ID: ObjectId('6658acbf6fd215de8acdcdf6')



Documents, Collections, Database

Documents :

In MongoDB, a document is a basic unit of data. It's a record in a collection, similar to a row in a table in a relational database. Documents are stored in a flexible, JSON-like format called BSON (Binary JSON), which allows for the storage of complex data types and hierarchical relationships.

Example of a MongoDB Document:

Consider a collection named users which stores user information. A document in this collection might look like this:

```
{
  "_id": ObjectId("60d5ecb5a7c2a5a3b3a8e93c"),
  "name": "John Doe",
  "email": "johndoe@example.com",
  "age": 29,
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  },
  "interests": ["reading", "traveling", "swimming"]
}
```

Output :

```
"acknowledged": true,
 "insertedId": ObjectId("60d5ecb5a7c2a5a3b3a8e93c")
```

Breakdown of the Example Document:

- **_id:** This is a unique identifier for the document, automatically generated by MongoDB if not provided.
- **name, email, age:** These are fields with simple data types (string and number).
- **address:** This is an embedded document (a document within a document), containing the user's address details.
- **interests:** This is an array of strings, representing the user's interests

CRUD Operations on Documents:

1. **Create:** Insert a new document into a collection.

```
db.users.insertOne({
  "name": "John Doe",
  "email": "johndoe@example.com",
  "age": 29,
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zip": "12345"
  },
  "interests": ["reading", "traveling", "swimming"]
});
```

Output :

```
"acknowledged": true,
"insertedId": ObjectId("60d5ecb5a7c2a5a3b3a8e93c")
```

Explination :

- acknowledged": true indicates that the operation was acknowledged by the MongoDB server, meaning it was successfully completed.
- "insertedId": ObjectId("60d5ecb5a7c2a5a3b3a8e93c") is the unique identifier (ID) assigned to the document that was inserted.

2. **Read:** Query documents from a collection.

```
db.users.find({ "name": "John Doe" });
```

Output :

```
{
"_id": ObjectId("60d5ecb5a7c2a5a3b3a8e93c"),
"name": "John Doe",
"email": "johndoe@example.com",
"age": 29,
 "address": { "street": "123 Main St",
 "city": "Anytown",
"state": "CA",
"zip": "12345"
 },
 "interests": ["reading", "traveling", "swimming"]
 }
```

Explanation :

The query db.users.find({ "name": "John Doe" }); retrieves a user document with the name "John Doe" from the users collection, including fields such as email, age, address, and interests

3. **Update:** Modify an existing document.

```
db.users.updateOne(
  { "name": "John Doe" },
  { $set: { "age": 30 } }
);
```

Output :

```
{
 "acknowledged": true,
"matchedCount": 0,
 "modifiedCount": 0
}
```

Explanation :

The update query db.users.updateOne({ "name": "John Doe" }, { $set: { "age": 30 } }); attempts to set John Doe's age to 30. The output indicates that while the operation was acknowledged, no documents matched the criteria or were modified.

4. **Delete:** Remove a document from a collection.

```
db.users.deleteOne({ "name": "John Doe" });
```

output :

```
{
"acknowledged": true,
 "deletedCount": 0
}
```

Explanation:

The output of the delete operation db.users.deleteOne({ "name": "John Doe" }); shows that while the operation was acknowledged, no documents matching the criteria were found or deleted.

5. **Delete:** Remove a document from a collection.

```
db.users.deleteOne({ "name": "John Doe" });
```

output :

```
{
"acknowledged": true,
 "deletedCount": 0
}
```

Explanation :

The code db.users.deleteOne({ "name": "John Doe" }); attempts to delete a document from the users collection where the name is "John Doe." The output indicates that the operation was acknowledged but no matching document was found to delete, resulting in a deletedCount of 0.

MongoDB documents provide a flexible and powerful way to store and manipulate data, making it well-suited for applications with evolving data requirements.

Collections :

In MongoDB, a collection is a grouping of MongoDB documents. Collections are akin to tables in relational databases, but they do not enforce a schema, meaning documents within a collection can have different fields and structures. Collections store documents in BSON (Binary JSON) format

Key Features of MongoDB Collections:

1. **Schema-less:** Collections do not require a predefined schema. Each document can have a different structure.
2. **Dynamic:** Collections can grow dynamically as more documents are inserted.
3. **Indexing:** Collections can be indexed to improve query performance.
4. **Flexibility:** Collections can store a wide variety of data types and complex structures, including arrays and nested documents.

Example of Creating and Using a Collection:

Let's create a collection named users and insert some documents into it.

1. **Creating a Collection and Inserting Documents:** While MongoDB implicitly creates a collection when a document is inserted, you can also explicitly create a collection.

```
// Implicitly creating a collection by inserting a document
 db.users.insertOne({
 "name": "Alice",
"email": "alice@example.com",
 "age": 25,
 "address": {
"street": "456 Elm St",
"city": "Othertown",
 "state": "NY",
"zip": "67890"
 },
 "interests": ["painting", "cycling"]
});
// Explicitly creating a collection db.createCollection("users");
```

Output :

```
{
"acknowledged": true,
"insertedId": ObjectId("some_unique_id")
 }
```

Explanation :

The first command implicitly creates the `users` collection by inserting a document with details about Alice. The output confirms the insertion with an acknowledgment and provides the unique `insertedId` of the new document.

2. Inserting Multiple Documents :

```
db.users.insertMany([
 {
"name": "Bob",
"email": "bob@example.com",
 "age": 30,
"address": {
"street": "789 Maple St",
"city": "Sometown",
 "state": "TX",
"zip": "12345"
 },
 "interests": ["gaming", "hiking"]
 },
 {
"name": "Charlie",
"email": "charlie@example.com",
 "age": 28,
 "address": {
 "street": "101 Pine St"
, "city": "Anytown",
"state": "CA",
"zip": "67890"
 },
"interests": ["cooking", "swimming"]
}
 ]);
```

Output :

```
{
  "acknowledged": true,
  "insertedIds": {
    "0": ObjectId("some_unique_id1"),
    "1": ObjectId("some_unique_id2")
  }
}
```

Explanation:

The code inserts multiple documents for Bob and Charlie into the users collection. The output confirms the operation was successful and provides unique IDs for the newly inserted document

3.Querying Documents in a Collection :

```
// Find all documents in the users collection
db.users.find();
 // Find documents where the name is "Bob"
 db.users.find({ "name": "Bob" });
```

output :

```
[
  {
    "_id": ObjectId("some_unique_id2"),
    "name": "Bob",
    "email": "bob@example.com",
    "age": 30,
    "address": {
      "street": "789 Maple St",
      "city": "Sometown",
      "state": "TX",
      "zip": "12345"
    },
    "interests": ["gaming", "hiking"]
  }
]
[
  {
    "_id": ObjectId("some_unique_id1"),
  "name": "Alice",
  "email": "alice@example.com",
  "age": 25,
  "address": {
    "street": "456 Elm St",
    "city": "Othertown",
    "state": "NY",
    "zip": "67890"
  },
  "interests": ["painting", "cycling"]
},
{
  "_id": ObjectId("some_unique_id2"),
  "name": "Bob",
  "email": "bob@example.com",
  "age": 30,
  "address": {
    "street": "789 Maple St",
    "city": "Sometown",
    "state": "TX",
    "zip": "12345"
  },
  "interests": ["gaming", "hiking"]
},
{
  "_id": ObjectId("some_unique_id3"),
  "name": "Charlie",
  "email": "charlie@example.com",
  "age": 28,
    "address": {
      "street": "101 Pine St",
      "city": "Anytown",
      "state": "CA",
      "zip": "67890"
    },
    "interests": ["cooking", "swimming"]
  }
]
```

Explanation:

The first query retrieves all documents from the users collection, displaying Alice, Bob, and Charlie's details. The second query retrieves only the document where the name is "Bob," displaying Bob's details.

4.Updating Documents in a Collection:

```
// Update a document where the name is "Alice"
 db.users.updateOne(
{ "name": "Alice" },
{ $set: { "age": 26 } }
 );
```

Output :

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

Explanation:

The code updates the document where the name is "Alice," setting her age to 26. The output confirms the option was acknowledged, with one document matched and modified.

5.Deleting Documents in a Collection:

```
// Delete a document where the name is "Charlie"
 db.users.deleteOne({ "name": "Charlie" });
```

Output :

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

Explanation:

The code deletes the document where the name is "Charlie" from the users collection. The output confirms that the operation was acknowledged, and one document was successfully deleted.

Summary :

MongoDB collections provide a flexible and powerful way to store and manage diverse and dynamic datasets, making them suitable for a wide range of applications.

Data Base :

a database is a container for collections of documents. It serves as the first level of organization in the database hierarchy. Databases are identified by names. These names are case-sensitive and can include characters, numbers, and certain special characters (like _ and -). However, they cannot contain spaces or be an empty string.

Operations on Databases

- **Creating a Database**: Databases are created implicitly in MongoDB. A database is created when you first store data in it.

```
use myDatabase
```

This command switches to the myDatabase database; if it doesn't exist, it will be created upon the first insertion of data.

Dropping a Database: You can delete a database using the dropDatabase command.

```
db.dropDatabase();
```

Listing Databases: You can list all databases in a MongoDB instance with the show dbs command.

```
show dbs;
```

Insert a Document into a Collection:

```
db.exampleCollection.insertOne({ name: "John Doe", age: 30, profession: "Software Engineer" });
```

output :

```
{
  "acknowledged": true,
  "insertedId": ObjectId("60c6d2f5b4d7a8b7f79e7b1b")
}
```

Explanation :

acknowledged: true: This field indicates that the MongoDB server has acknowledged the receipt and execution of the insert operation. It confirms that the document was successfully inserted into the collection.

insertedId: ObjectId("60c6d2f5b4d7a8b7f79e7b1b"): This field contains the unique identifier (_id) assigned to the newly inserted document

Query the Collection:

```
db.exampleCollection.find({ name: "John Doe" });
```

output :

```
{
  "_id": ObjectId("60c6d2f5b4d7a8b7f79e7b1b"),
  "name": "John Doe",
  "age": 30,
  "profession": "Software Engineer"
}
```

Drop the Collection:

```
db.dropDatabase();
```

Output :

```
{
  "dropped": "exampleDB",
  "ok": 1
}
```

Explanation :

1. **dropped: "exampleDB"**: This field indicates the name of the database that was dropped. In this example, the database exampleDB was the one currently in use and was successfully deleted.
2. **ok: 1**: This field indicates the success of the operation. In MongoDB, an ok value of 1 means that the operation was successful. If the ok value were 0, it would indicate a failure.

. Advantages of MongoDB Databases

- Schema Flexibility: No rigid schema makes it easy to evolve data models.
- Scalability: MongoDB supports horizontal scaling through sharding.
- Rich Query Language: Supports a rich and expressive query language.
- Geospatial Indexing: Provides support for geospatial queries and indexes.

Summary

A MongoDB database is a high-level container for organizing collections of documents. Each database can hold a large number of collections, which in turn hold documents with dynamic schemas, providing flexibility and ease of use for developers. This structure makes MongoDB particularly well-suited for applications requiring scalability and diverse data models.

Where, AND, OR & CRUD

In MongoDB, the where, and, and or operators are used for querying documents in a collection based on specified conditions. Here's a brief explanation of each, along with examples and explanations of the output.

1. $where

The $where operator allows you to specify JavaScript expressions for more complex queries. It is generally less efficient than other operators.

Example:

```
db.collection.find({
  $where: function() {
    return this.age > 25 && this.age < 35;
  }
}
```

Explanation :

This query finds all documents where the age field is greater than 25 and less than 35. The JavaScript function executes for each document, which can be slower than using built-in operators.

2. $and :

The $and operator is used to join query clauses with a logical AND. This operator requires an array of one or more query expressions.

Example:

```
db.collection.find({
  $and: [
    { age: { $gt: 25 } },
    { age: { $lt: 35 } }
  ]
})
```

Explanation

This query returns all documents where the age field is greater than 25 and less than 35. It's more efficient than $where for simple logical conditions.

3. $or :

The $or operator is used to join query clauses with a logical OR. This operator requires an array of one or more query expressions.

```
db.collection.find({
  $or: [
    { age: { $lt: 25 } },
    { age: { $gt: 35 } }
  ]
})
```

Explanation

This query returns all documents where the age field is either less than 25 or greater than 35.

Some of the examples of when,where,and OR

1.$ where Example :

```
db.people.find({
  $where: function() {
    return this.age > 25 && this.age < 35;
  }
})
```

Output:

```
[
  { "_id": 1, "name": "Alice", "age": 28 },
  { "_id": 2, "name": "Bob", "age": 32 }
]
```

Explanation: This query finds Alice and Bob because their ages are between 25 and 35.

$and Example :

```
db.people.find({
       2.$and:
          [
    { age: { $gt: 25 } },
    { age: { $lt: 35 } }
  ]
})
```

Output :

```
[
  { "_id": 1, "name": "Alice", "age": 28 },
  { "_id": 2, "name": "Bob", "age": 32 }
]
```

 Explanation: This query also finds Alice and Bob for the same reason as above.

$or Example :

```
db.people.find({
```

```
 3. $or: [
    { age: { $lt: 25 } },
    { age: { $gt: 35 } }
  ]
})
```

Output :

```
[
  { "_id": 3, "name": "Charlie", "age": 22 },
  { "_id": 4, "name": "Dave", "age": 40 }
```

]

Explanation: This query finds Charlie and Dave because Charlie's age is less than 25 and Dave's age is greater than 35.

- $where : is flexible but less efficient for complex conditions.
- $and: is used for logical AND conditions and is efficient for straightforward queries.
- $or :is used for logical OR conditions and is also efficient for straightforward queries.

4.Curd :

CRUD operations in MongoDB refer to the basic operations you can perform on a database: Create, Read, Update, and Delete. These operations are fundamental for working with any database, and MongoDB, a NoSQL database, provides easy-to-use commands for each of these operations.

1. Create

The Create operation in MongoDB is used to insert documents into a collection. The insertOne method is used to insert a single document, and the insertMany method is used to insert multiple documents.

Example: Insert one document

```
db.students.insertOne({

   name: "John Doe",

   age: 22,

   subjects: ["Math", "Science"]

})
```

Output:

```
{
   "acknowledged" : true,
   "insertedId" : ObjectId("60d5ec3f8f634cf7f3f5c8b7")
}
```

Example: Insert multiple documents

```
db.students.insertMany([
   { name: "Jane Smith", age: 23, subjects: ["English", "History"] },
   { name: "Sam Brown", age: 21, subjects: ["Art", "Math"] }
])
```

Output:

```
{
   "acknowledged" : true,
   "insertedIds" : [
      ObjectId("60d5ec3f8f634cf7f3f5c8b8"),
      ObjectId("60d5ec3f8f634cf7f3f5c8b9")
   ]
}
```

2. Read

The Read operation is used to query documents in a collection. The find method retrieves documents that match the given query criteria.

Example: Find all documents

```
db.students.find({})
```

output:

```
[
   { "_id": ObjectId("60d5ec3f8f634cf7f3f5c8b7"), "name": "John Doe", "age": 22, "subjects": ["Math", "Science"] },
   { "_id": ObjectId("60d5ec3f8f634cf7f3f5c8b8"), "name": "Jane Smith", "age": 23, "subjects": ["English", "History"] },
   { "_id": ObjectId("60d5ec3f8f634cf7f3f5c8b9"), "name": "Sam Brown", "age": 21, "subjects": ["Art", "Math"] }
]
```

Example: Find documents with a specific condition

```
[
   { "_id": ObjectId("60d5ec3f8f634cf7f3f5c8b7"), "name": "John Doe", "age": 22, "subjects": ["Math", "Science"] }
]
```

3. Update

The Update operation modifies existing documents. The updateOne method updates a single document, and the updateMany method updates multiple documents that match the criteria.

Example: Update one document

```
db.students.updateOne(
   { name: "John Doe" },
   { $set: { age: 23 } }
)
```

Output:

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

Example: Update multiple documents

```
db.students.updateMany(
   { age: { $gt: 21 } },
   { $set: { status: "senior" } }
)
```

Output :

```
{
    "acknowledged" : true,
    "matchedCount" : 2,
    "modifiedCount" : 2
}
```

4. Delete

The Delete operation removes documents from a collection. The deleteOne method deletes a single document, and the deleteMany method deletes multiple documents that match the criteria.

Example: Delete one document

```
db.students.deleteOne({ name: "John Doe" })
```

output:

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

Example: Delete multiple documents

```
db.students.deleteMany({ age: { $lt: 23 } })
```

Output:

```
{
   "acknowledged" : true,

   "deletedCount" : 1

}
```

Explanation of the Code and Output

1. **Create**:
    - insertOne and insertMany methods are used to add new documents to the collection. The output contains the acknowledged status and the IDs of the inserted documents.
2. **Read**:
    - find method is used to retrieve documents. An empty query {} returns all documents, while specific criteria can be passed to filter the results. The output is a list of documents that match the query.
3. **Update**:
    - updateOne and updateMany methods are used to modify existing documents. The output indicates the number of documents matched and modified.
4. **Delete**:
    - deleteOne and deleteMany methods are used to remove documents from the collection. The output shows the count of documents deleted.

These examples demonstrate how to perform basic CRUD operations in MongoDB using the Mongo Shell or a MongoDB client library like PyMongo. The operations are straightforward and provide the necessary functionality to manage data within a MongoDB database.

Selectors :

In MongoDB, selectors are used to query documents in a collection. They define criteria that the documents must meet to be selected. Selectors are expressed as JSON-like documents, and they use various operators to specify the conditions. Here's a comprehensive explanation of selectors in MongoDB, along with examples and their outputs.

Basic Selectors

1.Equality Selector ({ field: value }): Select documents where the value of a field equals the specified value.

```
db.collection.find({ age: 25 })
```

output:

```
[
  { "_id": 1, "name": "Alice", "age": 25 },
  { "_id": 3, "name": "Charlie", "age": 25 }
]
```

Comparison Selectors

2. **Greater Than ($gt)**: Select documents where the value of a field is greater than the specified value.

```
db.collection.find({ age: { $gt: 30 } })
```

output :

```
[
  { "_id": 2, "name": "Bob", "age": 35 },
  { "_id": 4, "name": "David", "age": 40 }
]
```

Less Than ($lt):

Select documents where the value of a field is less than the specified value.

```
db.collection.find({ age: { $lt: 30 } })
```

```
output :
[
  { "_id": 1, "name": "Alice", "age": 25 },
  { "_id": 3, "name": "Charlie", "age": 25 }
]
```

4.Greater Than or Equal ($gte): Select documents where the value of a field is greater than or equal to the specified value

```
db.collection.find({ age: { $gte: 30 } })
output :
[
  { "_id": 2, "name": "Bob", "age": 35 },
  { "_id": 4, "name": "David", "age": 40 }
]
```

5.Less Than or Equal ($lte): Select documents where the value of a field is less than or equal to the specified value.

```
db.collection.find({ age: { $lte: 30 } })
output :
[
  { "_id": 1, "name": "Alice", "age": 25 },
  { "_id": 3, "name": "Charlie", "age": 25 }
]
```

6.Not Equal ($ne): Select documents where the value of a field is not equal to the specified value.

```
db.collection.find({ age: { $ne: 25 } })
output :
[
  { "_id": 2, "name": "Bob", "age": 35 },
  { "_id": 4, "name": "David", "age": 40 }
]
```

Logical Selectors :

7.AND ($and): Combine multiple conditions that must all be met.

```
db.collection.find({ $and: [ { age: { $gt: 25 } }, { age: { $lt: 40 } } ] })
```

output :

```
[
  { "_id": 2, "name": "Bob", "age": 35 }
]
```

8.OR ($or): Combine multiple conditions where at least one must be met.

```
db.collection.find({ $or: [ { age: { $lt: 30 } }, { age: { $gt: 35 } } ] })
```

output :

```
[
  { "_id": 1, "name": "Alice", "age": 25 },
  { "_id": 3, "name": "Charlie", "age": 25 },
  { "_id": 4, "name": "David", "age": 40 }
]
```

9.NOT ($not): Select documents where the value of a field does not match the specified condition.

```
db.collection.find({ age: { $not: { $gt: 30 } } })
```

output :

```
[
  { "_id": 1, "name": "Alice", "age": 25 },
  { "_id": 3, "name": "Charlie", "age": 25 }
```

]

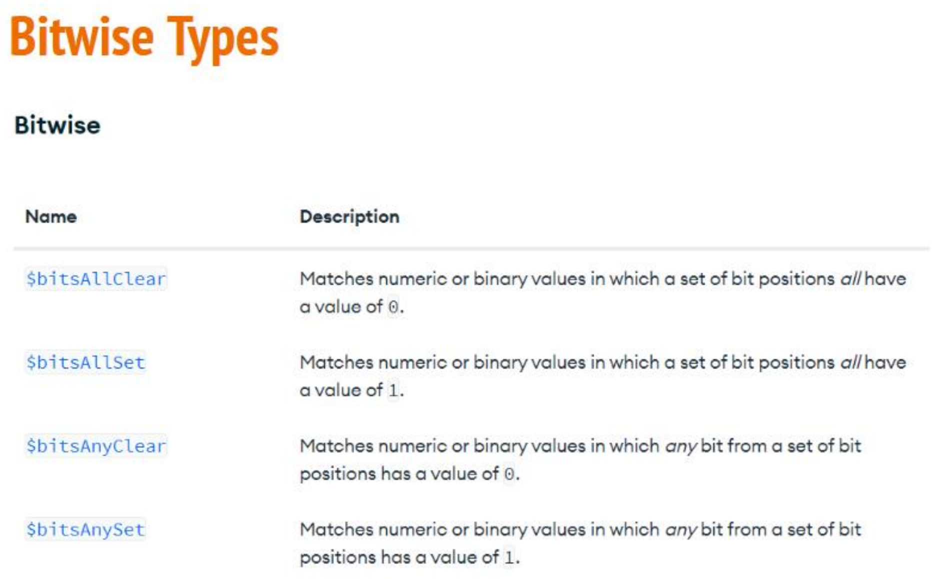
10.NOR ($nor): Combine multiple conditions where none must be met.

```
db.collection.find({ $nor: [ { age: { $gt: 30 } }, { name: "Alice" } ] })
```

output :

```
[
  { "_id": 3, "name": "Charlie", "age": 25 }
]
```

Bitwise types :



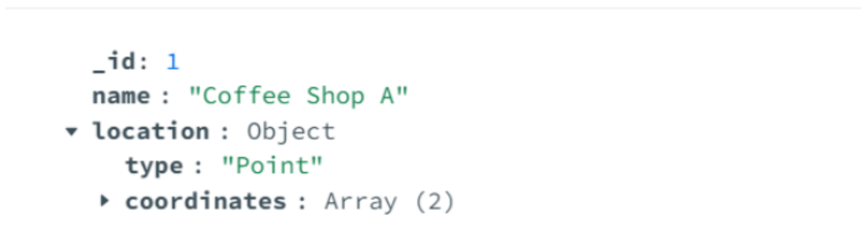
Bitwise Types

Bitwise

Name	Description
`$bitsAllClear`	Matches numeric or binary values in which a set of bit positions *all* have a value of `0`.
`$bitsAllSet`	Matches numeric or binary values in which a set of bit positions *all* have a value of `1`.
`$bitsAnyClear`	Matches numeric or binary values in which *any* bit from a set of bit positions has a value of `0`.
`$bitsAnySet`	Matches numeric or binary values in which *any* bit from a set of bit positions has a value of `1`.

Geospatial

- Official Documentation link
- Create collection called “locations”
- Upload the dataset using json link



```
  _id: 1
  name : "Coffee Shop A"
▾ location : Object
    type : "Point"
  ▸ coordinates : Array (2)
```

Official Documentation link

- Upload the dataset using json link
- Geospatial queries in MongoDB enable you to perform location-based searches on your data. MongoDB supports various geospatial queries and indexes that allow you to store and query data based on geographical locations.

  Creating Geospatial Indexes

2d Index

Use a 2d index for flat, two-dimensional geometry.

db.places.createIndex({ location: "2d" })

2dsphere Index

Use a 2dsphere index for spherical geometry.

db.places.createIndex({ location: "2dsphere" })

Example Data

Let's assume we have a collection called places with documents that include a location field storing GeoJSON Point objects.

```
db.places.insertMany([

  { name: "Central Park", location: { type: "Point", coordinates: [-73.9712, 40.7831] } },

  { name: "Empire State Building", location: { type: "Point", coordinates: [-73.9857, 40.7488] }
},

  { name: "Statue of Liberty", location: { type: "Point", coordinates: [-74.0445, 40.6892] } }

])
```

Geospatial Queries

Finding Places Near a Point

Use the $near operator to find places near a specific point

```
db.places.find({

  location: {

    $near: {

      $geometry: {

        type: "Point",

        coordinates: [-73.9857, 40.7488]

      },

      $maxDistance: 5000 // in meters

    }

  }

})
```

Output :

```
[

  { "_id": ObjectId("..."), "name": "Empire State Building", "location": { "type": "Point",
"coordinates": [-73.9857, 40.7488] } },

  { "_id": ObjectId("..."), "name": "Central Park", "location": { "type": "Point", "coordinates": [-
73.9712, 40.7831] } }

]
```

Explanation :

- **$near Query Output**: The query finds places near the specified coordinates within a maximum distance of 5000 meters. The output includes "Empire State Building" and "Central Park" because they are within 5000 meters of the given point.

- **$geoWithin Polygon Query Output**: The query finds places within the defined polygon. The output includes "Statue of Liberty" because it falls within the specified polygon coordinates.

Conclusion :

MongoDB, as a NoSQL database, offers significant flexibility and scalability compared to traditional relational databases. Its document-oriented structure, which stores data in BSON format, allows for a more natural representation of complex data models, making it ideal for applications requiring rapid development and iteration. MongoDB's schema-less design ensures adaptability to evolving data needs without requiring extensive schema modifications. Features such as horizontal scaling, sharding, and replication enhance its capability to handle large volumes of data and ensure high availability and fault tolerance. As organizations increasingly deal with diverse and unstructured data, MongoDB's robust and versatile nature positions it as a powerful tool in modern data management strategies.