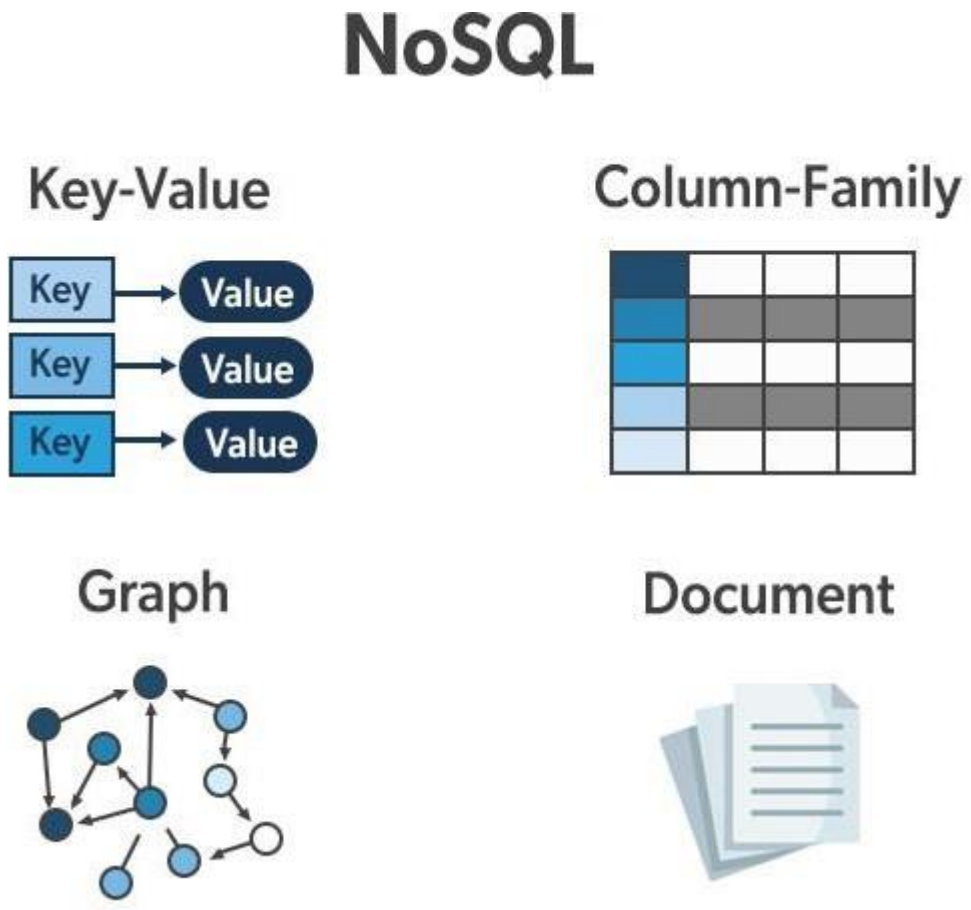
NoSQL Databases

A database is a collection of structured data or information which is stored in a computer system and can be accessed easily. A database is usually managed by a Database Management System (DBMS).

NoSQL is a non-relational database that is used to store the data in the nontabular form. NoSQL stands for Not only SQL. The main types are documents, key-value, wide-column, and graphs.

Types of NoSQL Database:

- Document-based databases
- Key-value stores
- Column-oriented databases
- Graph-based databases



Document-Based Database:

The document-based database is a nonrelational database. Instead of storing the data in rows and columns (tables), it uses the documents to store the data in the database. A document database stores data in JSON, BSON, or XML documents.

Documents can be stored and retrieved in a form that is much closer to the data objects used in applications which means less translation is required to use these data in the applications. In the Document database, the particular elements can be accessed by using the index value that is assigned for faster querying.

Collections are the group of documents that store documents that have similar contents. Not all the documents are in any collection as they require a similar schema because document databases have a flexible schema.

Key features of documents database:

- Flexible schema: Documents in the database has a flexible schema. It means the documents in the database need not be the same schema.
- Faster creation and maintenance: the creation of documents is easy and minimal maintenance is required once we create the document.
- No foreign keys: There is no dynamic relationship between two documents so documents can be independent of one another. So, there is no requirement for a foreign key in a document database.
- Open formats: To build a document we use XML, JSON, and others.

Key-Value Stores:

A key-value store is a nonrelational database. The simplest form of a NoSQL database is a key-value store. Every data element in the database is stored in key-value pairs. The data can be retrieved by using a unique key allotted to each element in the database. The values can be simple data types like strings and numbers or complex objects.

A key-value store is like a relational database with only two columns which is the key and the value.

Key features of the key-value store:

- Simplicity.
- Scalability.
- Speed.

Column Oriented Databases:

A column-oriented database is a non-relational database that stores the data in columns instead of rows. That means when we want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data.

Columnar databases are designed to read data more efficiently and retrieve the data with greater speed. A columnar database is used to store a large amount of data. Key features of columnar oriented database:

- Scalability.
- Compression.
- Very responsive.

Graph-Based databases:

Graph-based databases focus on the relationship between the elements. It stores the data in the form of nodes in the database. The connections between the nodes are called links or relationships.

Key features of graph database:

- In a graph-based database, it is easy to identify the relationship between the data by using the links.
- The Query's output is real-time results.
- The speed depends upon the number of relationships among the database elements.
- Updating data is also easy, as adding a new node or edge to a graph database is a straightforward task that does not require significant schema changes.

Time Series Database

A **time-series database (TSDB)** is a computer system that is designed to store and retrieve data records that are part of a "time series," which is a set of data points that are associated with timestamps. The timestamps provide a critical context for each of the data points in how they are related to others. Time series data is often a continuous flow of data like measurements from sensors and intraday stock prices. A time-series database lets you store large volumes of timestamped data in a format that allows fast insertion and fast retrieval to support complex analysis on that data.

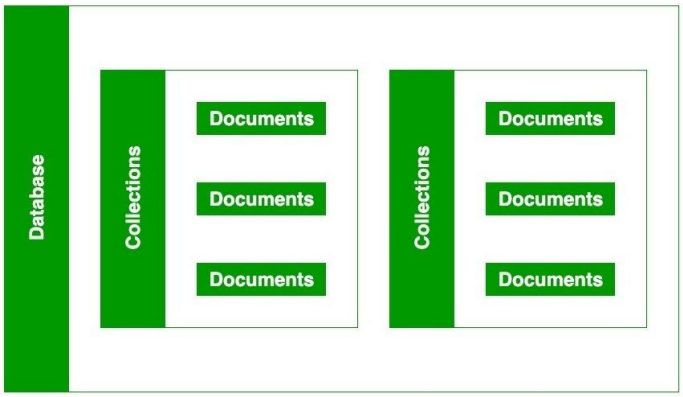
MongoDB

MongoDB is a popular open source document database that's widely used in modern web and mobile applications. It's categorized as a NoSQL database, which means it takes a flexible, document-oriented approach to storing data rather than a traditional table-based relational method.The **MongoDB** is an open-source document database and leading NoSQL database.

MongoDB features are flexible data models that allows the storage of unstructured data. This provides full support indexing, replication, capabilities and also user friendly APIs.

Working of MongoDB

MongoDB is a database server and the data is stored in these databases**.** In other words, MongoDB environment gives us a server that we can start and then create multiple databases on it using MongoDB. Because of its NoSQL database, the data is stored in the collections and documents. Hence the database, collection, and documents are related to each other as shown below:



Basic Architecture of MongoDB

MongoDB database structure consists of the following components:

Component	Equivalent in RDBMS	Description
Database	Database	Stores multiple collections.
Collection	Table	Groups related documents.
Document	Row	A BSON object containing key-value pairs.
Field	Column	Stores data attributes within documents.

2. Storage Format: BSON (Binary JSON)

MongoDB uses **BSON**, an extended version of JSON, which supports additional **data types like binary data, dates, and nested arrays**, improving **query efficiency and storage performance**.

3. Data Storage and Querying

Data is stored in **collections of documents**, making it **highly flexible**.
MongoDB supports **powerful indexing** and **real-time queries**, ensuring **faster read/write operations**.

Developers can **store, retrieve, and update** data using **MongoDB Query Language (MQL)**, which is similar to JSON queries.

4. Schema Flexibility

MongoDB allows **schema-less data storage**, meaning:Collections can store **documents with different structures**.Fields are **not predefined**, allowing **dynamic updates**.Suitable for applications where **data structures frequently change**.

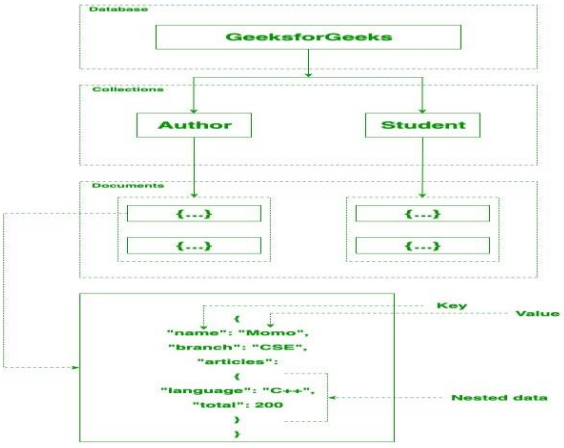
5. Data Relationships in MongoDB

Unlike SQL databases that rely on **foreign keys and JOIN operations**, MongoDB supports:
Embedded Documents – Store related data in a single document (reducing the need for joins).
Reference Documents – Use unique identifiers to establish relationships between documents.

6. Scalability: Horizontal vs. Vertical Scaling

MongoDB is designed for **horizontal scaling**, meaning it can:Distribute data across **multiple servers using sharding**.Support **automatic load balancing and replication** for high availability.Handle **big data and distributed workloads** efficiently.

For example: we have a database named GeeksforGeeks. Inside this database, we have two collections and in these collections we have two documents. And in these documents we store our data in the form of fields. As shown in the below image:



Features of MongoDB

MongoDB offers a wide range of features that make it **a preferred choice for modern applications**.

1. Schema-less Database

Unlike traditional relational databases, MongoDB collections:

- Allow **different structures** within the same collection.
- Do not require **fixed column definitions**.
- Enable **easy updates and modifications**.

2. Document Oriented

In MongoDB, **all the data stored in the documents instead of tables like in** RDBMS**.** In these documents, the data is stored in fields(key-value pair) instead of rows and columns which make the data much more flexible in comparison to RDBMS. And each document contains its unique object id.

3. Indexing

In MongoDB database, **every field in the documents is indexed with primary and secondary indices this makes easier and takes less time to get or search data from the pool of the data**. If the data is not indexed, then database search each document with the specified query which takes lots of time and not so efficient.

4. Scalability

MongoDB **provides horizontal scalability with the help of sharding.** Sharding means to distribute data on multiple servers, here a large amount of data is partitioned into data chunks using the shard key, and these data chunks are evenly distributed across shards that reside across many physical servers. It will also add new machines to a running database.

5. Replication

MongoDB provides high availability and redundancy with the help of replication, it creates multiple copies of the data and sends these copies to a different server so that if one server fails, then the data is retrieved from another server.

6. Aggregation

It **allows to perform operations on the grouped data and get a single result or computed result**. It is similar to the SQL GROUPBY clause.

7. High Performance

The performance of MongoDB is very high and data persistence as compared to another database due to its features like scalability, indexing, replication, etc.

Uses of MongoDB

MongoDB is a popular NoSQL database known for its flexibility, scalability, and performance. It is widely used in various applications across different industries. Here are some common uses of MongoDB:

1. Content Management Systems (CMS)**:**

MongoDB's flexible schema and powerful query capabilities make it an ideal choice for content management systems. It can efficiently handle diverse content types and structures, enabling dynamic and scalable content management solutions.

2. E-commerce Platforms

E-commerce platforms benefit from MongoDB's ability to store and retrieve large amounts of product data quickly. Its flexible schema supports dynamic product catalos, user profiles, shopping carts, and transaction histories.

3. Real-Time Analytics

MongoDB is well-suited for real-time analytics applications due to its high-performance data ingestion and querying capabilities. It can handle large volumes of data in real-time, making it ideal for monitoring, fraud detection, and personalized recommendations.

4. Internet of Things (IoT)

IoT applications generate vast amounts of data from sensors and devices. MongoDB's scalability and flexible data model allow it to efficiently store and process this data, enabling real-time analysis and decision-making for IoT systems.

5. Gaming Applications

Gaming applications generate complex data structures, such as player profiles, scores, achievements, and game states. MongoDB's document-based model allows for efficient storage and retrieval of this data, supporting high-performance gaming experiences.

6. Customer Relationship Management (CRM)

CRM systems use MongoDB to manage customer data, interactions, and sales pipelines. Its ability to handle complex relationships and unstructured data enables more personalized and effective customer engagement strategies.

7. Social Networks

Social networking applications require a database that can handle complex relationships, user-generated content, and real-time interactions. MongoDB's flexibility and scalability make it an excellent choice for building social networks and community platforms.

8. Big Data Applications

MongoDB is used in big data applications for its ability to store and process large volumes of diverse data types. It integrates well with big data technologies like Hadoop and Spark, enabling advanced data analytics and processing

.

9. Healthcare Systems

Healthcare applications use MongoDB to manage patient records, clinical data, and medical images. Its flexible schema allows for the efficient storage of complex healthcare data, supporting better patient care and data analysis.

Advantages of MongoDB

- It is a schema-less NoSQL database. We need not to design the schema of the database when we are working with MongoDB.

- It does not support join operation.
- It provides great flexibility to the fields in the documents.
- It contains heterogeneous (Diffrent) data.
- It provides high performance, availability, scalability.
- It is a document oriented database and the data is stored in BSON documents.
- It also supports multiple document ACID transition.
- It does not require any SQL injection.
- It is easily integrated with Big Data Hadoop

Disadvantages of MongoDB
- High Memory Usage - Requires additional storage
- No Complex Joins - Relies on embedding or referencing instead
- Limited Document Size - Maximum 16MB per document
- Nesting Limits - Supports up to 100 levels of nested documents

Document Data Model

Data in MongoDB has a flexible schema. Documents in the same collection. They do not need to have the same set of fields or structure Common fields in a collections documents may hold different types of data.

Data Model Design

MongoDB provides two types of data models:

1. Embedded data model
2. Normalized data model.

Based on the requirement, you can use either of the models while preparing your document.

Embedded Data Model

In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.

For example, assume we are getting the details of employees in three different documents namely, Personal_details, Contact and, Address, you can embed all the three documents in a single one as shown below −

```
{
      _id: ,
      Emp_ID: "10025AE336"
      Personal_details:{
                  First_Name: "Radhika",
                  Last_Name: "Sharma",
                  Date_Of_Birth: "1995-09-26"
      },
      Contact: {
                  e-mail: "radhika_sharma.123@gmail.com",
                  phone: "9848022338"
      },
      Address: {
                  city: "Hyderabad",
                  Area: "Madapur",
                  State: "Telangana"
      }
}
```

Normalized Data Model

In this model, you can refer the sub documents in the original document, using references. For example, you can re-write the above document in the normalized model as:

Employee:

```
{
    _id: <ObjectId101>,
    Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
    _id: <ObjectId102>,
    empDocID: " ObjectId101",
    First_Name: "Radhika",
    Last_Name: "Sharma",
    Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
    _id: <ObjectId103>,
    empDocID: " ObjectId101",
    e-mail: "radhika_sharma.123@gmail.com",
    phone: "9848022338"
}
```

Address:

```
{
    _id: <ObjectId104>,
    empDocID: " ObjectId101",
    city: "Hyderabad",
    Area: "Madapur",
    State: "Telangana"
}
```

What is a Database in MongoDB?

A **Database** in MongoDB is a container for data that holds **multiple collections**. MongoDB allows the creation of **multiple databases** on a single server, enabling efficient data organization and management for various applications. It's the highest level of structure within the MongoDB system.

1. Multiple Databases: MongoDB allows you to create multiple databases on a single server. Each database is logically isolated(different) from others.

2. Default Databases: When you start MongoDB, three default databases are created: admin, config, and local. These are used for internal purposes.
3. Database Creation: Databases are created when you insert data into them. You can create or switch to a database using the following command:

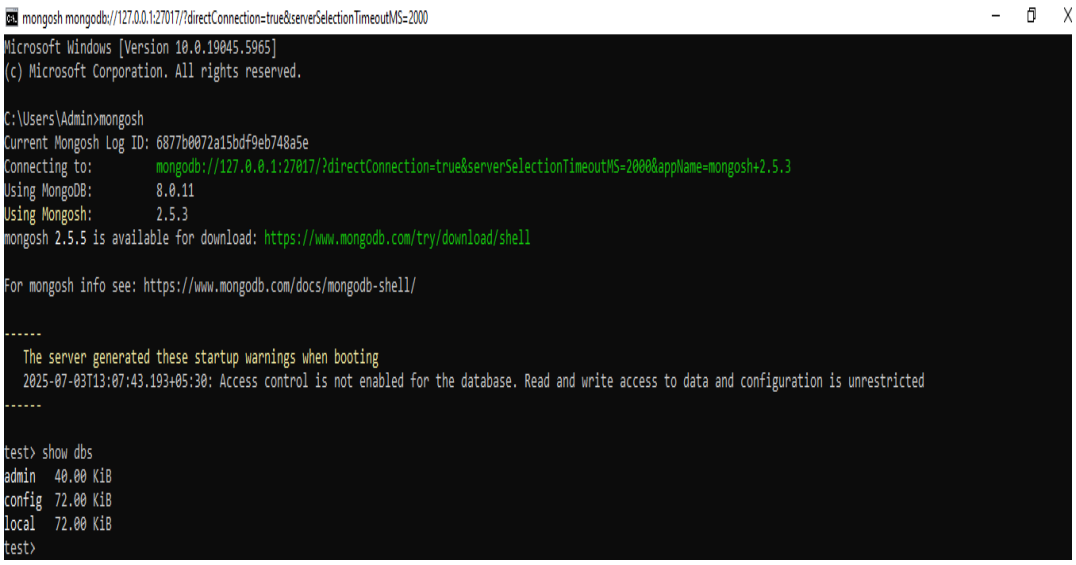
```
use <database_name>
```

This command actually switches you to the new database if the given name does not exist and if the given name exists, then it will switch you to the **existing database**. Now at this stage, if you use the show command to see the database list where you will find that your **new database** is not present in that database list because, in **MongoDB**, the database is actually created when we start entering data in that database.

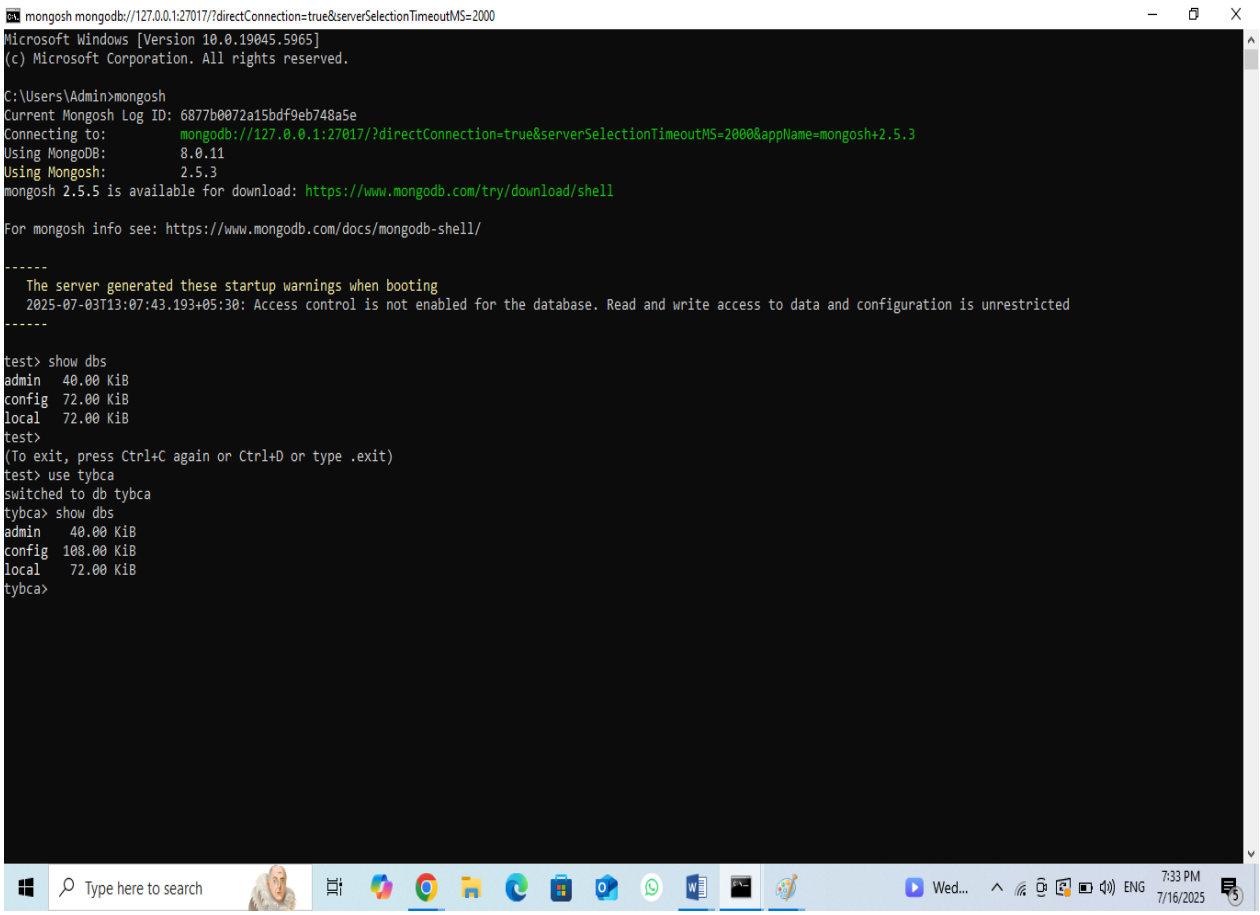
4. View Database: To see how many databases are present in your MongoDB server, write the following statement in the mongo shell:

```
show dbs
```

Here, we freshly started MongoDB so we do not have a database except these three default databases, i.e, admin, config, and local.



Here, we create a **new** database named **tybca** using the use command. After creating a database when we check the database list we do not find our database on that list because we do not enter any data in the **tybca database**.



Naming Restriction for Database:

Before creating a database we should first learn about the **naming restrictions** for databases:

- Database names must be case-insensitive.
- The names cannot contain special characters such as /, ., $, *, |, etc.
- MongoDB database names cannot contain null characters (in windows, Unix, and Linux systems).
- MongoDB database names cannot be empty and must contain less than 64 characters.

What is a Collection in MongoDB?

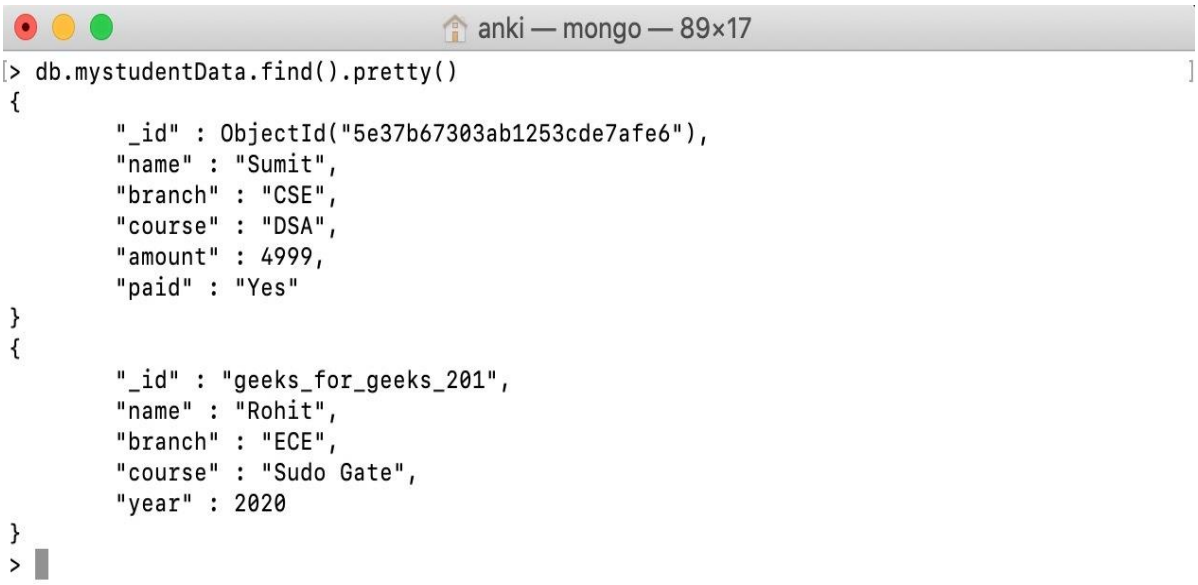
A Collection in MongoDB is similar to a table in **relational databases**. It holds a group of documents and is a part of a database. Collections provide structure to data, but like the rest of MongoDB, they are schema-less.

Schemaless

As we know that **MongoDB databases** are schema less. So, it is not necessary in a collection that the schema of one document is similar to another document. Or in other words, a single collection contains different types of documents like as shown in the below example where **mystudentData** collection contain two different types of documents:

Multiple Collections per Database

A single database can contain multiple collections, each storing different types of documents.



Naming Restrictions for Collection:

Before creating a collection we should first learn about the naming restrictions for collections:

- Collection name must starts with an **underscore** (`_`) or a **letter** (a-z or A-Z)
- Collection name should not start with a number, and does not contain $, empty string, null character and does not begin with prefix `system.` as this is reserved for **MongoDB system collections**.
- The maximum length of the collection name is **120 bytes**(including the database name, dot separator, and the collection name).

Example:

```
db.books.insertOne({ title: "Learn MongoDB", author: "Jane Doe", year: 2023 })
```

Creating collection

After creating database now we create a **collection** to store documents. The collection is created using the following syntax:

```
db.collection_name.insertOne({..})
```

Here, **insertOne()** function is used to store single data in the specified collection. And in the **curly braces {}** we store our data or in other words, it is a document.

For Example:

- In this example, we create a collection named as the **Author** and we insert data in it with the help of insertOne() function. Or in other words, {name: "Ankita"} is a document in the **Author collection**, and in this document, the name is the key or field and "**Ankita**" is the value of this key or field.

- After pressing enter we got a **message** (as shown in the above image) and this message tells us that the data enters successfully (i.e., "acknowledge": true) and also assigns us an automatically created id.

- It is the **special feature** provided by MongoDB that every document provided a **unique id** and generally, this id is created automatically, but you are allowed to create your own id (must be unique).

What is a Document in MongoDB?

In **MongoDB**, the data records are stored as BSON documents. Here, BSON stands for binary representation of **JSON documents**, although BSON contains more data types as compared to JSON. The document is created using **field-value pairs** or **key-value pairs** and the value of the field can be of any BSON type.

Syntax:
{
field1: value1
field2: value2
....
fieldN: valueN
}

Document Structure:

A document in MongoDB is a flexible data structure made up of **field-value pairs**. For instance:

```
{
  title: "MongoDB Basics",
  author: "John Doe",
  year: 2025
}
```

Naming restriction for Document Fields:

Before moving further first you should learn about the naming restrictions for fields:

- Fields in documents must be named with strings
- The _id field name is reserved to use as a primary key. And the value of this field must be unique, immutable, and can be of any type other than an array.
- The field name cannot contain null characters.
- The top-level field names should not start with a **dollar sign** ($).

Document Size:

The maximum size of the **BSON document** is 16MB. It ensures that the single document does not use too much amount of **RAM** or **bandwidth**(during transmission). If a document contains more data than the specified size, then MongoDB provides a **GridFS API** to store such type of documents. A single document may contain duplicate fields.

MongoDB always saves the order of the fields in the documents except for the _id field (which always comes in the first place) and the renaming of fields may change the order of the fields in the documents.

What is the _id Field in MongoDB?

In **MongoDB**, every document store in the collection must contain a **unique _id** field it is just like a primary key in a relational database. The value of the _id field can be set by the user or by the system (if the user does not create an _id field, then the system will automatically generate an ObjectId for _id field).

- **Automatic ObjectId Generation**: When you don't define the _id field, MongoDB generates a unique ObjectId by default.
- **Custom _id**: You can set the _id field to a custom value, provided it is unique within the collection.

Example with ObjectId:

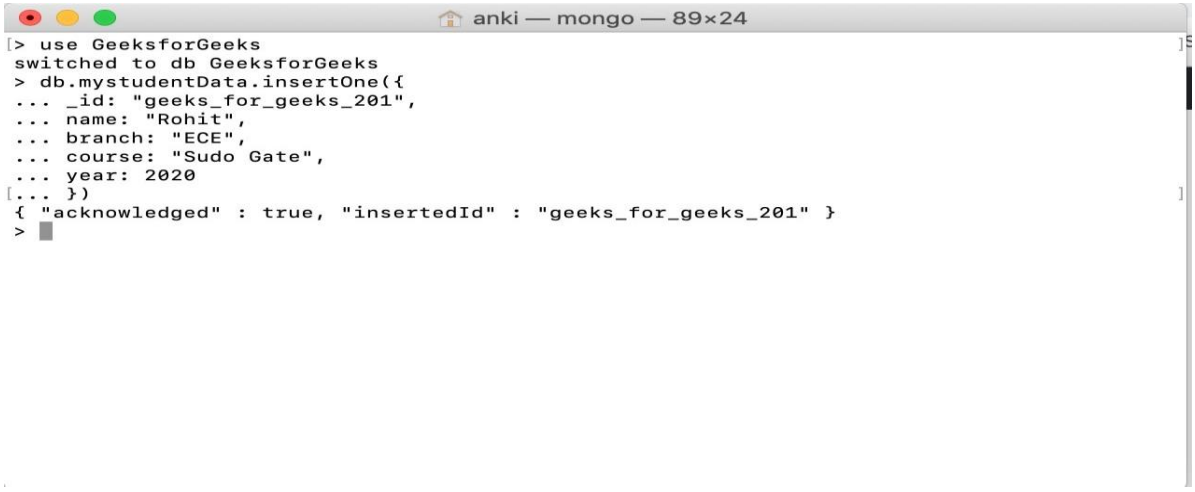
```
anki — mongo — 89×24
. Number of files is 256, should be at least 1000
---
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible t
o you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring
()
---

> use GeeksforGeeks
switched to db GeeksforGeeks
> db.mystudentData.insertOne({ name: "Sumit", branch: "CSE", course: "DSA", amount: 4999,
 paid: "Yes"})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("5e37b67303ab1253cde7afe6")
}
>
```

Here, **name**, **branch**, **course**, and paid field contain values of string type. amount field contains the value of integer type and _id field is generated by the system.
Example with Custom _id:



Here, the _id field is created by the user. When you paste data in the functions always use **close parenthesis** after pasting the data into the function. If you use close parenthesis before pasting data in the function, then you will get an error.

Key Differences Between Databases, Collections, and Documents

- **Database**: A container for collections, providing structure and logical isolation for data.
- **Collection**: A group of documents within a database, similar to a table in relational databases.
- **Document**: A single data record within a collection, stored as a BSON object.

Practical Example: Creating a Database, Collection, and Document
Here's how you can create a database, collection, and document in MongoDB step by step:

1. Create or Switch to a Database

```
use LibraryDB
```

2. Create a Collection and Insert a Document

```
db.books.insertOne({
  title: "MongoDB for Beginners",
  author: "Alice Johnson",
  year: 2023
})
```

3. Verify the Insertion

```
db.books.find()
```

This will display the document stored in the books collection within the **LibraryDB database**.
MongoDB Indexes and the _id Field
Automatic Index on _id: MongoDB automatically creates a unique index on the _id field for every collection. This index helps MongoDB quickly find documents based on their unique identifier.

Best Practices for MongoDB Databases, Collections, and Documents

- **Descriptive Names**: Name databases and collections based on their content for better organization. For example, use user_data or transaction_records for database and collection names.
- **Avoid Special Characters**: Ensure that database and collection names do not contain special characters like $ or spaces.
- **Design Efficient Documents**: While MongoDB is schema-less, it's important to design documents that make sense for your application's data structure. Use nested documents and arrays when appropriate to model complex data.

MongoDB Shell

The **MongoDB** Shell, also known as **mongosh** is a powerful **command-line interface** that allows users to interact with **MongoDB databases**.

The MongoDB Shell (mongosh) is a **JavaScript interface** for interacting with MongoDB databases. It provides a **command-line environment** where you can execute **MongoDB commands** to manage your database, perform **administrative tasks**, and **manipulate data**. In short, the MongoDB Shell allows you to manage and work with your MongoDB databases directly through a **flexible** and powerful **command-line interface**.

Key features of the MongoDB Shell include:

- **Interacting with MongoDB databases**: Connect to and manage MongoDB instances.
- **Running administrative commands**: Perform tasks like managing users, roles, and collections.
- **Querying and manipulating data**: Use MongoDB's powerful querying capabilities to read, write, and update data.
- **Automating tasks**: Execute JavaScript code within the shell to automate repetitive tasks or run scripts for bulk operations.

How to Start MongoDB Using the Shell?

Starting MongoDB using the Shell is straightforward. Below are the steps to **start the MongoDB using Shell** are as follows:

Step 1: Connecting to MongoDB Server

- **Open your terminal** (Command Prompt on Windows, or terminal on macOS/Linux).
- **Run the following command** to start the MongoDB shell:

```
mongosh
```

```
C:\Users\bisha>mongosh
Current Mongosh Log ID: 65d512ce4932abd79f29859c
Connecting to:          mongodb://127.0.0.1:27017/?directConnection=true
&serverSelectionTimeoutMS=2000&appName=mongosh+1.8.0
Using MongoDB:          7.0.5
Using Mongosh:          1.8.0

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

------
   The server generated these startup warnings when booting
   2024-02-21T01:57:06.899+05:30: Access control is not enabled for the
database. Read and write access to data and configuration is unrestricte
d
------

test> |
```

Output after 'mongosh' command

Step 2: Executing MongoDB Commands

Like any other command-line interface, we can enter MongoDB commands directly in the shell. Here are some essential ones:

1. Listing Databases

To list all databases on your MongoDB instance, use:

```
show dbs
```

```
test> show dbs
admin     40.00 KiB
config   108.00 KiB
local     40.00 KiB
test> |
```

displays all databases present locally

2. Switching Databases

To switch to a specific database, use the use command followed by the database name:

```
use <database_name>
```

Example:

```
use admin
```

3. Creating Collections

To create a collection within the selected database, you can simply insert a document into a collection, and MongoDB will create it automatically. Here's an example:

```
db.createCollection("myCollection")
```

Alternatively, you can insert data directly into a non-existing collection, and it will be created:

```
db.myCollection.insert({ name: "MongoDB", type: "Database" })
```

4. Querying Data

To query data from a collection, use the find method:

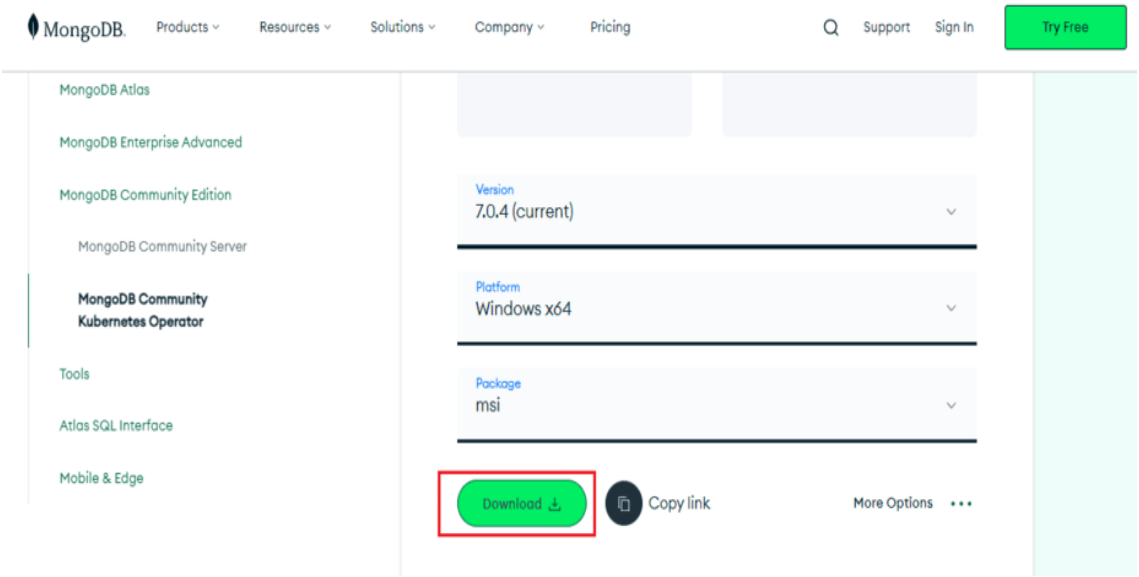
```
db.myCollection.find({ name: "MongoDB" })
```

This will return all documents where the name is "MongoDB".

Installation and Configuration of MongoDB

Step 1: Download MongoDB Community Server

Go to the MongoDB Download Center to download the MongoDB Community Server.

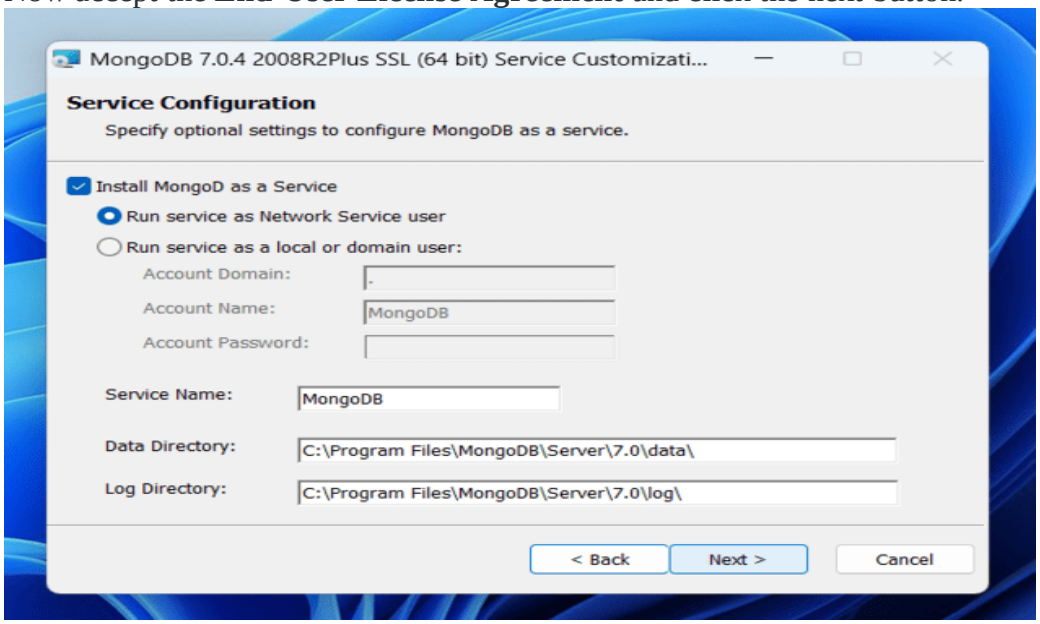


Here, You can select any version, Windows, and package according to your requirement. For Windows, we need to choose:
Version: 7.0.4
OS: Windows x64
Package: msi

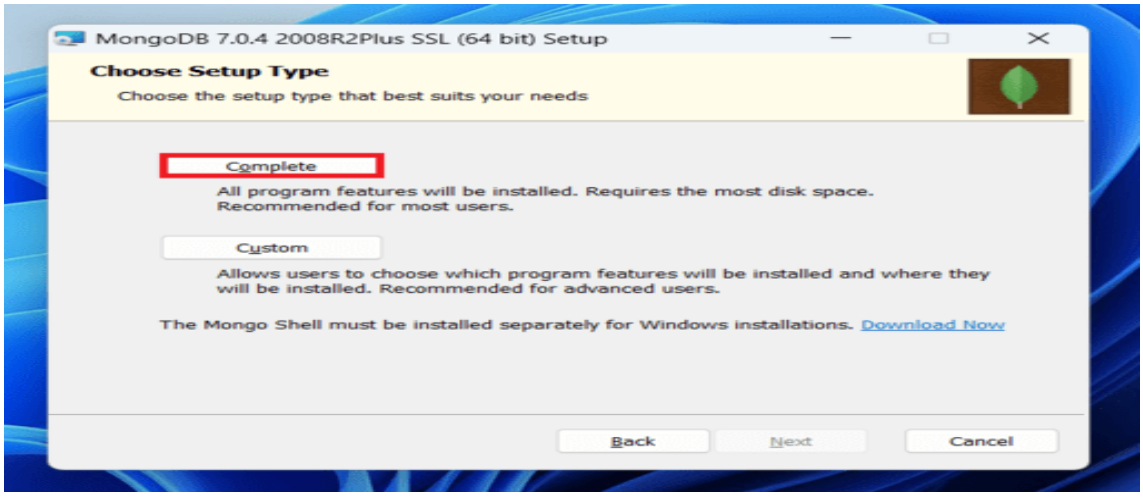
Step 2: Install MongoDB

When the download is complete open the **msi file** and click the **next button** in the startup screen:

Now accept the **End-User License Agreement** and click the next button:

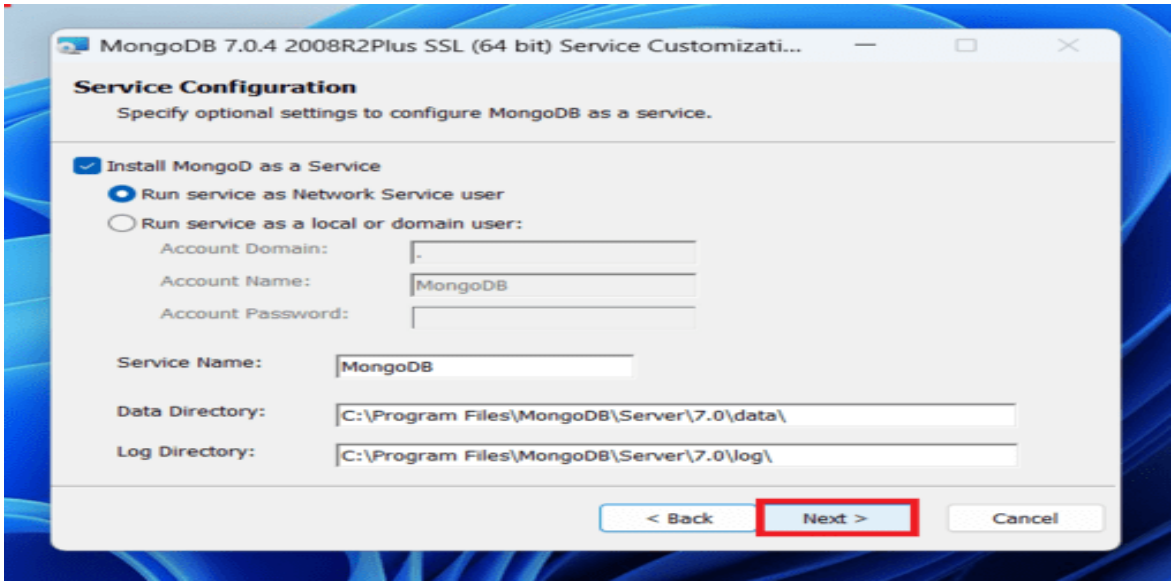


Now select the **complete option** to install all the program features. Here, if you can want to install only selected program features and want to select the location of the installation, then use the **Custom option***:*

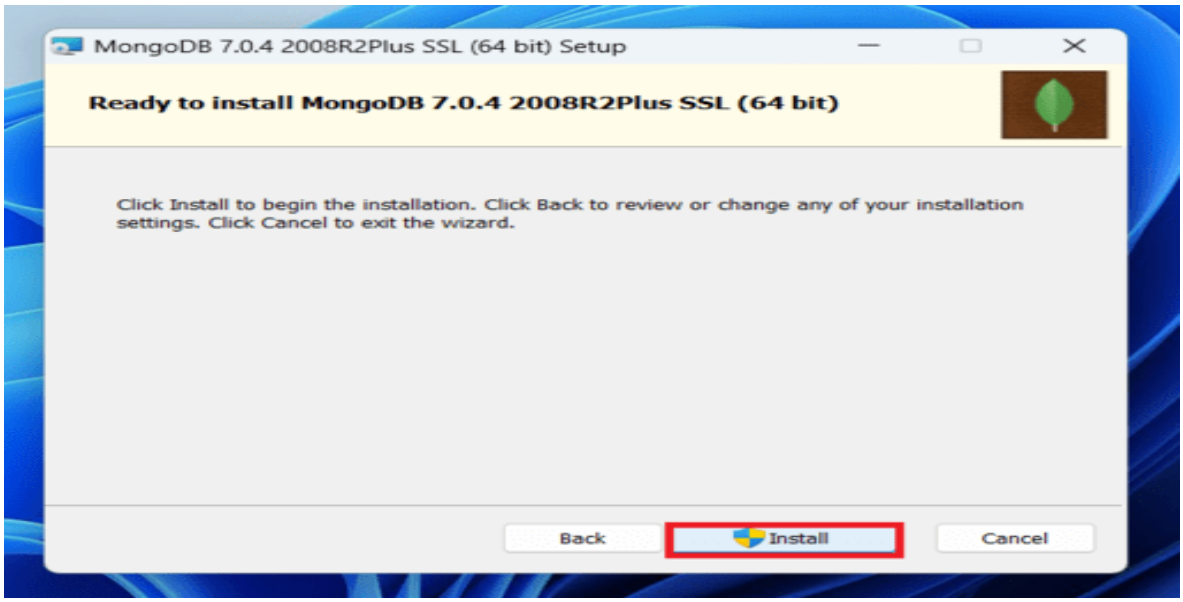


Step 3: Configure MongoDB Service

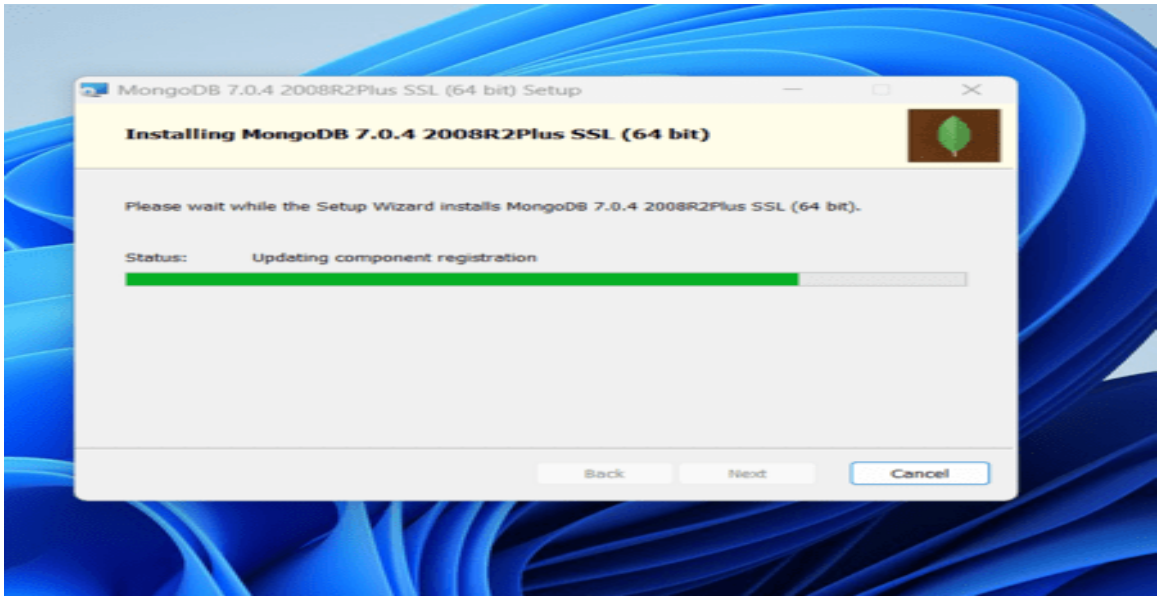
Select "**Run service as Network Service use**r" and copy the path of the data directory. **Click Next**:



Click the **Install button** to start the MongoDB installation process:



After clicking on the install button installation of MongoDB begins:

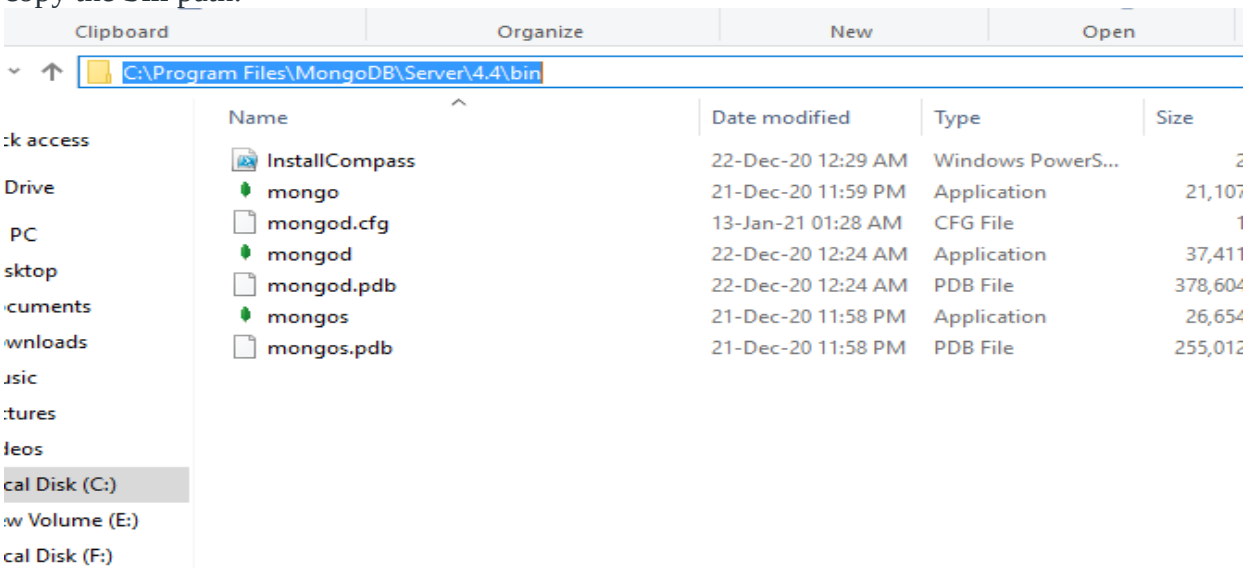


Step 4: Complete Installation

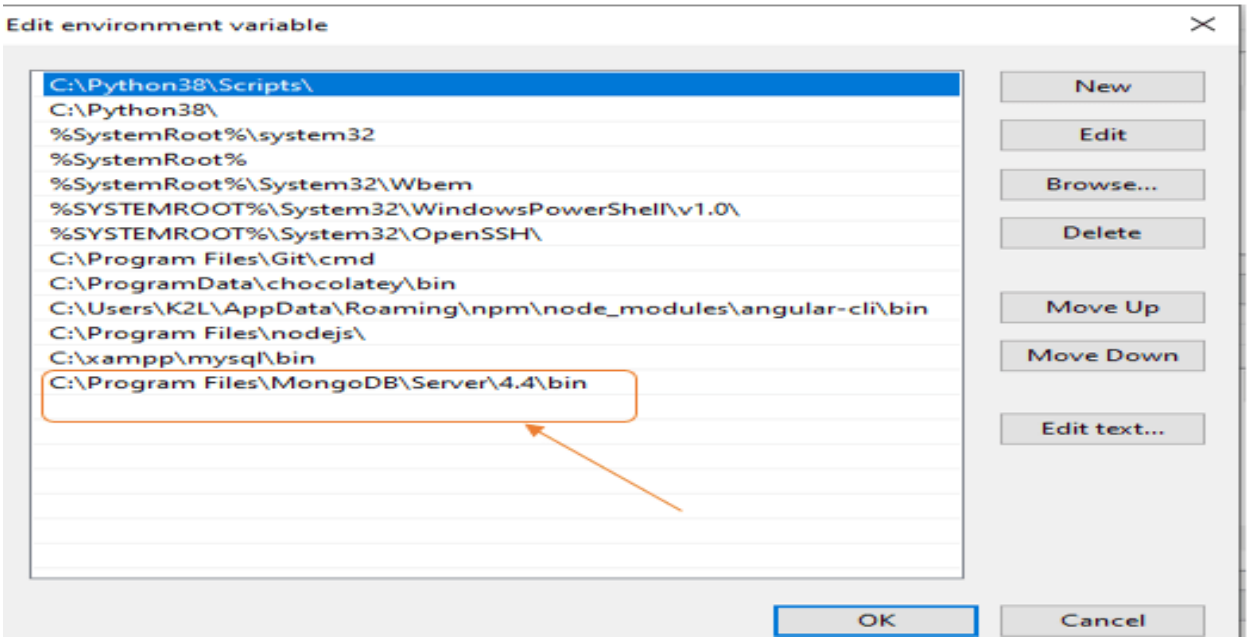
Now click the **Finish button** to complete the MongoDB installation process:

Step 5: Set Environment Variables

Now we go to the location where MongoDB installed in step 5 in your system and copy the **bin** path:



Now, to create an environment variable open system **properties >> Environment Variable >> System variable >> path >> Edit Environment variable** paste the copied link to your environment system and **click Ok**:



Run MongoDB Server (mongod)

Step 1. Start MongoDB Service

After setting the environment variable, we will run the MongoDB server, i.e. **mongod**. So, open the **command prompt** and run the following command:

mongod

When you run this command you will get an error i.e. **C:/data/db/ not found**.

Step 2. Create Required Folders

Now, Open **C** drive and create a folder named "**data**"Inside the **data** folder create another folder named "**db**".

Step 3. Restart MongoDB

After creating these folders. Again open the command prompt and run the following command:

mongod

Now, this time the MongoDB server(i.e., mongod) will run successfully.

```
C:\Users\NIkhil Chhipa>mongod
{"t":{"$date":"2021-01-31T00:56:54.081+05:30"},"s":"I",  "c":"CONTROL",  "id":23285,   "ctx"
ify --sslDisabledProtocols 'none'"}
{"t":{"$date":"2021-01-31T00:56:54.087+05:30"},"s":"W",  "c":"ASIO",     "id":22601,   "ctx"
}
{"t":{"$date":"2021-01-31T00:56:54.088+05:30"},"s":"I",  "c":"NETWORK",  "id":4648602, "ctx"
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"STORAGE",  "id":4615611, "ctx"
bPath":"C:/data/db/","architecture":"64-bit","host":"DESKTOP-L9MUQ7N"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":23398,   "ctx"
rgetMinOS":"Windows 7/Windows Server 2008 R2"}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":23403,   "ctx"
gitVersion":"913d6b62acfbb344dde1b116f4161360acd8fd13","modules":[],"allocator":"tcmalloc","
}}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":51765,   "ctx"
ndows 10","version":"10.0 (build 14393)"}}}
{"t":{"$date":"2021-01-31T00:56:54.090+05:30"},"s":"I",  "c":"CONTROL",  "id":21951,   "ctx"
{"t":{"$date":"2021-01-31T00:56:54.157+05:30"},"s":"I",  "c":"STORAGE",  "id":22270,   "ctx"
:{"dbpath":"C:/data/db/","storageEngine":"wiredTiger"}}
{"t":{"$date":"2021-01-31T00:56:54.158+05:30"},"s":"I",  "c":"STORAGE",  "id":22315,   "ctx"
ize=1491M,session_max=33000,eviction=(threads_min=4,threads_max=4),config_base=false,statist
le_manager=(close_idle_time=100000,close_scan_interval=10,close_handle_minimum=250),statisti
ess],"}}
{"t":{"$date":"2021-01-31T00:56:54.395+05:30"},"s":"I",  "c":"STORAGE",  "id":22430,   "ctx"
95788][3708:140713908197088], txn-recover: [WT_VERB_RECOVERY_PROGRESS] Recovering log 20 thr
{"t":{"$date":"2021-01-31T00:56:54.631+05:30"},"s":"I",  "c":"STORAGE",  "id":22430,   "ctx"
```

Run the MongoDB Shell (mongosh)

Starting from MongoDB version 5.0, the traditional MongoDB shell (mongo) has been **deprecated**. The recommended shell for interacting with MongoDB databases is now **mongosh**, which provides improved functionality, better syntax, and full compatibility with the latest MongoDB features.

Step 1. Connect to MongoDB Server with mongosh

Now we are going to connect our server (mongod) with the mongo shell. So, keep that mongod window

open a new command prompt window and type:

mongosh

You are now connected to the MongoDB shell.

Please do not close the mongod window if you close this window your server will stop working and it will not able to connect with the mongo shell.

```
C:\Users\NIkhil Chhipa>mongo
MongoDB shell version v4.4.3
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("96cca5da-dc9f-4a40-aabb-732ee37600c0") }
MongoDB server version: 4.4.3
---
The server generated these startup warnings when booting:
        2021-01-28T20:56:52.570+05:30: Access control is not enabled for the database. Read and write access
configuration is unrestricted
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
>
```

What is a Database in MongoDB?

A **Database** in MongoDB is a container for data that holds **multiple collections**. MongoDB allows the creation of **multiple databases** on a single server, enabling efficient data organization and management for various applications. It's the highest level of structure within the MongoDB system.

1. Multiple Databases: MongoDB allows you to create multiple databases on a single server. Each database is logically isolated(different) from others.
2. Default Databases: When you start MongoDB, three default databases are created: admin, config, and local. These are used for internal purposes.
3. Database Creation: Databases are created when you insert data into them. You can create or switch to a database using the following command:

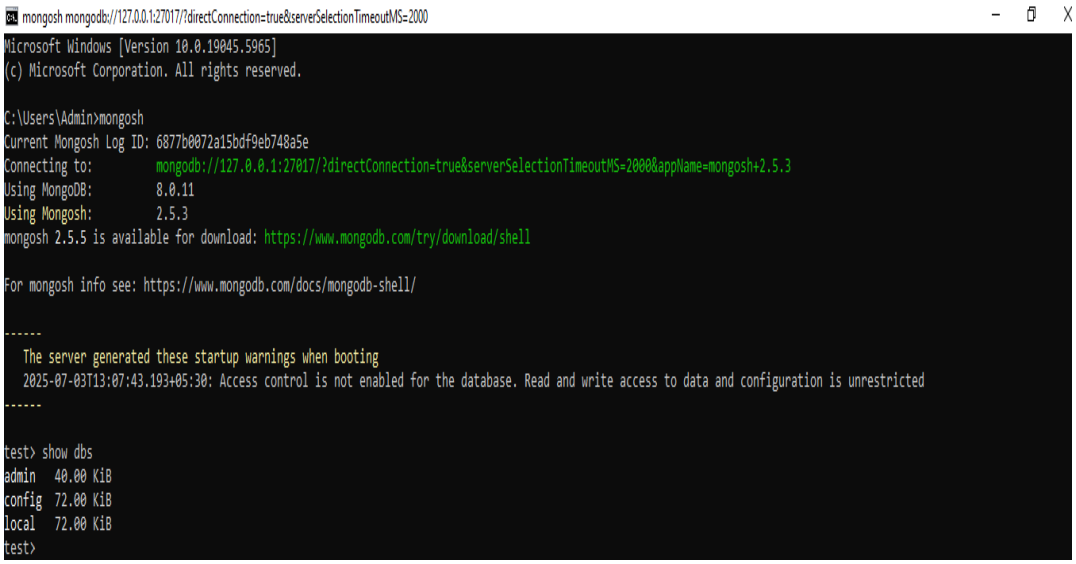
```
use <database_name>
```

This command actually switches you to the new database if the given name does not exist and if the given name exists, then it will switch you to the **existing database**. Now at this stage, if you use the show command to see the database list where you will find that your **new database** is not present in that database list because, in **MongoDB**, the database is actually created when we start entering data in that database.

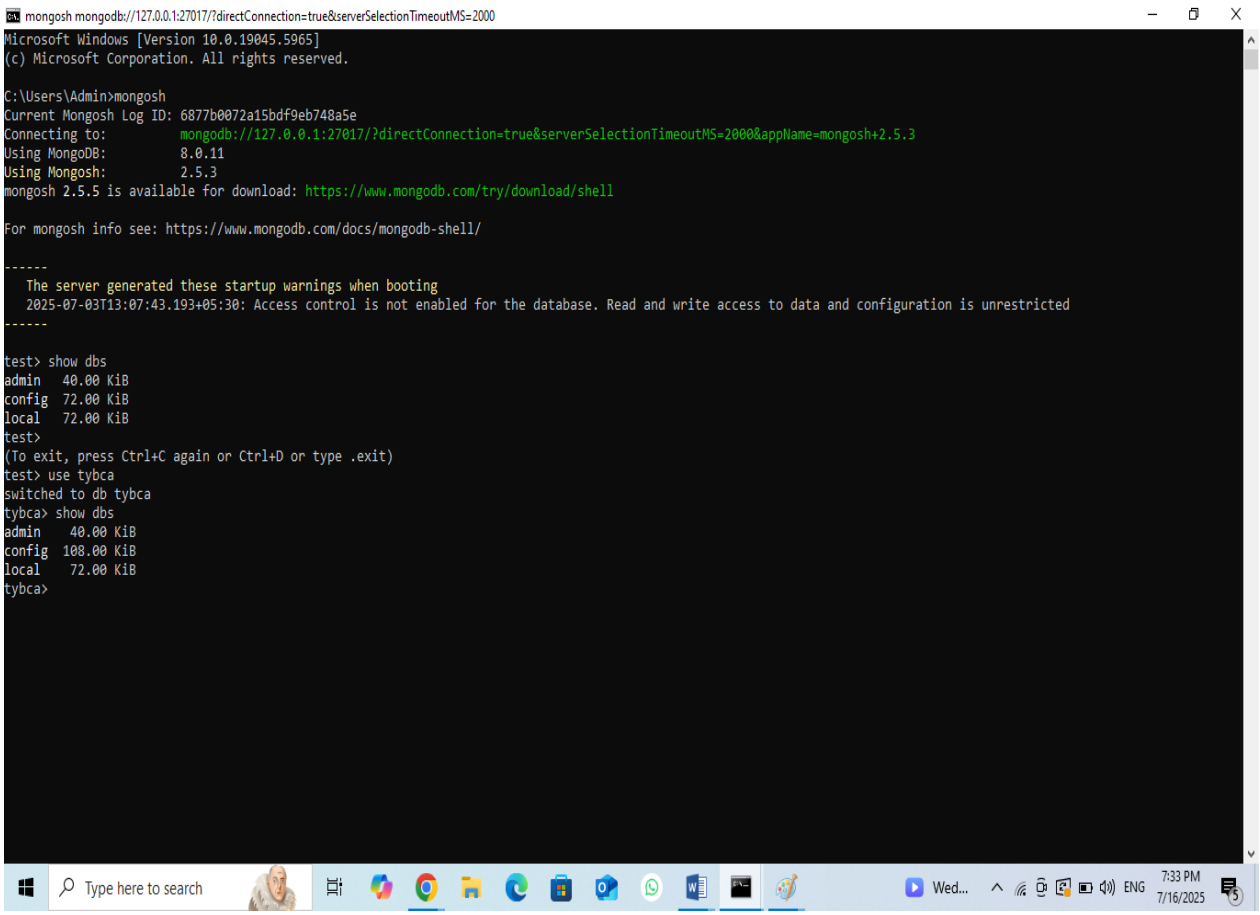
4. View Database: To see how many databases are present in your MongoDB server, write the following statement in the mongo shell:

```
show dbs
```

Here, we freshly started MongoDB so we do not have a database except these three default databases, i.e, admin, config, and local.



Here, we create a **new** database named **tybca** using the use command. After creating a database when we check the database list we do not find our database on that list because we do not enter any data in the **tybca database**.



Naming Restriction for Database:

Before creating a database we should first learn about the **naming restrictions** for databases:

- Database names must be case-insensitive.
- The names cannot contain special characters such as /, ., $, *, |, etc.
- MongoDB database names cannot contain null characters(in windows, Unix, and Linux systems).
- MongoDB database names cannot be empty and must contain less than 64 characters.

Drop a Database in MongoDB?

Dropping a database in MongoDB is a **permanent** action that deletes all collections and documents within it. This operation is performed using the **db.dropDatabase()** method. Let's assume we have a database called **sampleDB** and we want to delete it. Follow the steps below to safely **drop a database** while ensuring proper execution.

Step 1: List Available Databases

Before dropping a database, it's essential to check which databases exist in MongoDB. To list all available databases, run the following command in the MongoDB shell:

```
show dbs
```

Example Output:

```
admin          0.000GB
local          0.781GB
sampleDB       0.230GB
myDatabase  0.150GB
```

This output displays all the databases on the MongoDB server along with their sizes.

Step 2: Select the Database to Drop

Now, we need to **switch to the database** that we want to drop. Use the use command followed by the database name:

```
use sampleDB
```

Example Output:

```
switched to db sampleDB
```

After executing this command, MongoDB sets sampleDB as the **active database**.

Step 3: Drop the Selected Database

Once the database is selected, we can execute the db.dropDatabase() command to delete it.

```
db.dropDatabase()
```

Example Output:

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

- This command will permanently delete the current database sampleDB and return a confirmation message.
- The "dropped" field confirms that **sampleDB has been deleted**.
- The "ok": 1 indicates that the operation was **successful**.

Important: This action is **permanent**. All collections and documents inside the database will be removed.

Step 4: Verify That the Database Has Been Dropped

To ensure the database has been dropped successfully, we can list the databases again:

```
show dbs
```

Example Output After Dropping testDB:

```
admin       0.000GB
local       0.781GB
myDatabase  0.150GB
```

Notice that **sampleDB is no longer listed**, meaning it has been successfully deleted.
Precautions Before Dropping a Database

- **Backup Your Data:** If you might need the data in the future, create a backup using MongoDB's mongodump command.
- **Verify the Database Name:** Ensure you are working with the correct database before executing db.dropDatabase().
- **Check User Permissions:** Make sure your MongoDB user has **admin privileges** to drop databases.

Create a Collection in MongoDB

In MongoDB, a collection is a **group of documents**. Collections are similar to tables in relational databases. However, collections do not enforce schema, allowing allowing you to store documents with varying structures in the same collection. This flexibility is one of the **key features** of MongoDB.

To explicitly create a collection in MongoDB, you can use the **createCollection()** method. However, it's important to note that collections are also automatically created when you insert the first document into them.

Syntax:
db.createCollection(' collection_name');

Example:
To create a collection called Student, you would use the following command:

```
db.createCollection('Student');
```

Explanation:

- The **createCollection()** method is used to explicitly create a collection in MongoDB.
- If you attempt to insert a document into a collection that doesn't exist, MongoDB will automatically create it for you without needing the **createCollection()** method.

For example, running the following command will automatically create the **Student collection** if it doesn't exist:

```
db.Student.insertOne({Name: "Om", age: 19})
```

Output

```
test> use GeeksforGeeks;
switched to db GeeksforGeeks
GeeksforGeeks> db.createCollection('Student');
{ ok: 1 }
GeeksforGeeks> db.Student.insertOne({Name:"Om",Age:19});
{
  acknowledged: true,
  insertedId: ObjectId("65a66db13cfbaabe01a3670b")
}
GeeksforGeeks> db.Student.find();
[ { _id: ObjectId("65a66db13cfbaabe01a3670b"), Name: 'Om', Age: 19 } ]
GeeksforGeeks>
```

createCollection() in MongoDB

Note:

- Collections are created automatically when the first document is inserted, so using **createCollection()** is optional.
- MongoDB allows for collections to contain documents with different fields and data types, which provides flexibility when structuring your data.

Insert Documents into a Collection in MongoDB

Once you have a collection, you can start **adding documents** to it. MongoDB stores data in the form of documents, which are essentially JSON**-like objects**. There are two primary methods for inserting documents:

1. insertOne() Method

The **insertOne() method** in MongoDB is used to insert a single document into a collection. This is the simplest way to add a new record into a collection when you need to insert just one document at a time.

Syntax:
db.collection_name.insertOne({ field1: value1, field2: value2, ... });

Example:

```
db.myNewCollection1.insertOne( { name:"geeksforgeeks" } )
```

Output:

```
> db.myNewCollection1.insertOne( { name:"geeksforgeeks" } )
{
        "acknowledged" : true,
        "insertedId" : ObjectId("6017a92c0cf217478ba93593")
}
```

Explanation:

- The insertOne () method adds a single document to the collection.
- In this example, we create a collection named as "myNewCollection1" by inserting a document that contains a "name" field with its value in it using insertOne() method.

2. insertMany() method

The insertMany() method in MongoDB allows you to **insert multiple documents** into a collection at once. This method is ideal when you need to add multiple records in one operation, improving performance compared to inserting documents one by one.

Syntax:

db.collection_name.insertMany([{ field1: value1, field2: value2, ... }, { field1: value1, field2: value2, ... }, ...]);

Example:

Let's say we want to create a collection named **myNewCollection2** and insert two documents. The first document will have a name field with the value "gfg" and country field with "India". The second document will have name as "rahul" and age as 20.

```
db.myNewCollection2.insertMany([{name:"gfg", country:"India"},
                    {name:"rahul", age:20}])
```

Output:

```
> db.myNewCollection2.insertMany([{name:"gfg", country:"India"}, {name:"rahul",
age: 20}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("6017a97d0cf217478ba93594"),
                ObjectId("6017a97d0cf217478ba93595")
        ]
}
>
```

View Existing Collections

After **creating collections** and **inserting data**, it's important to check the contents of your **current database** by listing all the collections. In MongoDB, you can view all the collections in your **current database** by using the **show collections command**.

Syntax:

show collections

Example:

If you're working in the **gfgDB**database and you want to see all the collections, simply run:

```
show collections
```

Output:

```
> show collections
myNewCollection1
myNewCollection2
>
```

This shows that the Student collection exists within the gfgDB database.

MongoDB Drop Collection

- In MongoDB, the **drop method** is used to remove a collection from a database. When we drop a collection, it deletes the entire collection along with all the documents it contains.
- The **drop command** is used to permanently delete a collection from a MongoDB database.

Syntax:

db.Collection_name.drop({writeConcern: <document>})

Examples of MongoDB Drop Collection

Let's look at some of the examples of **db.collection.drop()** method in MongoDB. These examples explain **how to delete a collection in MongoDB**.

Suppose we have callection called **students** in **gfg database** and we will perfrom all the operation on that collection for better understanding.

Example 1

We will drop the **student collection** in the **gfg database**.

```
db.student.drop()
```

Output:

The output confirms that the **student collection** has been dropped successfully:

```
{ "ok" : 1 }
```

It drops the student collection and all the indexes associated with the collection.

Example 2

In the following example, we are working with:

- **Database**: gfg
- **Collections**: student_gfg, teacher, semester

Suppose we want to drop the **teacher collection** from the **gfg database**. So, we use the drop method:

```
db.teacher.drop()
```

Output:

The output confirms that the **teacher collection** has been dropped successfully:

```
{ "ok" : 1 }
```

This method drops the teacher collection along with its documents.

Working with document

In **MongoDB**, the data records are stored as BSON documents. Here, BSON stands for binary representation of **JSON documents**, although BSON contains more data types as compared to JSON. The document is created using **field-value pairs** or **key-value pairs** and the value of the field can be of any BSON type.
Syntax:

{
field1: value1
field2: value2
....
fieldN: valueN
}

Document Structure:
A document in MongoDB is a flexible data structure made up of **field-value pairs**.

For instance:

```
{
  title: "MongoDB Basics",
  author: "John Doe",
  year: 2025
}
```

Insert documents

The MongoDB shell provides the following methods to insert documents into a collection:

- To insert a single document, use db.collection.insertOne().
- To insert multiple documents, use db.collection.insertMany().

MongoDB insertOne() Method

The MongoDB **insertOne()** method is used to add a single document to a collection. It adds the document to the specified collection and assigns it a unique **_id** if one is not provided. This **insertOne()** method is part of the MongoDB driver for various programming languages and can be used in MongoDB Shell, Node.js, Python and other environments.

- We can insert documents with or without the **_id** field. If we insert a document in the collection without the _id field, MongoDB will automatically add an _id field and assign it with a unique **ObjectId**.
- if we insert a document with the _id field, then the value of the _id field must be unique to avoid the **duplicate key error.**
- This method can also throw either **writeError** or **writeConcernError** exception.
- This method can also be used inside multi-document transactions.

Syntax:

db.Collection_name.insertOne(
<document>,
{
writeConcern: <document>
}
)

Key Terms

- **<document>**: The document we want to insert. A document is a set of key-value pairs similar to a JSON object.
- **writeConcern (optional)**: If we need to specify a custom write concern (e.g., to ensure the data is written to multiple nodes), you can include this option.

Examples of MongoDB insertOne()

Let's go over a few examples to understand how insertOne() works in MongoDB. In the following examples, we are working with:

- **Database:** gfg
- **Collection:** student
- **Document:** No document but, we want to insert in the form of the student name and student marks.

```
> use gfg
switched to db gfg
> db.student.find().pretty()
>
```

Example 1: Insert a Document without Specifying an _id Field

Here, we are inserting the document whose name is Akshay and marks is 500 in the student collection. MongoDB will automatically assign a unique _id field to this document.

Query:

```
db.student.insertOne({Name: "Akshay", Marks: 500})
```

Output:

```
> use gfg
switched to db gfg
> db.student.insertOne({Name: "Akshay", Marks: 500})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("600e8c710cf217478ba93565")
}
> db.student.find().pretty()
{
        "_id" : ObjectId("600e8c710cf217478ba93565"),
        "Name" : "Akshay",
        "Marks" : 500
}
>
```

Explanation: As shown above, MongoDB has inserted the document with a new ObjectId automatically generated for the _id field.

Example 2: Insert a Document Specifying an _id Field

Here, we are inserting a document whose unique id is Stu102, name is Vishal, and marks is 230 in the student collection

Query:

```
db.student.insertOne({_id: "Stu102", Name: "Vishal", Marks: 230})
```

Output:

```
> db.student.insertOne({_id: "Stu102",Name: "Vishal", Marks: 230})
{ "acknowledged" : true, "insertedId" : "Stu102" }
> db.student.find().pretty()
{
        "_id" : ObjectId("600e8c710cf217478ba93565"),
        "Name" : "Akshay",
        "Marks" : 500
}
{ "_id" : "Stu102", "Name" : "Vishal", "Marks" : 230 }
>
```

Explanation: Here, we specified the _id as "Stu102", and MongoDB inserts the document successfully.

find() document

find() method in **MongoDB** is a tool for retrieving documents from a collection. It supports various **query** operators and enabling complex queries. The find() method is the primary way to retrieve documents from a collection in **MongoDB**.

findOne() Method

The findOne() in MongoDB is a method used to retrieve a **single document** from a collection that matches a specified criteria. It returns **only one** document, even if multiple documents match the criteria. If no matching document is found, it returns **null**.

Examples of Find() Method

To understand **find() method** in MongoDB we need a collection and some documents on which we will perform various operations and queries. Here we will consider a collection called **student** of **gfg** database which contains the following documents:

- **Database:** gfg
- **Collections:** student
- **Document:** Three documents contains the details of the students

Example 1: Find All Documents in a Collection

```
db.student.find()
```

Output:

```
> db.student.find().pretty()
{
        "_id" : ObjectId("60227eaff19652db63812e8d"),
        "name" : "Akshay",
        "age" : 18
}
{
        "_id" : ObjectId("60227eaff19652db63812e8e"),
        "name" : "Bablue",
        "age" : 17,
        "score" : {
                "math" : 230,
                "science" : 234
        }
}
{
        "_id" : ObjectId("60227eaff19652db63812e8f"),
        "name" : "Chandhan",
        "age" : 18
}
>
```

Explanation: In the above query, we have found all the documents in the **students** collections in MongoDB.

Example 2: Find Documents with a Specific Condition
Find all the students whose age is exactly 18:

```
db.student.find({age:18})
```

Output:

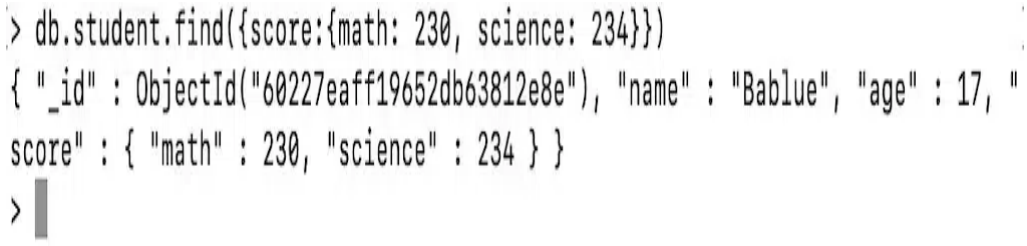
```
> db.student.find({age:18})
{ "_id" : ObjectId("60227eaff19652db63812e8d"), "name" : "Akshay", "age" : 18 }
{ "_id" : ObjectId("60227eaff19652db63812e8f"), "name" : "Chandhan", "age" : 18
}
>
```

Explanation: In the above query, we have find those **students** whose age is 18.

Example 3: Using Nested Documents in Queries
Let's Find student records from the "**student**" collection where the student's **math** score is **230** and **science** score is **234**.

```
db.student.find({score:{math: 230, science: 234}})
```

Output:



Modify documents

Update operations allow us to modify documents in a collection. These operations can update a single document or multiple documents based on specified criteria. MongoDB offers various update operators to perform specific actions like **setting a value, incrementing a value** or **updating elements within arrays**.

update() Method

The **MongoDB update()** method is a method that is used to update a single document or **multiple** documents in the collection. When the document is updated the **_id field** remains unchanged. The **db.collection.update()** method updates a single document by default. To update all documents that match the given query, use the multi: true option. Include the option "**multi: true**".

Syntax

db.COLLECTION_NAME.update({SELECTION_CRITERIA}, {$set:{UPDATED_DATA}},

Examples of MongoDB update

To understand **MongoDB update** we need a collection called **students** on which we will perform various operations and queries.

```
[
  { _id: 1, name: "Alice", age: 25, grades: [
      { grade: "Maths", score: 80 },
      { grade: "Science", score: 85 }
    ]
  },
  { _id: 2, name: "Bob", age: 30, grades: [
      { grade: "Maths", score: 75 },
      { grade: "Science", score: 90 }
    ]
  },
  { _id: 3, name: "Charlie", age: 35, grades: [
      { grade: "Maths", score: 95 },
      { grade: "Science", score: 88 }
    ]
  }
]
```

Example 1: Update a Single Document

Write a query to update the age of a student named Alice to 26 in the "**students**" collection.

Query:

```
db.students.updateOne(
   { name: "Alice" },
   { $set: { age: 26 } }
)
```

Output:

```
[
  {
    _id: 1,
    name: 'Alice',
    age: 26,
    grades: [ { grade: 'Maths', score: 80 }, { grade: 'Science', score: 85 } ]
  },
  {
    _id: 2,
    name: 'Bob',
    age: 30,
    grades: [ { grade: 'Maths', score: 75 }, { grade: 'Science', score: 90 } ]
  },
  {
    _id: 3,
    name: 'Charlie',
    age: 35,
    grades: [ { grade: 'Maths', score: 95 }, { grade: 'Science', score: 88 } ]
  }
]
```

Example 2: Use Update Operator Expressions ($inc and $set)

Write a query to increment the age of a student named Alice by 1 in the "**students**" collection.

Query:

```
db.students.updateOne(
   { name: "Alice" },
   { $inc: { age: 1 } }
)
```

Output:

```
[
  {
    _id: 1,
    name: 'Alice',
    age: 27,
    grades: [ { grade: 'Maths', score: 80 }, { grade: 'Science', score: 85 } ]
  },
  {
    _id: 2,
    name: 'Bob',
    age: 30,
    grades: [ { grade: 'Maths', score: 75 }, { grade: 'Science', score: 90 } ]
  },
  {
```

```
    _id: 3,
    name: 'Charlie',
    age: 35,
    grades: [ { grade: 'Maths', score: 95 }, { grade: 'Science', score: 88 } ]
  }
]
```

Example 3: Insert a New Document if No Match Exists (Upsert)

Write a query to update the age of a student named Charlie to 30 in the "students" collection. If Charlie does not exist, a new document should be inserted with the specified age.

Query:

```
db.students.updateOne(
   { name: "Charlie" },
   { $set: { age: 30 } },
   { upsert: true }
)
```

Output:

```
[
  {
    _id: 1,
    name: 'Alice',
    age: 26,
    grades: [ { grade: 'Maths', score: 80 }, { grade: 'Science', score: 85 } ]
  },
  {
    _id: 2,
    name: 'Bob',
    age: 31,
    grades: [ { grade: 'Maths', score: 75 }, { grade: 'Science', score: 90 } ]
  },
  {
    _id: 3,
    name: 'Charlie',
    age: 30,
    grades: [ { grade: 'Maths', score: 95 }, { grade: 'Science', score: 88 } ]
  }
]
```

Removing Document

The MongoDB remove() method allows users to remove documents from a collection based on specific criteria. It is a powerful tool in MongoDB that enables both single and bulk document deletion, offering flexibility in managing your database. It supports various options like removing only one document and specifying a **write concern.**

The **remove()** method in MongoDB is used to delete documents from a collection. It can remove a single document or multiple documents that match the given query condition.

By passing an **empty query document** ({}), we can remove all documents from the collection. However, it's important to note that MongoDB's remove() method has been deprecated in newer versions, and you should prefer using the **deleteOne()** or **deleteMany()** methods for future-proof applications.

Syntax:

```
db.Collection_name.remove(
<matching_criteria>,
{
   justOne: <boolean>,
   writeConcern: <document>,
   collation: <document>
})
```

Parameters

- **query**: The condition or criteria to match the documents for deletion. If an empty document {} is provided, all documents in the collection are removed.
- **justOne**: (Optional) If set to true, only the first matching document is removed. The default is false, which removes all matching documents.
- **writeConcern**: (Optional) Specifies the level of acknowledgment requested from MongoDB for write operations. This can be used to adjust the write concern if you want more control over the operation's durability and safety.
- **collation**: (Optional) Allows the specification of language-specific rules for string comparison, such as case sensitivity and accent marks.

Examples of MongoDB remove()

In these examples, we demonstrate how to use the **remove() method** on a MongoDB collection. The student collection in the **gfg database** contains student records, each with fields like **name** and **age**. We'll show different scenarios, including removing documents based on specific conditions or clearing all documents from the collection.

In the following examples, we are working with:

- **Database:** gfg
- **Collections:** student
- **Document:** Three documents contains name and the age of the students

```
> use gfg
switched to db gfg
> db.student.find().pretty()
{
        "_id" : ObjectId("600ee5c60cf217478ba93578"),
        "name" : "Akshay",
        "age" : 19
}
{
        "_id" : ObjectId("600ee5c60cf217478ba93579"),
        "name" : "Bablu",
        "age" : 18
}
{
        "_id" : ObjectId("600ee5c60cf217478ba9357a"),
        "name" : "chetan",
        "age" : 18
}
>
```

Example 1: Remove All Documents that Match a Condition

To remove all documents that match a specific condition, pass a query to the remove() method. Here, we remove all the documents from the student collection where the name is "Akshay".

Query:

```
db.student.remove({name: "Akshay"})
```

Output:

```
> db.student.remove({name: "Akshay"})
WriteResult({ "nRemoved" : 1 })
> db.student.find().pretty()
{
        "_id" : ObjectId("600ee5c60cf217478ba93579"),
        "name" : "Bablu",
        "age" : 18
}
{
        "_id" : ObjectId("600ee5c60cf217478ba9357a"),
        "name" : "chetan",
        "age" : 18
}
>
```

Example 2: Remove All Documents from a Collection

To delete all documents from a collection, we can pass an empty document ({}) as the query. Here, we remove all the documents from the student collection by passing empty document(i.e., {}) in the remove() method.

Query:

```
db.student.remove({})
```

Output:

```
> db.student.remove({})
WriteResult({ "nRemoved" : 2 })
> db.student.find().pretty()
>
```

Example 3: Remove a Single Document that Matches a Condition

If we want to remove only one document that matches the query criteria, set the justOne option to true. Here, two documents matched the specified condition, but we only want to remove one document, so we set the value of justOne option to true.

Query:

```
db.student.remove({age:{$eq:18}}, true)
```

Output:

```
> db.student.find().pretty()
{
        "_id" : ObjectId("601158af0cf217478ba9357c"),
        "name" : "chetan",
        "age" : 18
}
>
```

Indexing

Indexing in **MongoDB** is a crucial feature that enhances query processing efficiency. Without indexing, MongoDB must scan every document in a collection to retrieve the **matching** documents and leading to **slower query performance**.

Indexes are special data structures that store information about the documents in a way that makes it easier for MongoDB to quickly locate the right data.

What is Indexing in MongoDB?

Indexing in MongoDB is a technique that improves the speed and efficiency of queries. An index is a special data structure that stores a subset of data in a way that allows MongoDB to quickly locate documents in a collection. When an index is applied, MongoDB doesn't have to scan the entire collection for the query results but instead uses the index to directly find the relevant data.

Indexes in MongoDB are **ordered** by the value of the field that the index is created on. This ordered structure enables faster searching, sorting, and filtering operations. Indexes also help to improve query performance when using conditions, sorting, and aggregation

Why is Indexing Important in MongoDB?

MongoDB provides a method called createIndex() that allows users to create an index. The key determines the field on the basis of which we want to create an index and 1 (or -1) determines the order in which these indexes will be arranged(ascending or descending).

Indexing improves the performance of:

- **Find queries** (db.collection.find()).
- **Range queries** (e.g., queries with <, >, <=, >= operators).
- **Sorting** (e.g., db.collection.find().sort()).
- **Aggregation operations** involving filtering, grouping, and sorting.

How to Create an Index in MongoDB

To create an index, MongoDB provides the createIndex() method. The method requires us to specify the field(s) to index and the order (ascending or descending). We can also specify optional parameters to customize the index creation.

Syntax

db.collection.createIndex({ <field>: <1 or -1> });

Example

```
db.users.createIndex({ username: 1 });
```

Parameters:

- **unique:** Ensures the indexed field contains unique values.
- **background:** Creates the index in the background to avoid blocking other database operations.
- **sparse:** Only indexes documents that contain the indexed field.
- **expireAfterSeconds:** Used for time-to-live (TTL) indexes to automatically remove documents after a certain time.
- **hidden:** Marks the index as hidden, meaning it will not be used for queries but still exists in the system.

How to Drop an Index in MongoDB

We can drop an index using the dropIndex() method. To drop multiple indexes, use dropIndexes(). The dropIndex() methods can only delete one index at a time. In order to delete (or drop) multiple indexes from the collection, MongoDB provides the dropIndexes() method that takes multiple indexes as its parameters.

Syntax (drop a single index):

db.NAME_OF_COLLECTION.dropIndex({KEY:1})

Syntax (drop multiple indexes):
db.NAME_OF_COLLECTION.dropIndexes({KEY1:1, KEY2: 1})

How to View all Indexes in MongoDB

The getIndexes() method in MongoDB gives a description of all the indexes that exists in the given collection.

Syntax

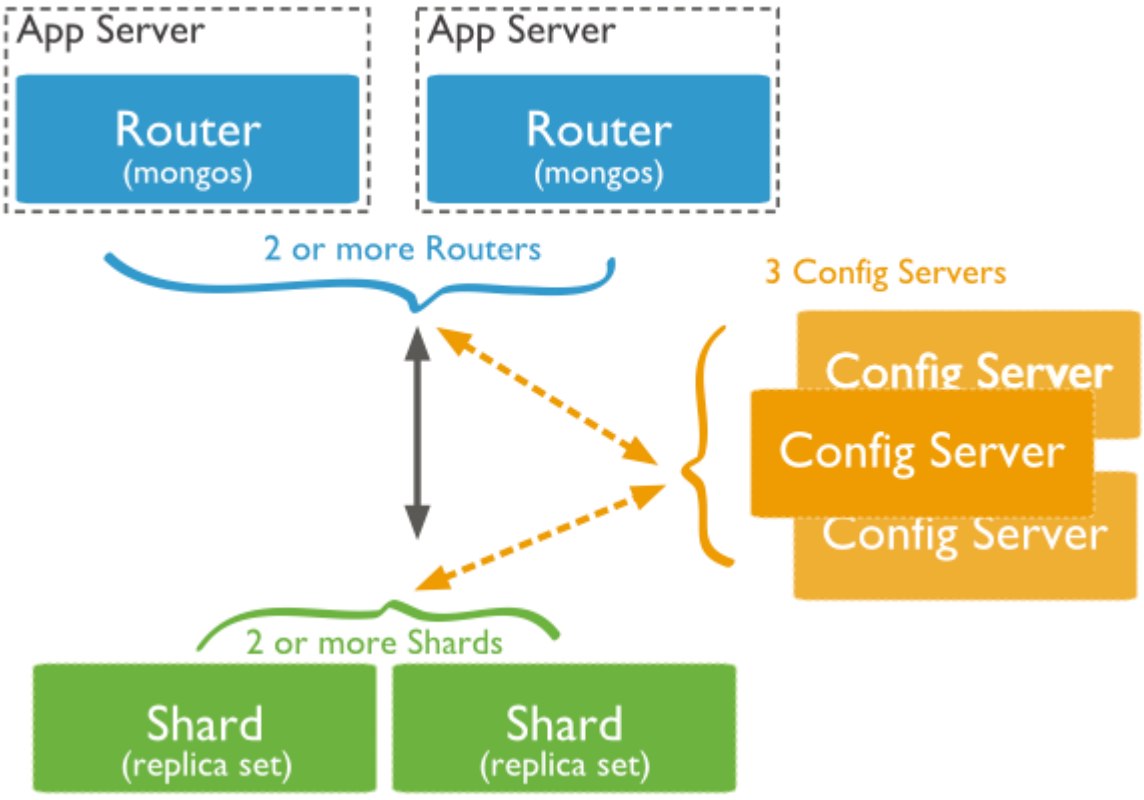
db.NAME_OF_COLLECTION.getIndexes()

It will retrieve all the description of the indexes created within the collection.

Sharding

Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

The following diagram shows the Sharding in MongoDB using sharded cluster.



In the following diagram, there are three main components −

Shards − Shards are used to store data. They provide high availability and data consistency. In production environment, each shard is a separate replica set.

Config Servers − Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. In production environment, sharded clusters have exactly 3 config servers.

Query Routers − Query routers are basically mongo instances, interface with client applications and direct operations to the appropriate shard. The query router processes and targets the operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Generally, a sharded cluster have many query routers.

Writing to a shard in MongoDB

Writing to a shard in MongoDB. In MongoDB, **sharding** is a method for distributing data across multiple servers (shards) to handle large datasets and high-throughput operations.

How Writing to Shards Works

When you insert or update documents in a **sharded collection**:

1. **MongoDB uses the shard key** to determine the target shard.
2. The **mongos router** routes the write request to the correct shard(s):
    - **Targeted write**: If the operation specifies a shard key, it goes directly to the relevant shard.
    - **Broadcast write**: If the shard key is not included, the operation might be sent to all shards.
3. The **primary of that shard** handles the write operation (MongoDB still uses replica sets under the hood for each shard).

2. Steps to Write to a Sharded Cluster

1. **Enable Sharding on the Database**

   sh.enableSharding("myDatabase")

2. **Shard the Collection**
   Define a **shard key** (a field that decides the document distribution across shards):

   sh.shardCollection("myDatabase.myCollection", { userId: 1 })

    - userId is the shard key.
    - The shard key **must be present in every write operation** for targeted writes.
3. **Insert Data**

   use myDatabase

   // Example write with shard key
   db.myCollection.insertOne({ userId: 101, name: "Alice", age: 25 })

This write will be **routed to the correct shard** based on userId.

Using **MongoDB as a file system** is possible through a feature called **GridFS**. MongoDB is not designed as a traditional file system like NTFS or ext4, but **GridFS** allows you to store and retrieve files (images, videos, PDFs, etc.) that are larger than the BSON document size limit of 16MB.

Here's a complete explanation:

1. What is GridFS in MongoDB?

- **GridFS** is a specification for storing and retrieving large files in MongoDB.
- Instead of storing a single file as a document, **GridFS splits the file into smaller chunks** (default 255 KB each) and stores them across two collections:
    1. `fs.files` – Metadata for each file (filename, upload date, length, etc.)
    2. `fs.chunks` – Binary chunks of the file

When to Use GridFS

Use GridFS if:

- Your file size is **greater than 16MB**.
- You want to **store files in MongoDB instead of the OS file system**.
- You need **file replication and high availability** using MongoDB's replica sets.
- You want to **stream files to/from MongoDB** in applications.

Following is a sample document of fs.files collection −

```
{
   "filename": "test.txt",
   "chunkSize": NumberInt(261120),
   "uploadDate": ISODate("2014-04-13T11:32:33.557Z"),
   "md5": "7b762939321e146569b07f72c62cca4f",
   "length": NumberInt(646)
}
```

The document specifies the file name, chunk size, uploaded date, and length.

Following is a sample document of fs.chunks document –

```
{
  "files_id": ObjectId("534a75d19f54bfec8a2fe44b"),
  "n": NumberInt(0),
  "data": "Mongo Binary Data"
}
```

Adding Files to GridFS

Now, we will store an mp3 file using GridFS using the put command. For this, we will use the mongofiles.exe utility present in the bin folder of the MongoDB installation folder.

Open your command prompt, navigate to the mongofiles.exe in the bin folder of MongoDB installation folder and type the following code –

```
>mongofiles.exe -d gridfs put song.mp3
```

Here, gridfs is the name of the database in which the file will be stored. If the database is not present, MongoDB will automatically create a new document on the fly. Song.mp3 is the name of the file uploaded. To see the file's document in database, you can use find query −

```
>db.fs.files.find()
```

The above command returned the following document −

```
{
  _id: ObjectId('534a811bf8b4aa4d33fdf94d'),
  filename: "song.mp3",
  chunkSize: 261120,
  uploadDate: new Date(1397391643474), md5: "e4f53379c909f7bed2e9d631e15c1c41",
  length: 10401959
}
```

We can also see all the chunks present in fs.chunks collection related to the stored file with the following code, using the document id returned in the previous query −

```
>db.fs.chunks.find({files_id:ObjectId('534a811bf8b4aa4d33fdf94d')})
```

In my case, the query returned 40 documents meaning that the whole mp3 document was divided in 40 chunks of data.

Unit 4: Cassandra

4.1 The Column-Family Data Model

Apache Cassandra employs a column-family data model, which differs significantly from the table-based structure of traditional relational databases. Instead of organizing data strictly into rows and columns, Cassandra uses a more flexible, schema-optional structure based on key-value pairs grouped into column families.

- **Column Family**: A column family is conceptually similar to a relational table but allows for a much more flexible design. Each row in a column family is identified by a unique key (row key), and the columns in each row can vary in number and name.
- **Rows and Columns**:
    - Each row is a collection of columns.
    - Each column is a triplet composed of a name, value, and a timestamp.
    - Columns within the same row are sorted by their names.

This model enables Cassandra to store sparse data efficiently and allows for rapid growth in data volume without requiring rigid schema modifications.

For example, a user profile table in Cassandra might look like this:

```
Row Key: user123
  Name: "Alice"
  Email: "alice@example.com"
  Age: 30

Row Key: user456
  Name: "Bob"
  Phone: "123-456-7890"
```

Note that each user (row) can have different columns, and Cassandra handles this gracefully.

4.2 Databases and Tables

In Cassandra, a **keyspace** is analogous to a **database** in relational systems. It is the outermost container for data and is used to define data replication strategies for the data stored within it.

- **Keyspace**:
    - Defines the **replication strategy** (SimpleStrategy, NetworkTopologyStrategy).
    - Sets the **replication factor**, which is the number of nodes that will receive copies of the same data.

- **Tables (Column Families)**:
    - Tables reside within a keyspace.
    - Tables consist of rows and columns but allow variable column structures per row.
    - Each table has a defined **schema** that includes data types for defined columns and a **primary key**.

Primary Key Design:

- A primary key is composed of:
    - **Partition Key**: Determines how data is distributed across nodes.
    - **Clustering Columns**: Determine the order of data storage within a partition.

Example:

```
CREATE TABLE users (
  user_id UUID,
  name TEXT,
  email TEXT,
  PRIMARY KEY (user_id)
);
```

Here, `user_id` is the partition key, and the table defines a fixed schema, although additional columns can be added.

4.3 Columns, Types, and Keys

Columns: Each column in Cassandra includes:

- **Column Name**: The identifier for the column.
- **Value**: The actual data stored.
- **Timestamp**: Used for resolving write conflicts (last-write-wins).

Data Types: Cassandra supports various built-in data types, such as:

- Primitive types: `int`, `text`, `boolean`, `float`, `double`, `timestamp`, `uuid`
- Collections: `list<type>`, `set<type>`, `map<key_type, value_type>`
- User-defined types (UDTs) allow for complex custom structures.

Keys:

- **Primary Key** = Partition Key [+ Clustering Columns]
- **Partition Key** determines which node will store the data.
- **Clustering Columns** define how data is ordered within a partition.

Designing primary keys wisely is crucial for performance and scalability in Cassandra. A poorly chosen key can result in uneven data distribution and hotspots.

Example with Clustering:

```
CREATE TABLE sensor_data (
  device_id UUID,
  timestamp TIMESTAMP,
  reading FLOAT,
  PRIMARY KEY (device_id, timestamp)
);
```

Here, `device_id` is the partition key, and `timestamp` is the clustering column that sorts the data within each device's partition.

4.4 Cassandra's Architecture

Cassandra's architecture is designed for high availability, fault tolerance, and horizontal scalability. It follows a **peer-to-peer** distributed model where all nodes are equal.

Key Architectural Features:

- **Decentralized Design**:
  - No single point of failure.
  - Each node in the cluster has the same role.
- **Partitioning and Distribution**:
  - Data is partitioned using **consistent hashing**.
  - The **partitioner** decides how data is distributed based on partition keys.
- **Replication**:
  - Data is replicated to multiple nodes.
  - The **replication factor** determines how many copies are maintained.
  - **Replica placement** is determined by the keyspace's replication strategy.
- **Tunable Consistency**:
  - Cassandra allows clients to choose consistency levels (e.g., ONE, QUORUM, ALL) based on application needs.

Core Components:

- **Gossip Protocol**:
  - Used for communication between nodes.
  - Exchanges metadata like state and health information.
- **Snitch**:

    - Helps Cassandra understand the network topology (datacenters and racks).
    - Optimizes replica placement and query routing.
- **Commit Log, Memtable, SSTable**:
    - **Commit Log**: Records all writes for durability.
    - **Memtable**: In-memory data structure storing recent writes.
    - **SSTable**: On-disk immutable files created from memtables.
- **Compaction**:
    - Merges SSTables to remove obsolete data and improve read efficiency.

Advantages of Cassandra's Architecture:

- Linear scalability: Add more nodes to handle more data.
- High availability: No downtime for maintenance or node failures.
- Fault tolerance: Data is replicated across multiple nodes.
- Write-optimized: Extremely fast write performance.

Cassandra is ideal for:

- Real-time big data applications
- Internet of Things (IoT) platforms
- Time-series data
- Logging and monitoring systems

Would you like this content exported as a formatted Word document (.docx)?

Unit 5: MongoDB with Python

5.1 Basics of Python Module Python modules are files containing Python code. These files can define functions, classes, and variables, and can also include runnable code. A module allows code reusability and organization, making it easier to maintain and manage.

- To create a module, save the code in a `.py` file.
- Use the `import` statement to include a module in another Python script.
- Python comes with many built-in modules like `math`, `os`, and `datetime`.

Example:

```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

# main.py
import mymodule
greeting = mymodule.greet("Alice")
print(greeting)
```

5.2 Introduction to PyMongo Module PyMongo is the official Python driver for MongoDB. It allows Python applications to connect to MongoDB, perform database operations, and manage data.

To install PyMongo:

```
pip install pymongo
```

PyMongo provides:

- Tools for connecting to MongoDB databases.
- Functions to perform CRUD (Create, Read, Update, Delete) operations.
- Integration with MongoDB queries and aggregation frameworks.

5.3 Working with Database with PyMongo To start working with MongoDB in Python:

1. Import `pymongo` and establish a connection.
2. Access the desired database and collection.

Example:

```
from pymongo import MongoClient

# Connect to MongoDB server
client = MongoClient("mongodb://localhost:27017/")

# Access database
db = client["mydatabase"]
```

```
# Access collection
collection = db["customers"]
```

5.4 CRUD Operations in PyMongo CRUD stands for Create, Read, Update, and Delete. These are the four basic functions of persistent storage.

- **Create**:

```
collection.insert_one({"name": "Alice", "age": 25})
```

- **Read**:

```
document = collection.find_one({"name": "Alice"})
print(document)
```

- **Update**:

```
collection.update_one({"name": "Alice"}, {"$set": {"age": 26}})
```

- **Delete**:

```
collection.delete_one({"name": "Alice"})
```

5.5 Flask API with MongoDB Database Flask is a lightweight web framework in Python. It can be used to build RESTful APIs that interact with MongoDB using PyMongo.

Steps:

1. Install Flask and PyMongo:

```
pip install flask pymongo
```

2. Create a Flask application that connects to MongoDB:

```
from flask import Flask, jsonify, request
from pymongo import MongoClient

app = Flask(__name__)
client = MongoClient("mongodb://localhost:27017/")
db = client["mydatabase"]
collection = db["customers"]

@app.route("/add", methods=["POST"])
def add_customer():
    data = request.json
    collection.insert_one(data)
    return jsonify({"message": "Customer added"}), 201

@app.route("/get", methods=["GET"])
def get_customers():
    customers = list(collection.find({}, {"_id": 0}))
    return jsonify(customers)
```

```python
if __name__ == "__main__":
    app.run(debug=True)
```

This simple API lets you insert and retrieve customer data from a MongoDB database using HTTP requests.

continue