MongoDB:-

1. **Install MongoDB**
2. **Data Modeling**
3. **Query and Projection**
4. **Aggregation Pipeline**

Introduction:-

1. **Data**
2. **Database**
3. **NoSQL**
4. **What is MongoDB**
5. **Features of MongoDB**
6. **How MongoDB works?**
7. **Database, Collection and Documents**

Introduction:-

1. Data:-

Data is information such as facts and numbers used to analyze something or make decisions. Computer data is information in a form that can be processed by a computer.

2. Database:-

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS).

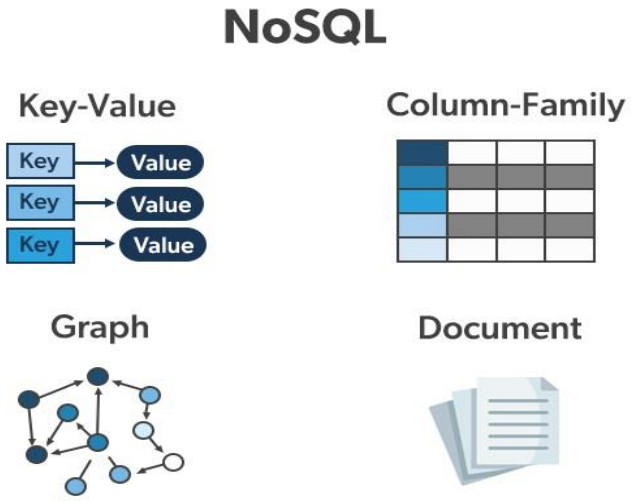
3. NoSQL:-

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



1. 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.

2. 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.

3. 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.

4. 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.

4. What is MongoDB?

MongoDB is a **document-oriented** NoSQL database system that provides high scalability, flexibility, and performance. Unlike standard relational databases, MongoDB stores data in

a JSON document structure form. This makes it easy to operate with dynamic and unstructured data and MongoDB is an open-source and cross-platform database System.

Why Use MongoDB?

Document Oriented Storage − Data is stored in the form of JSON documents.

- **Index on any attribute**: Indexing in MongoDB allows for faster data retrieval by creating a searchable structure on selected attributes, optimizing query performance.
- **Replication and high availability**: MongoDB's replica sets ensure data redundancy by maintaining multiple copies of the data, providing fault tolerance and continuous availability even in case of server failures.
- **Auto-Sharding**: Auto-sharding in MongoDB automatically distributes data across multiple servers, enabling horizontal scaling and efficient handling of large datasets.
- **Big Data and Real-time Application**: When dealing with massive datasets or applications requiring real-time data updates, MongoDB's flexibility and scalability prove advantageous.
- **Rich queries**: MongoDB supports complex queries with a variety of operators, allowing you to retrieve, filter, and manipulate data in a flexible and powerful manner.
- **Fast in-place updates**: MongoDB efficiently updates documents directly in their place, minimizing data movement and reducing write overhead.
- **Professional support by MongoDB**: MongoDB offers expert technical support and resources to help users with any issues or challenges they may encounter during their database operations.

Internet of Things (IoT) Applications: **Storing and analyzing sensor data with its diverse formats often aligns well with MongoDB's document structure.**

Where do we use MongoDB?

MongoDB is preferred over RDBMS in the following scenarios:

- **Big Data**: If you have huge amount of data to be stored in tables, think of MongoDB before RDBMS databases. MongoDB has built-in solution for partitioning and sharding your database.
- **Unstable Schema**: Adding a new column in RDBMS is hard whereas MongoDB is schema-less. Adding a new field does not effect old documents and will be very easy.
- **Distributed data** Since multiple copies of data are stored across different servers, recovery of data is instant and safe even if there is a hardware failure.

Language Support by MongoDB:

MongoDB currently provides official driver support for all popular programming languages like C, C++, Rust, C#, Java, Node.js, Perl, PHP, Python, Ruby, Scala, Go, and Erlang.

5. Features of MongoDB –

- **Schema-less Database:** It is the great feature provided by the MongoDB. A Schema-less database means one collection can hold different types of documents in it. Or in other words, in the MongoDB database, a single collection can hold multiple documents and these documents may consist of the different numbers of fields, content, and size. It is not

necessary that the one document is similar to another document like in the relational databases. Due to this cool feature, MongoDB provides great flexibility to databases.

- **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.
- **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.
- **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.
- **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.
- **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. It provides three different aggregations i.e, aggregation pipeline, map-reduce function, and single-purpose aggregation methods
- **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.

Advantages of MongoDB :

- It is a schema-less NoSQL database. You need not to design the schema of the database when you are working with MongoDB.
- It does not support join operation.
- It provides great flexibility to the fields in the documents.
- It contains heterogeneous data.
- It provides high performance, availability, scalability.
- It supports Geospatial efficiently.
- It is a document oriented database and the data is stored in BSON documents.
- It also supports multiple document ACID transition(string from MongoDB 4.0).
- It does not require any SQL injection.
- It is easily integrated with Big Data Hadoop

Disadvantages of MongoDB :

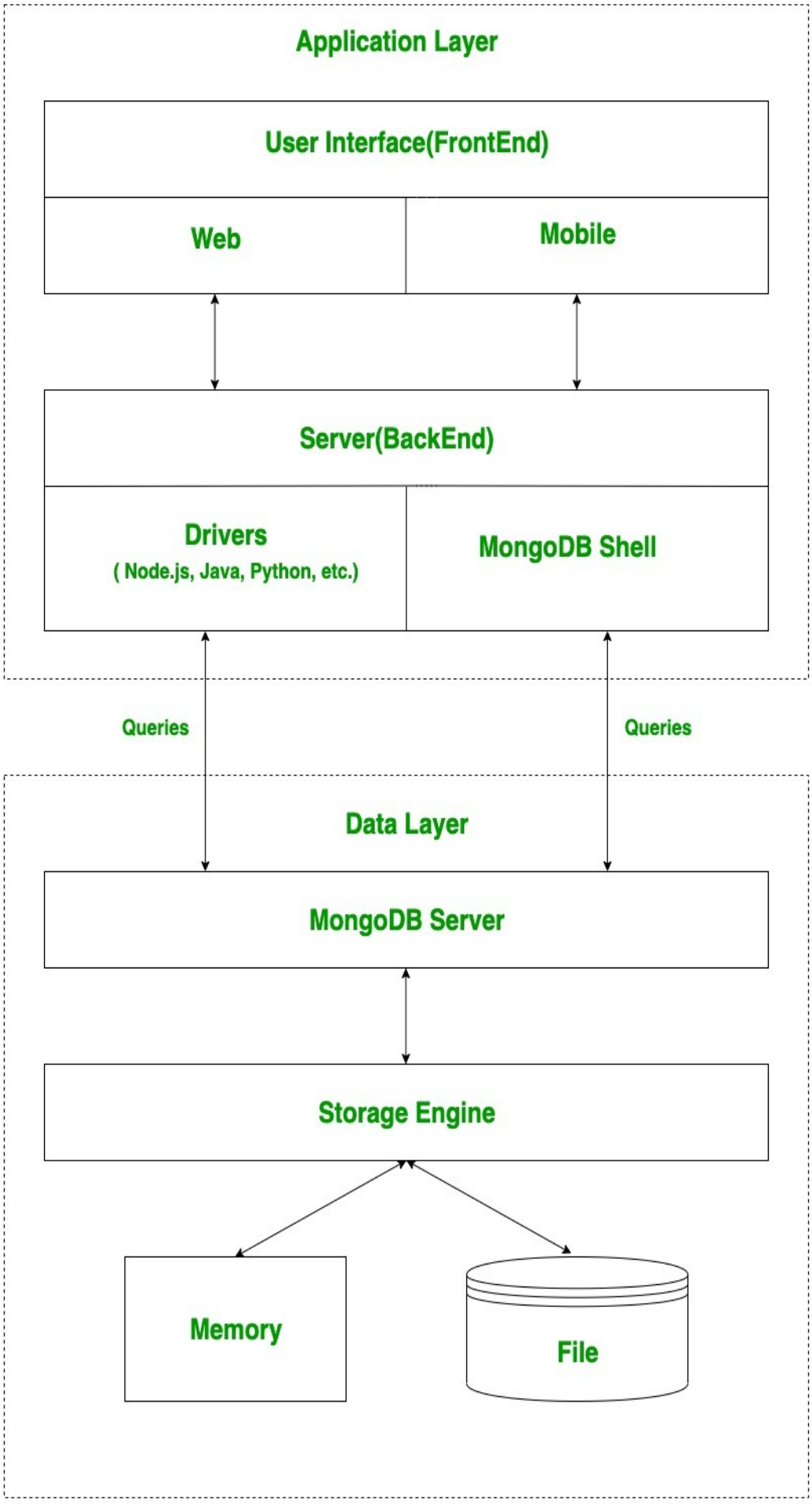
- It uses high memory for data storage.
- You are not allowed to store more than 16MB data in the documents.
- The nesting of data in BSON is also limited you are not allowed to nest data more than 100 levels.

6. How MongoDB works ?

MongoDB is an open-source document-oriented database. It is used to store a larger amount of data and also allows you to work with that data. MongoDB is not based on the table-like relational database structure but provides an altogether different mechanism for storage and retrieval of data, that's why known as NoSQL database. Here, the term 'NoSQL' means 'non-relational'. The format of storage is called BSON (similar to JSON format).
Now, let's see how actually this MongoDB works? But before proceeding to its working, first, let's discuss some important parts of MongoDB –

- **Drivers:** Drivers are present on your server that are used to communicate with MongoDB. The drivers support by the MongoDB are C, C++, C#, and .Net, Go, Java, Node.js, Perl, PHP, Python, Motor, Ruby, Scala, Swift, Mongoid.
- **MongoDB Shell:** MongoDB Shell or mongo shell is an interactive JavaScript interface for MongoDB. It is used for queries, data updates, and it also performs administrative operations.
- **Storage Engine:** It is an important part of MongoDB which is generally used to manage how data is stored in the memory and on the disk. MongoDB can have multiple search engines. You are allowed to use your own search engine and if you don't want to use your own search engine you can use the default search engine, known as *WiredTiger Storage Engine* which is an excellent storage engine, it efficiently works with your data like reading, writing, etc.

Working of MongoDB –The following image shows how the MongoDB works:



MongoDB work in two layers –

- **Application Layer** and
- **Data layer**

Application Layer is also known as the **Final Abstraction Layer**, it has two-parts, first is a **Frontend (User Interface)** and the second is **Backend (server)**. The frontend is the place where the user uses MongoDB with the help of a Web or Mobile. This web and mobile include web pages, mobile applications, android default applications, IOS applications, etc. The backend contains a server which is used to perform server-side logic and also contain drivers or mongo shell to interact with MongoDB server with the help of queries.
These queries are sent to the MongoDB server present in the **Data Layer**. Now, the MongoDB server receives the queries and passes the received queries to the storage engine. MongoDB server itself does not directly read or write the data to the files or disk or memory. After passing the received queries to the storage engine, the storage engine is responsible to read or write the data in the files or memory basically it manages the data.

MongoDB, the most popular NoSQL database, is an open-source document-oriented database. The term 'NoSQL' means 'non-relational'. It means that MongoDB isn't based on the table-like relational database structure but provides an altogether different mechanism for storage and retrieval of data. This format of storage is called BSON (similar to JSON format).
A simple MongoDB document Structure:

```
{
 title: 'Geeksforgeeks',
 by: 'Harshit Gupta',
 url: 'https://www.geeksforgeeks.org',
 type: 'NoSQL'
}
```

SQL databases store data in tabular format. This data is stored in a predefined data model which is not very much flexible for today's real-world highly growing applications. **Modern applications are more networked, social and interactive than ever**. Applications are storing more and more data and are accessing it at higher rates.
Relational Database Management System(RDBMS) i**s not the correct choice when it comes to handling big data by the virtue of their design since they are not horizontally scalable**. If the database runs on a single server, then it will reach a scaling limit. NoSQL databases are more scalable and provide superior performance. MongoDB is such a NoSQL database that scales by adding more and more servers and increases productivity with its flexible document model.

Getting Started

After you install MongoDB, you can see all the installed file inside C:\ProgramFiles\MongoDB\ (default location). In the C:\Program Files\MongoDB\Server\3.2\bin directory, there are a bunch of executables and a short-description about them would be:

mongo: The Command Line Interface to interact with the db.
mongod: This is the database. Sets up the server.
mongodump: It dumps out the Binary of the Database(BSON)
mongoexport: Exports the document to Json, CSV format
mongoimport: To import some data into the DB.
mongorestore: to restore anything that you've exported.
mongostat: Statistics of databases

7. Database, Collection and Documents:-

Database

- Database is a container for collections.
- Each database gets its own set of files.
- A single MongoDB server can has multiple databases.

Collection

- Collection is a group of documents.
- Collection is equivalent to RDBMS table.
- A collection consist inside a single database.
- Collections do not enforce a schema.
- A Collection can have different fields within a Documents.
-

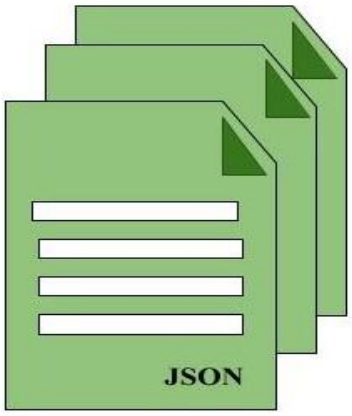
Document:-

A document database has information retrieved or stored in the form of a document or other words semi-structured database. Since they are non-relational, so they are often referred to as NoSQL data.

The document database fetches and accumulates data in forms of key-value pairs but here, the values are called as Documents. A document can be stated as a complex data structure. Document here can be a form of text, arrays, strings, JSON, XML, or any such format. The use of nested documents is also very common. It is very effective as most of the data created is usually in the form of JSON and is unstructured.

C1	C2	C3

Relational Data Model



Document Store Model

Consider the below example that shows a sample database stored in both Relational and Document Database

RELATIONAL

ID	first_name	last_name	cell	city	year_of_birth	location_x	location_y
1	'Mary'	'Jones'	'516-555-2048'	'Long Island'	1986	'-73.9876'	'40.7574'

ID	user_id	profession
10	1	'Developer'
11	1	'Engineer'

ID	user_id	name	version
20	1	'MyApp'	1.0.4
21	1	'DocFinder'	2.5.7

ID	user_id	make	year
30	1	'Bentley'	1973
31	1	'Rolls Royce'	1965

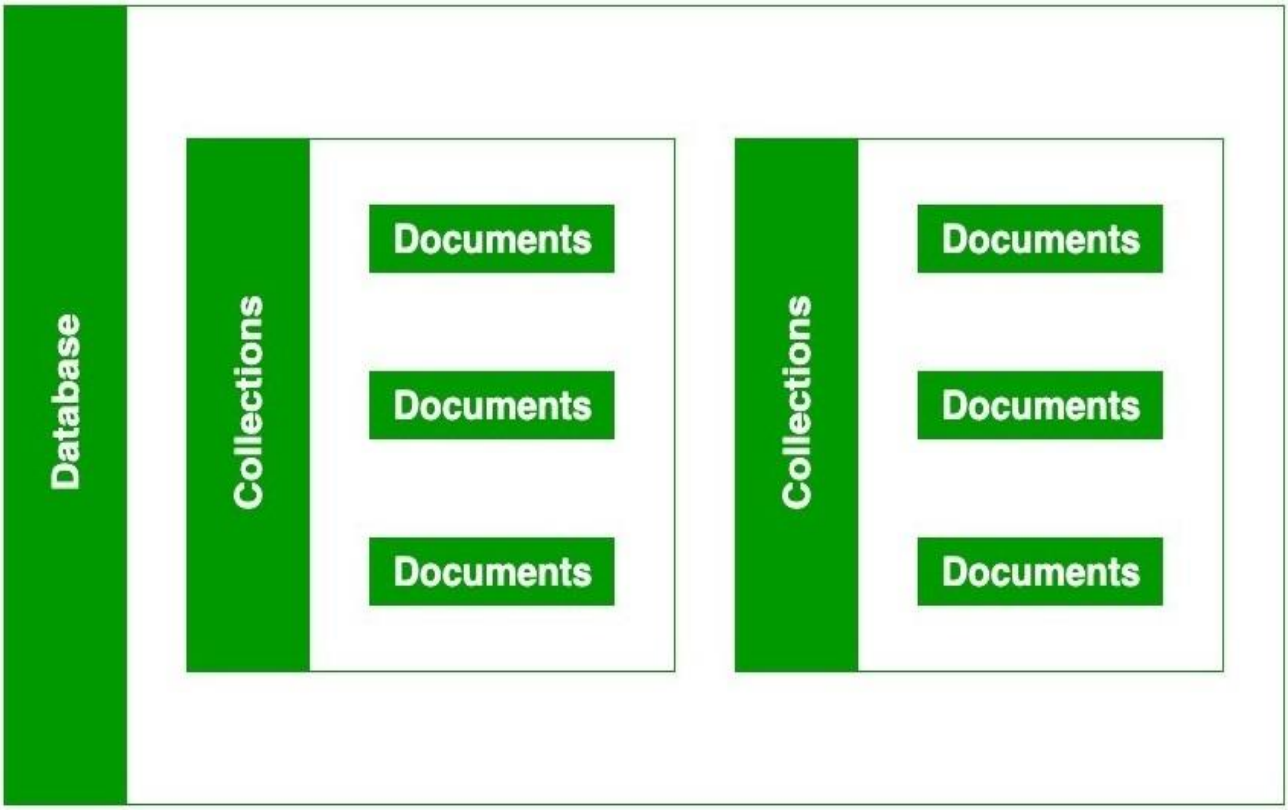
DOCUMENT

```
first_name: "Mary",
last_name: "Jones",
cell: "516-555-2048",
city: "Long Island",
year_of_birth: 1986,
location: {
        type: "Point",
        coordinates: [-73.9876, 40.7574]
},
profession: ["Developer", "Engineer"],
apps: [
 { name: "MyApp",
   version: 1.0.4 },
 { name: "DocFinder",
   version: 2.5.7 }
],
cars: [
   { make: "Bentley",
     year: 1973 },
   { make: "Rolls Royce",
     year: 1965 }
 ]
```

How it works ?

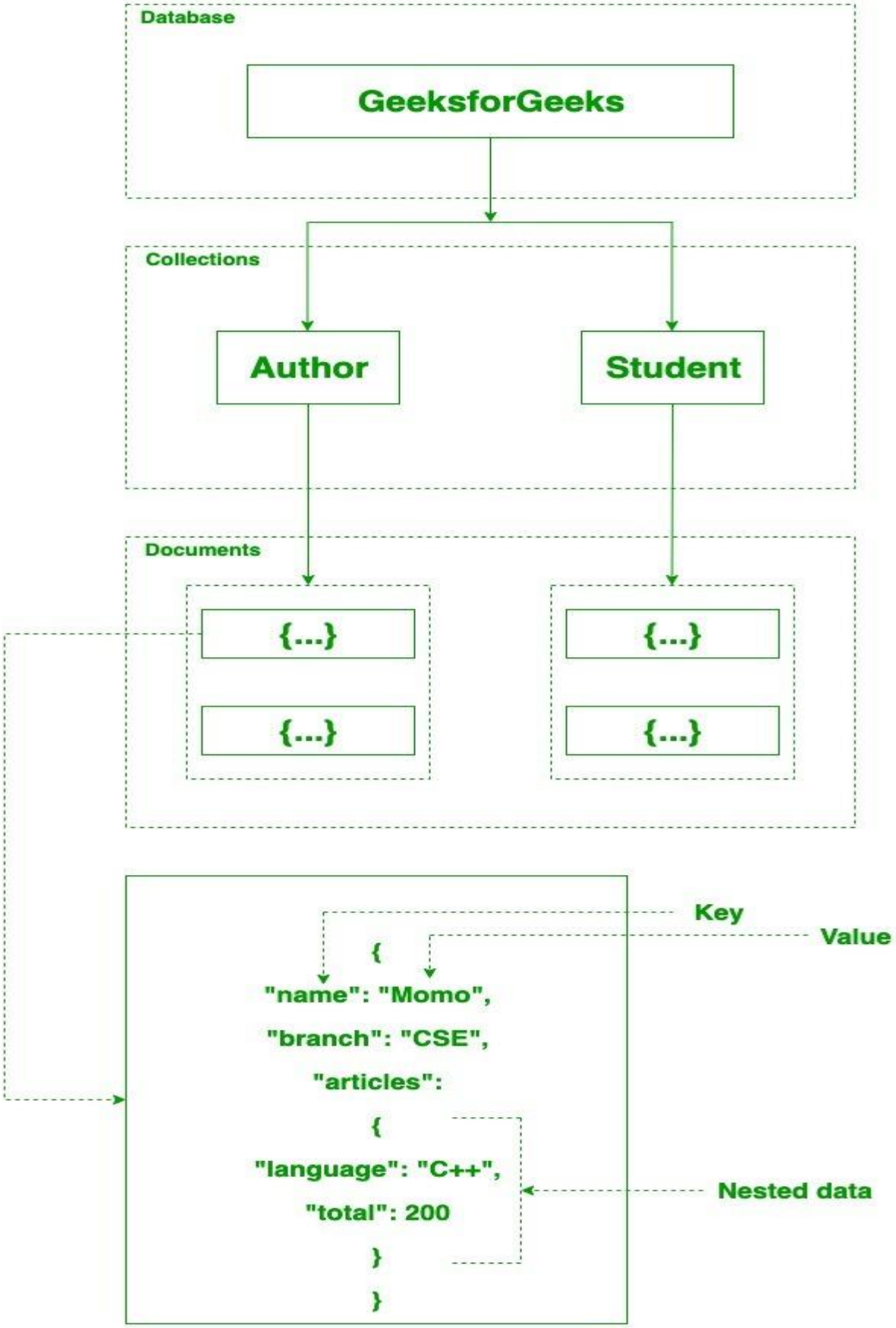
Now, we will see how actually thing happens behind the scene. As we know that MongoDB is a database server and the data is stored in these databases. Or in other words, MongoDB environment gives you a server that you can start and then create multiple databases on it using MongoDB.

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:



- The MongoDB database contains collections just like the MYSQL database contains tables. You are allowed to create multiple databases and multiple collections.
- Now inside of the collection we have documents. These documents contain the data we want to store in the MongoDB database and a single collection can contain multiple documents and you are schema-less means it is not necessary that one document is similar to another.
- The documents are created using the fields. Fields are key-value pairs in the documents, it is just like columns in the relation database. The value of the fields can be of any BSON data types like double, string, boolean, etc.
- The data stored in the MongoDB is in the format of BSON documents. Here, BSON stands for Binary representation of JSON documents. Or in other words, in the backend, the MongoDB server converts the JSON data into a binary form that is known as BSON and this BSON is stored and queried more efficiently.
- In MongoDB documents, you are allowed to store nested data. This nesting of data allows you to create complex relations between data and store them in the same document which makes the working and fetching of data extremely efficient as compared to SQL. In SQL, you need to write complex joins to get the data from table 1 and table 2. The maximum size of the BSON document is 16MB.

NOTE: In MongoDB server, you are allowed to run multiple databases.
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:


Database
GeeksforGeeks
Collections
Author
Student
Documents
{...}
{...}
{...}
{...}
Key
Value
{
"name": "Momo",
"branch": "CSE",
"articles":
{
"language": "C++",
"total": 200
}
}
Nested data

How mongoDB is different from RDBMS ?
Some major differences in between MongoDB and the RDBMS are as follows:

MongoDB	RDBMS
It is a non-relational and document-oriented database.	**It is a relational database.**
It is suitable for hierarchical data storage.	**It is not suitable for hierarchical data storage.**
It has a dynamic schema.	**It has a predefined schema.**
It centers around the CAP theorem (Consistency, Availability, and Partition tolerance).	**It centers around ACID properties (Atomicity, Consistency, Isolation, and Durability).**
In terms of performance, it is much faster than RDBMS.	**In terms of performance, it is slower than MongoDB.**

1.Install MongoDB

- **There are 3 ways to install and use MongoDB**
  1. Community Server(free and open source. after download use local system)
  2. Visual Studio Extension
  3. MongoDB Atlas(cloud hosted DB offered by MONGO DB company)

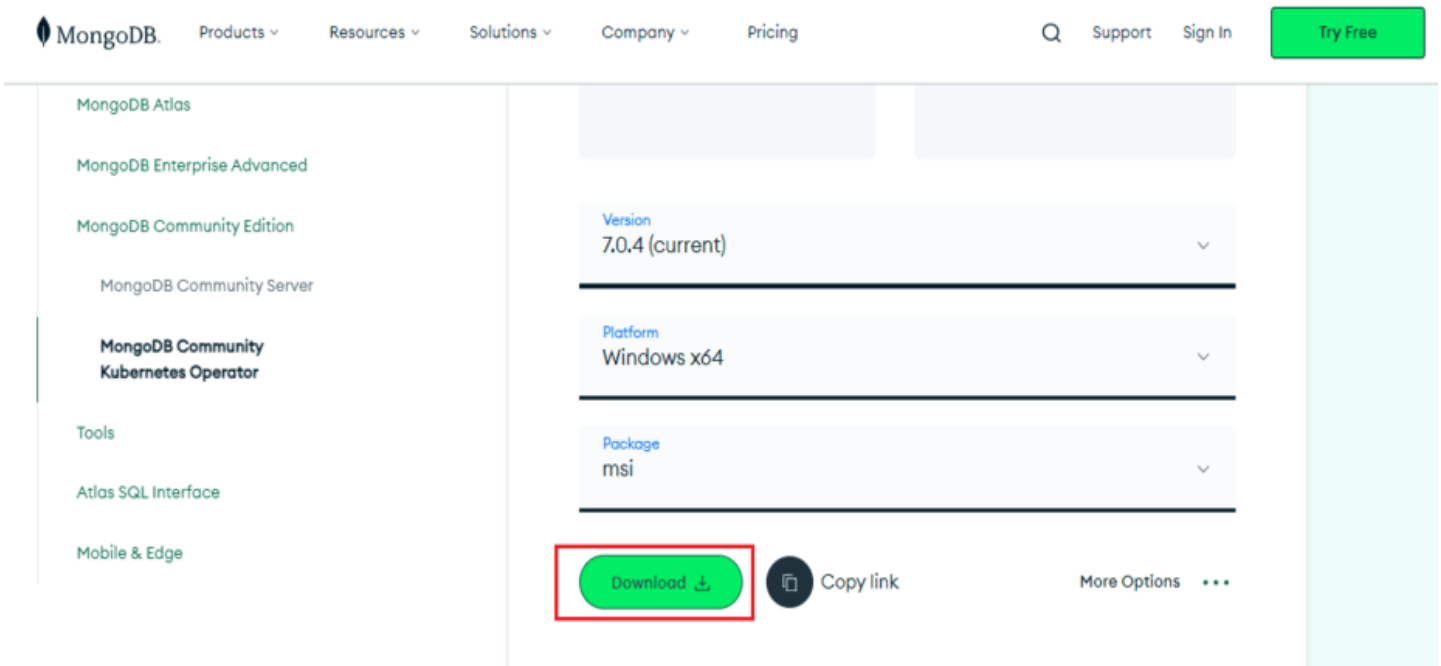
1. **Let's install MongoDB on our machines(Windows)**
   - Visit official website: http://mongodb.com
   - Download the latest stable version from Community Server
   - The Community server will also install the following apps
     a. Community Server
     b. Compass-GUI Tool for MongoDB

Install MongoDB on Windows using MSI
Requirements to Install MongoDB on Windows

- **MongoDB 4.4** and later only **support 64-bit versions** of Windows.
- MongoDB 7.0 Community Edition supports the following **64-bit versions** of Windows on x86_64 architecture:
  - **Windows Server 2022**
  - **Windows Server 2019**
  - **Windows 11**

To install MongoDB on windows, first, download the MongoDB server and then install the MongoDB shell. The Steps below explain the installation process in detail and provide the required resources for the smooth **download and install MongoDB**.

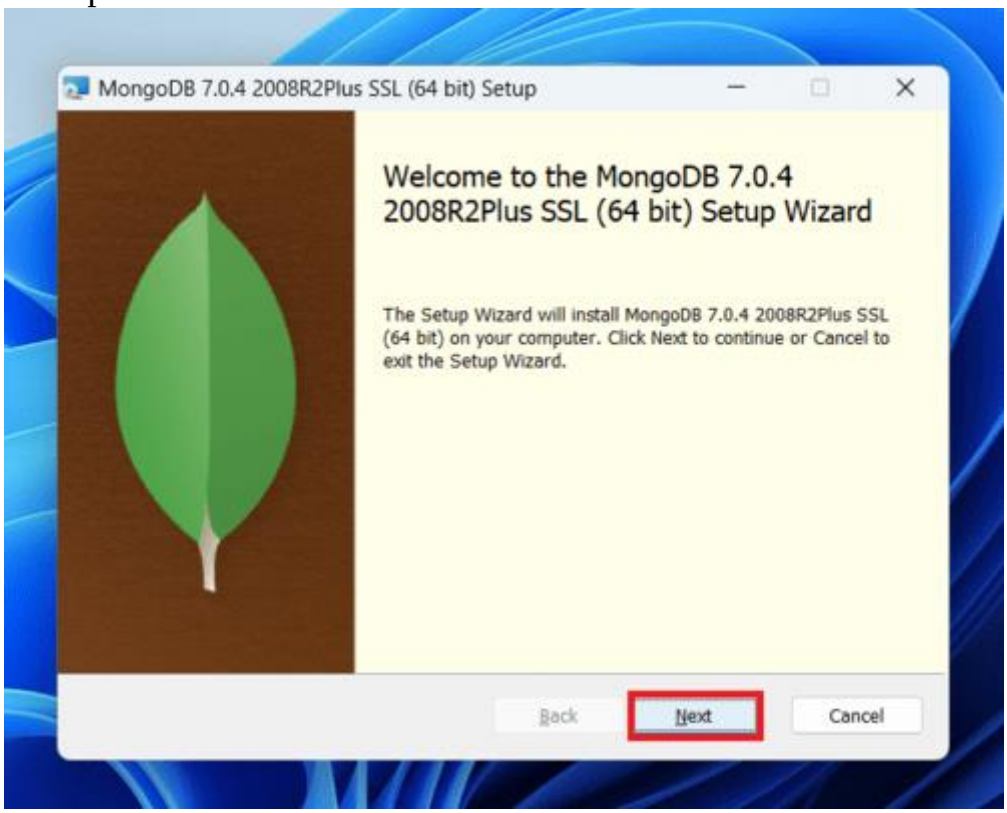
Step 1: Go to the MongoDB Download Center to download the MongoDB Community Server.



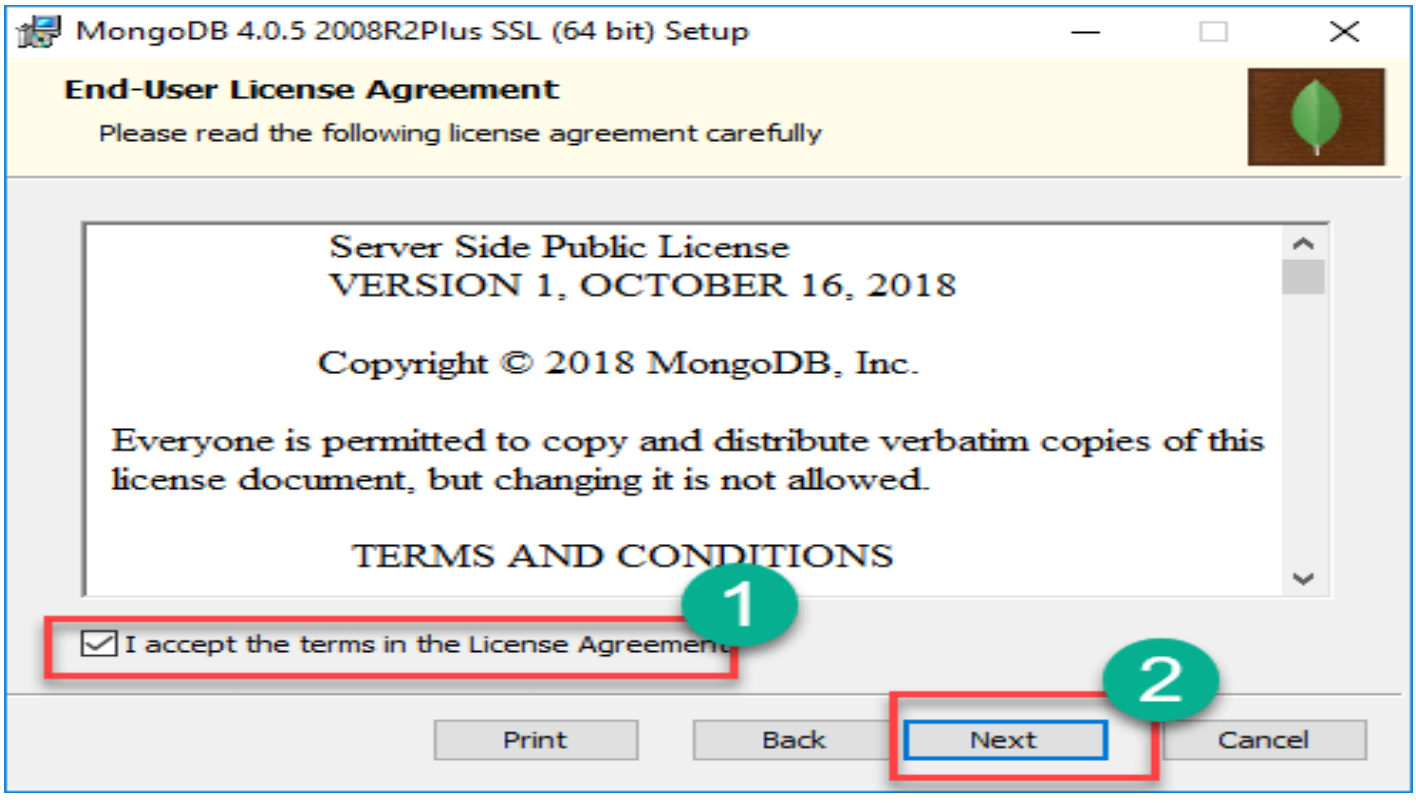
Here, You can select any version, Windows, and package according to your requirement. For Windows, we need to choose:

- Version: 7.0.4
- OS: Windows x64
- Package: msi

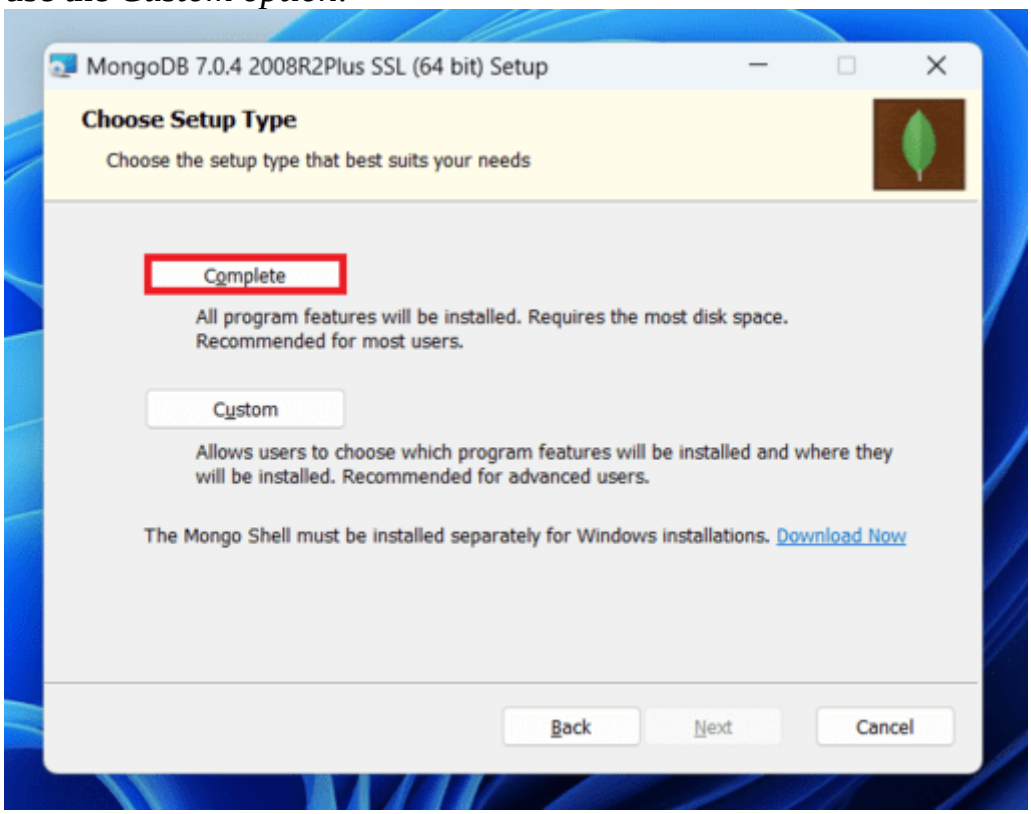
Step 2: When the download is complete open the msi file and click the *next button* in the startup screen:



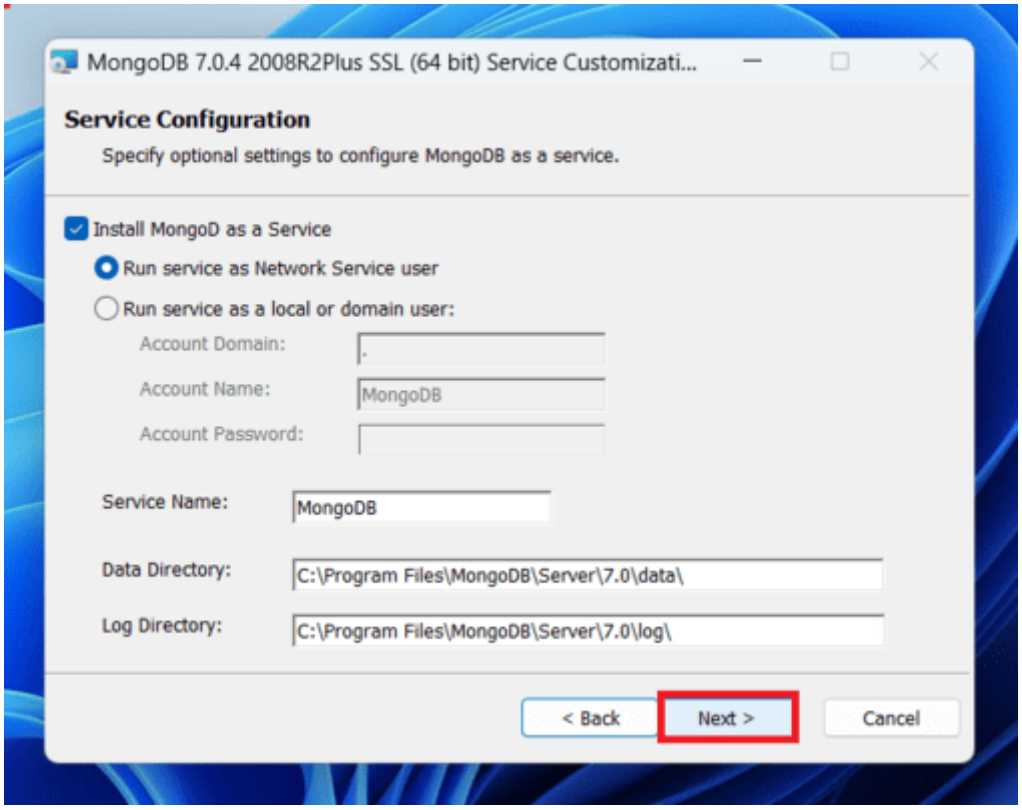
Step 3: Now accept the End-User License Agreement and click the next button:



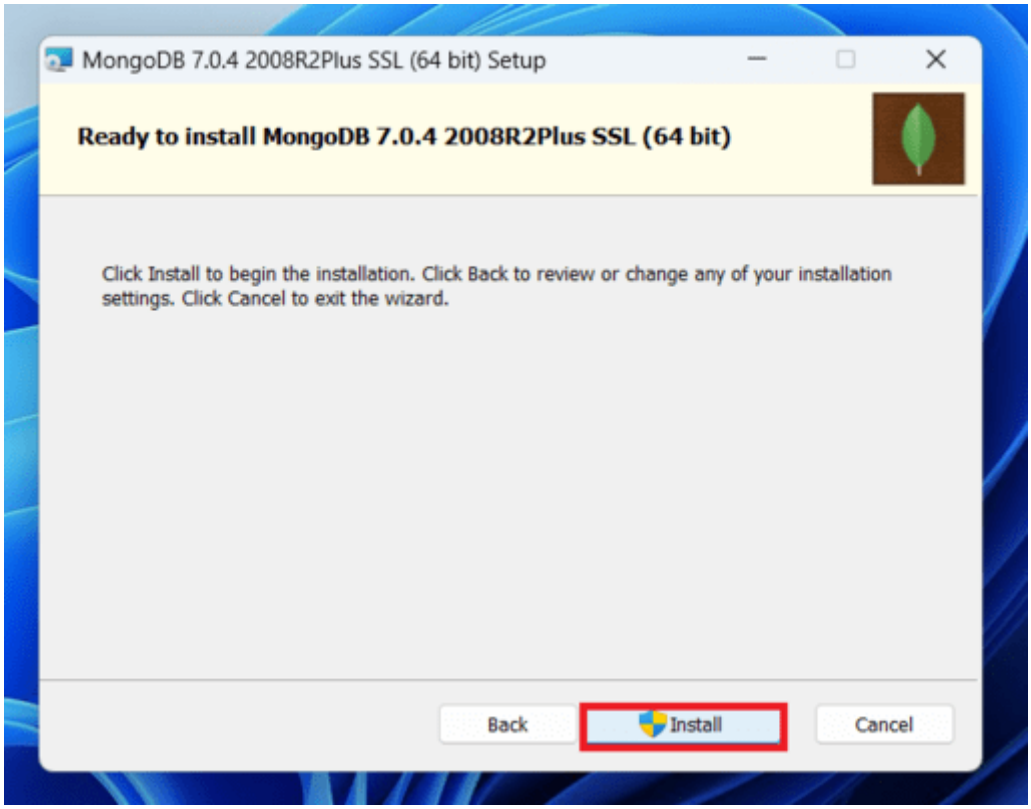
Step 4: Now select the *complete option* to install all the program features. Here, if you can want to install only selected program features and want to select the location of the installation, then use the *Custom option:*



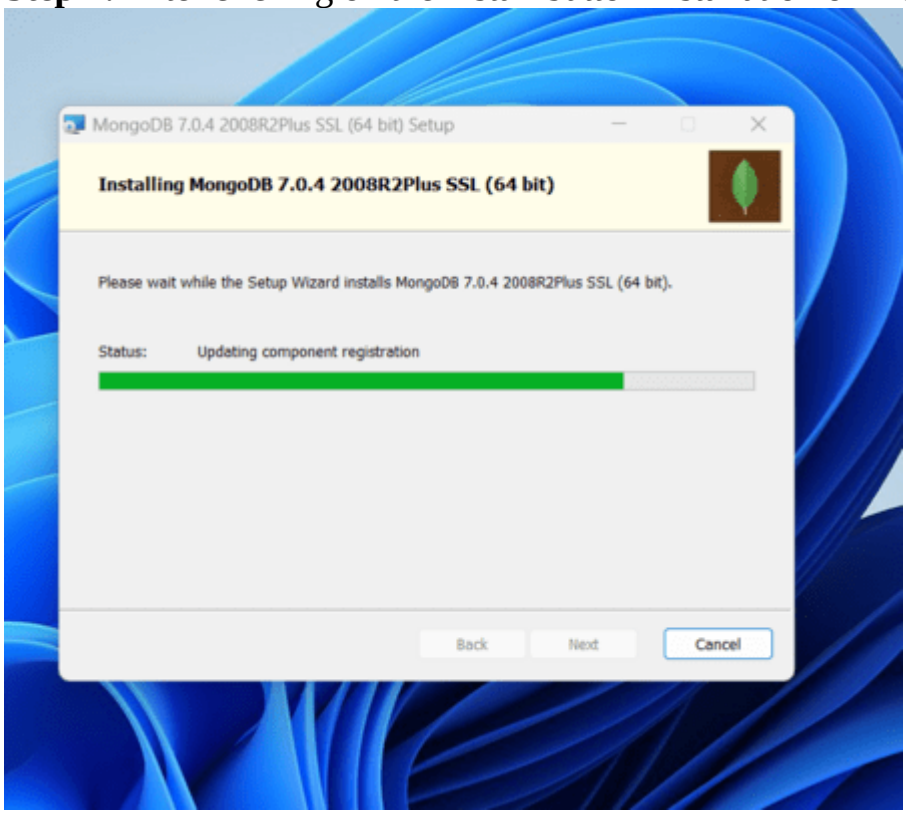
Step 5: Select "Run service as Network Service user" and copy the path of the data directory. Click Next:



Step 6: Click the *Install button* to start the MongoDB installation process:

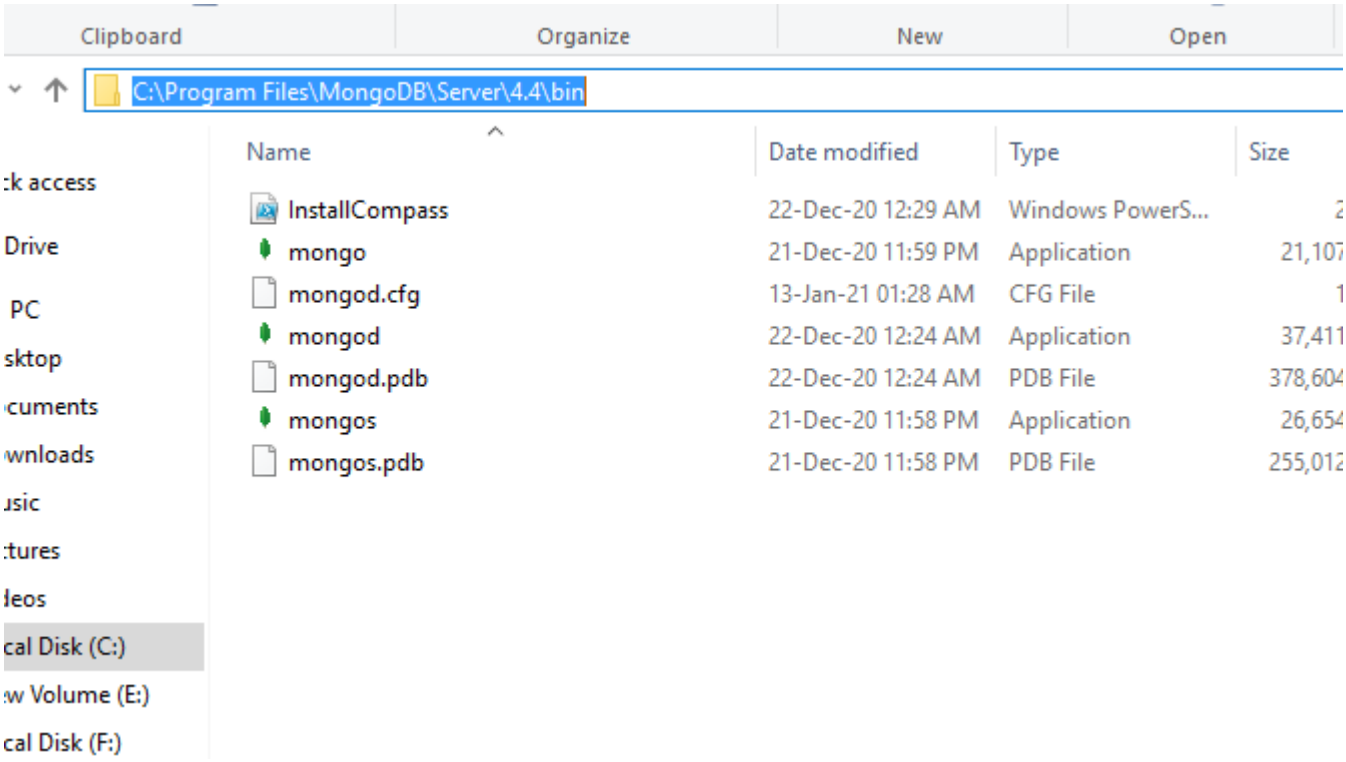


Step 7: After clicking on the install button installation of MongoDB begins:

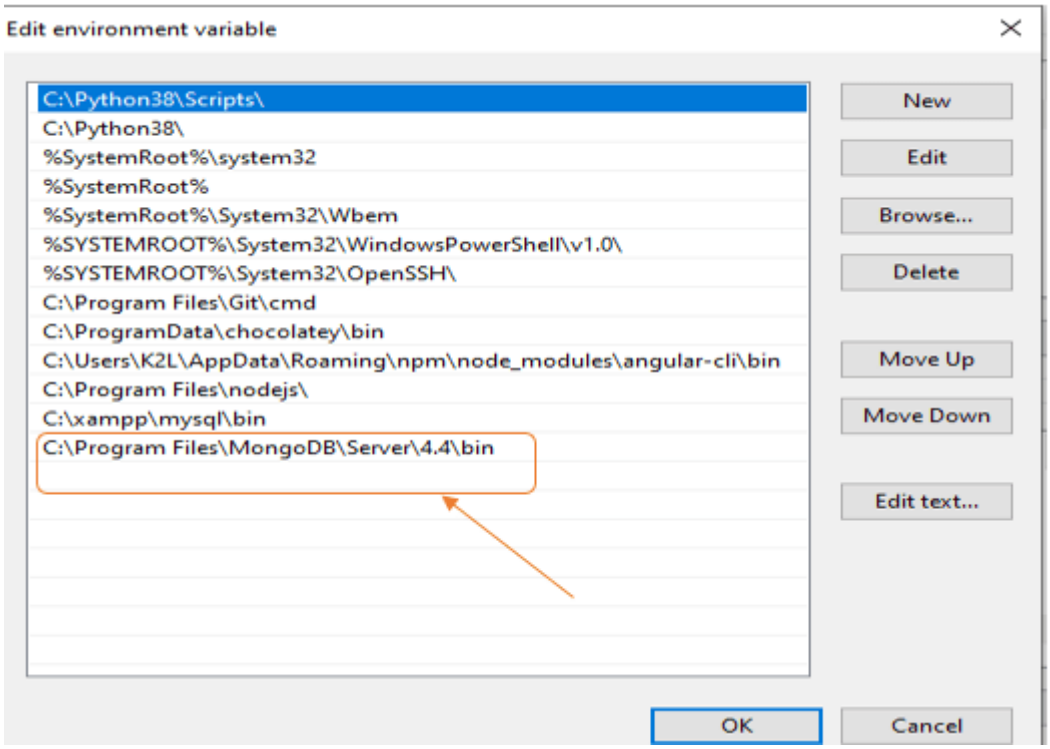


Step 8: Now click the ***Finish button*** to complete the MongoDB installation process:

Step 9: Now we go to the location where MongoDB installed in step 5 in your system and copy the bin path:



Step 10: Now, to create an environment variable open system properties << Environment Variable << System variable << path << Edit Environment variable and paste the copied link to your environment system and click Ok:



Step 11: 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 12: Now, Open C drive and create a folder named "data" inside this folder create another folder named "db". 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 mongo Shell

Step 13: Now we are going to connect our server (mongod) with the mongo shell. So, keep that mongod window and open a new command prompt window and write **mongo.** Now, our mongo shell will successfully connect to the mongod.

Important Point: 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()
---
>
```

Now, you are ready to write queries in the mongo Shell.

Run MongoDB

Now you can make a new database, collections, and documents in your shell. Below is an example of how to make a new database:

The *use Database_name* command makes a new database in the system if it does not exist, if the database exists it uses that database:

```
use gfg
```

Now your database is ready of name gfg.

The db.Collection_name command makes a new collection in the gfg database and the insertOne() method inserts the document in the student collection:

```
db.student.insertOne({Akshay:500})
```

```
> use gfg
switched to db gfg
> db.student.insertOne({Akshay:500})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("60083bf8b7388ed4d54157c9")
}
> db.student.find().pretty()
{ "_id" : ObjectId("60083bf8b7388ed4d54157c9"), "Akshay" : 500 }
>
```

2. **MongoDB – Visual Studio Extension**
    - Search and install MongoDB Visual Studio Code – Extension

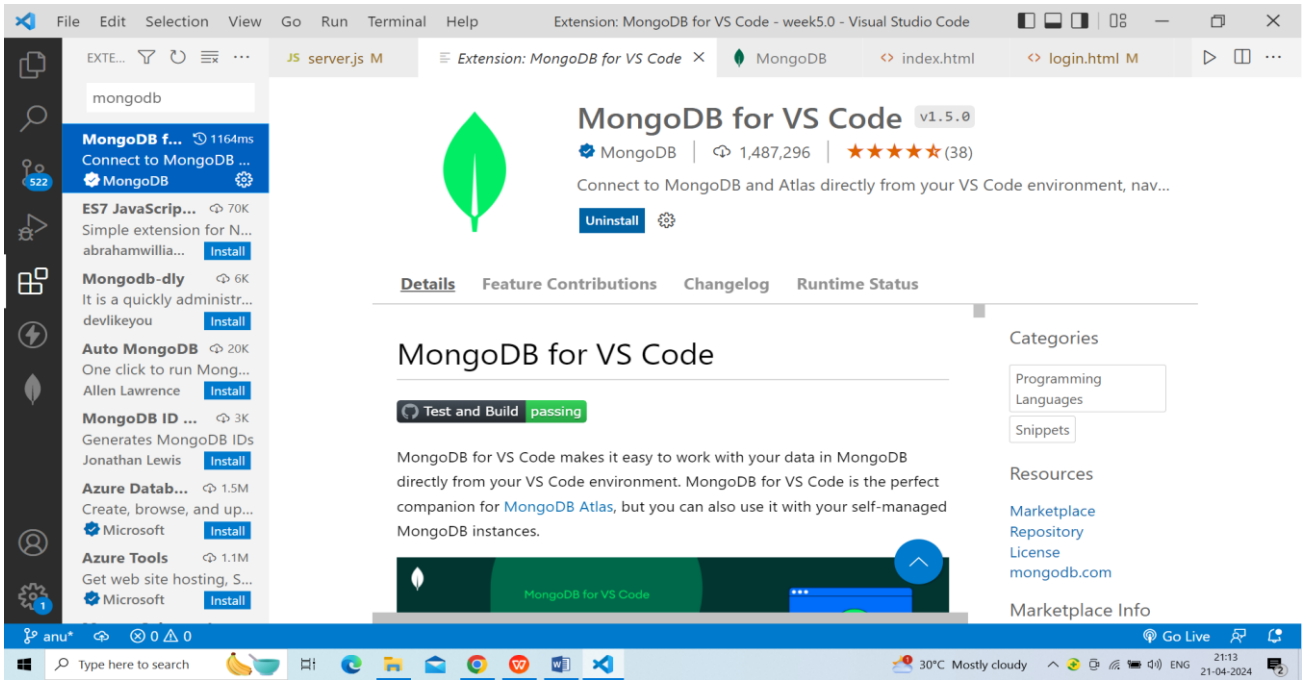


Fig.1. **MongoDB – Visual Studio Extension**

1. **MongoDB – Atlas**
    - Cloud-Hosted and Fully Managed MongoDB
    - Pay as you go model
    - Very cost-effective
    - Fully secured and reliable

2. Data Modeling

Definition:-

Data modelling refers to the organization of data within a database and the links between related entities. Data in MongoDB has a **flexible schema model**, which means:

- Documents within a single collection are not required to have the same set of fields.
- A field's data type can differ between documents within a collection.

The primary problem in data modeling is balancing application needs, **database engine performance** features, and **data retrieval patterns**. Always consider the application uses of the data (i.e. **queries**, **updates**, and **data processing**) as well as the fundamental design of the data itself when creating data models.

➢ Advantages Of Data Modelling

Data modelling is essential for a successful application, even though at first it might just seem like one more step. In addition to increasing **overall efficiency** and **improving development cycles**, data modelling helps you better understand the data at hand and identify future business requirements, which can save time and money. In particular, applying suitable data models:

- Improves application performance through better database strategy, design, and implementation.
- Allows faster application development by making object mapping easier.
- Helps with better data learning, standards, and validation.
- Allows organizations to assess long-term solutions and model data while solving not just current projects but also future application requirements, including maintenance.

Different Types of Data Models

The three types of data models that are typically classified as follows:

1. Conceptual data model

Conceptual Data Models are **rough sketches** that provide the big picture, detailing where data/information from various business processes will be stored in the database system and the relationships they will be involved with. A conceptual data model typically includes the **entity class**, **attributes**, **constraints**, and the relationship between security and data integrity requirements.

This model describes the types of data that should be in the system and how they relate to one another. This model, which is typically developed with the support of the business stakeholders, it contains the business logic of the application, often involves **domain-driven design** (DDD) principles, and serves as the foundation for one or more of the following models. The primary purpose of the conceptual model is to identify the **information that will be essential to an organization**.

2. Logical data model

Logical data models provide more detailed, **subjective information about data set relationships**. At this stage, we can clearly connect what data types and relations are used. Logical data models are generally missed in rapid business contexts, having their utility in **data-driven** initiatives requiring important procedure execution.

The logical data model specifies **how data will be organized**. The relationship between entities is established at a high level .In this model, and a list of entity properties is also provided. This data model can be viewed as a "**blueprint**" for the data that will be used.

3. Physical data model

The schema/layout for data storage routines within a database is defined by the **physical data model**. A physical data model is a ready-to-implement plan that can be stored in a **relational database**.

The physical data model is a representation of **how data will be stored in a particular database** management system (DBMS). In this approach, main and secondary keys in a relational database are defined, or the decision to include or connect data in a document database such as MongoDB based on entity relationships is made. This is also where you will define the **data types for each of your fields**, which will create the database structure.

➢ Data Model Design (or) Types

For modelling data in MongoDB, two strategies are available. These strategies are different and it is recommended to analyze our scenario for a better flow. The two methods are as follows:

1. Embedded Data Model
2. Normalized Data Model

1. Embedded Data Model

This method, also known as the **de-normalized** data model, allows you to have (embed) all of the **related data in a single document**.

For example, if we obtain student information in three different documents, Personal_details, Contact, and Address, we can embed all three in a single one, as shown below.

```
{
   _id: ,
   Std_ID: "987STD001"
   Personal_details:{
      First_Name: "Rashmika",
      Last_Name: "Sharma",
      Date_Of_Birth: "1999-08-26"
   },
   Contact: {
      e-mail: "rashmika_sharma.123@gmail.com",
      phone: "9987645673"
   },
   Address: {
      city: "Karnataka",
      Area: "BTM2ndStage",
      State: "Bengaluru"
   }
}
```

2. Normalized Data Model (or) Reference Data Model:

In a normalized data model, object references are used to express the **relationships between documents and data objects**. Because this approach **reduces data duplication**, it is relatively simple to document **many-to-many relationships** without having to repeat content. Normalized data models are the most effective technique to model large **hierarchical data** with cross-collection relationships.

Student:

```
{
   _id: <StudentId101>,
   Std_ID: "10025AE336"
}
```

Personal_Details:

```
{
   _id: <StudentId102>,
   stdDocID: " StudentId101",
   First_Name: "Rashmika",
   Last_Name: "Sharma",
   Date_Of_Birth: "1999-08-26"
}
```

Contact:

```
{
   _id: <StudentId103>,
   stdDocID: " StudentId101",
   e-mail: "rashmika_sharma.123@gmail.com",
   phone: "9987645673"
}
```

Address:

```
{
   _id: <StudentId104>,
   stdDocID: " StudentId101",
   city: "Karnataka",
   Area: "BTM2ndStage",
   State: "Bengaluru"
}
```

Considerations while designing Schema in MongoDB

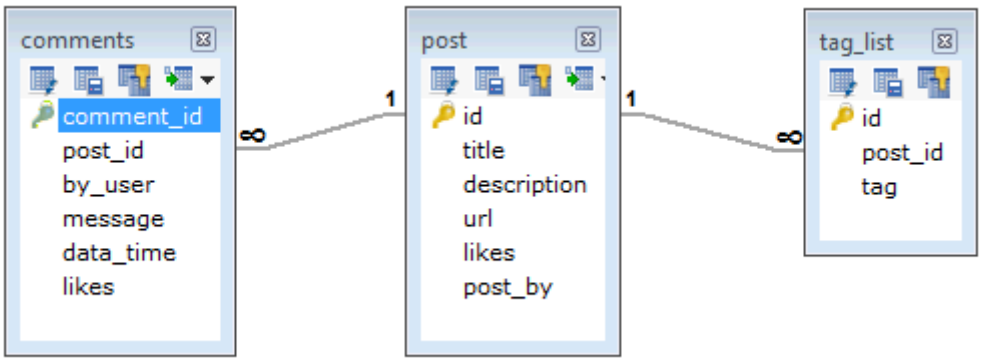
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.
- Do complex aggregation in the schema.

Example

Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

- Every post has the unique title, description and url.
- Every post can have one or more tags.
- Every post has the name of its publisher and total number of likes.
- Every post has comments given by users along with their name, message, data-time and likes.
- On each post, there can be zero or more comments.

In RDBMS schema, design for above requirements will have minimum three tables.



While in MongoDB schema, design will have one collection post and the following structure −

```
{
  _id: POST_ID
  title: TITLE_OF_POST,
  description: POST_DESCRIPTION,
  by: POST_BY,
  url: URL_OF_POST,
  tags: [TAG1, TAG2, TAG3],
  likes: TOTAL_LIKES,
  comments: [
    {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    },
    {
      user:'COMMENT_BY',
      message: TEXT,
      dateCreated: DATE_TIME,
      like: LIKES
    }
  ]
```

```
}
```

So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

➢ Connect MongoDB:-

Mongodb compass

- Connect with compass app
- Understand basics of compass app
- Get your hands-on examples with compass

Hostname:-Localhost

Port:-27017

Visual Studio:-

- Connect with Visual Studio Code Extension
- Understand basics of visual Studio Code Extension
- Get your hands-on examples with Visual Studio Code Extension

Connect:- mongodb://localhost:27017

Shell:**mongosh**

(Or)

Mongod creating server and **mongo** for shell.

➢ CURD Operation:-

Creating and drop database:-

- use anu;//creating
- show dbs;//display all db
- db//current db
- db.dropDatabase();//deleting db

Creating and drop collections:-

Syntax:-

- db.createCollection(name,options)//creating
- db.collection.drop()
- db.collection.insertOne({key:”value”})

Ex:-

- db.createCollection(“products”);
- db.products.drop()

Inserting Documents into Collections:-

Syntax:-

- db.collection_name.insert({“name”:”aaa”})//one
- db.collection_name.insertMany([{“name”:”aaa”}, {“name”:”bbb”}])//many
- **Example:-** db.aaa.insert({“name”:”aaa”})//one
- db.bbb.insertMany([{"name":"aaa"}, {"name":"bbb"}])//many

Update:-

- db.bbb.update({"name":"bbb"},{$set:{"name":"ccc","isActive":true}});

Read:-

- db.bbb.find();
- db.bbb.findOne();
- db.bbb.find({"name":"ccc"});
- db.bbb.findOneAndReplace({"name":"ccc"},{"name":"eee"});
- db.bbb.findOneAndDelete({"name":"eee"});

Delete:-

db.orders.deleteOne({"name":"aaa"})

/*

Db.student.insertOne({name:”anusha”})

Db.student.find().pretty()

*/

3. Query and Projection

MongoDB Query

MongoDB Query Operators

Similar to **SQL** MongoDB have also some operators to operate on data in the collection. MongoDB query operators check the conditions for the given data and **logically** compare the data with the help of two or more fields in the document.

Query operators help to filter data based on specific conditions. **E.g.**, $eq,$and,$exists, etc.

MongoDB provides the function names as *db.collection_name.find()* to operate query operation on database.

Syntax:

db.collection_name.find()

Example:

db.article.find()

Types of Query Operators in MongoDB

The Query operators in MongoDB can be further classified into 8 more types. The 8 types of Query Operators in MongoDB are:

1. Comparison Operators
2. Logical Operators
3. Array Operators
4. Evaluation Operators
5. Element Operators
6. Bitwise Operators
7. Geospatial Operators
8. Comment Operators

1. Comparison Operators

The comparison operators in MongoDB are used to perform value-based comparisons in queries. The comparison operators in the MongoDB are shown as below:

Comparison Operator	Description	Syntax
$eq	Matches values that are equal to a specified value.	{ field: { $eq: value } }
$ne	Matches all values that are not equal to a specified value.	{ field: { $ne: value } }
$lt	Matches values that are less than a specified value.	{ field: { $lt: value } }
$gt	Matches values that are greater than a specified value.	{ field: { $gt: value } }
$lte	Matches values that are less than or equal to a specified value.	{ field: { $lte: value } }
$gte	Matches values that are greater than or equal to a specified value.	{ field: { $gte: value } }
$in	Matches any of the values specified in an array.	{ field: { $in: [<value1>, <value2>, ...] } }

Documents:

db.books.insertMany([{"p_name":"book","price":50},{"p_name":"pen","price":100},{"p_name":"pencilbox","price":500},{"p_name":"ball","price":200}]);

MongoDB Comparison Operators

1. $eq

The $eq specifies the equality condition. It matches documents where the value of a field equals the specified value.

Syntax:

1. { <field> : { $eq: <value> } }

Example:

db.books.find ({ price: { $eq: 200 } })

The above example queries the books collection to select all documents where the value of the price filed equals 300.

2. $gt

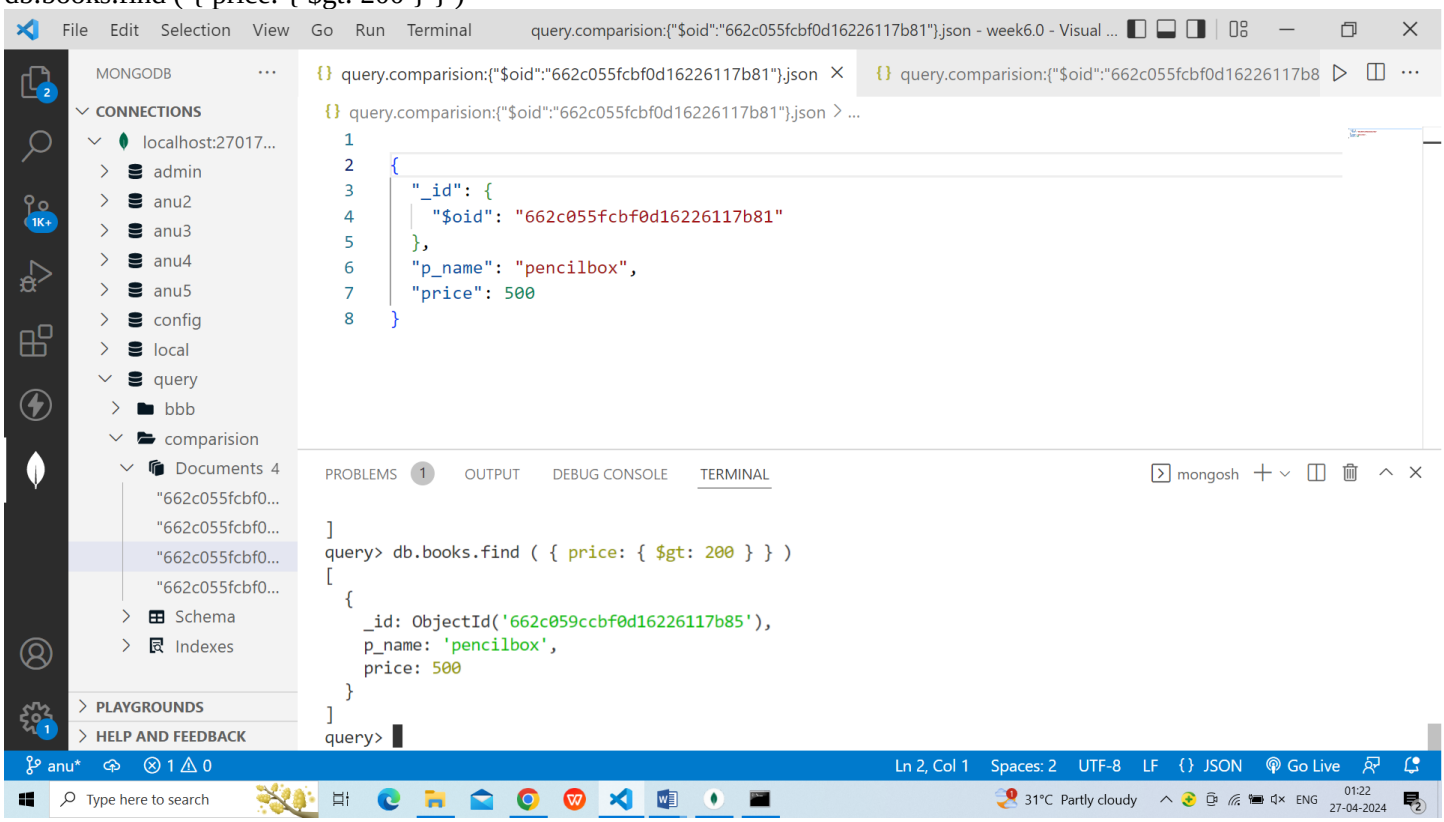
The $gt chooses a document where the value of the field is greater than the specified value.

Syntax:

1. { field: { $gt: value } }

Example:

1. db.books.find ({ price: { $gt: 200 } })



3.$gte

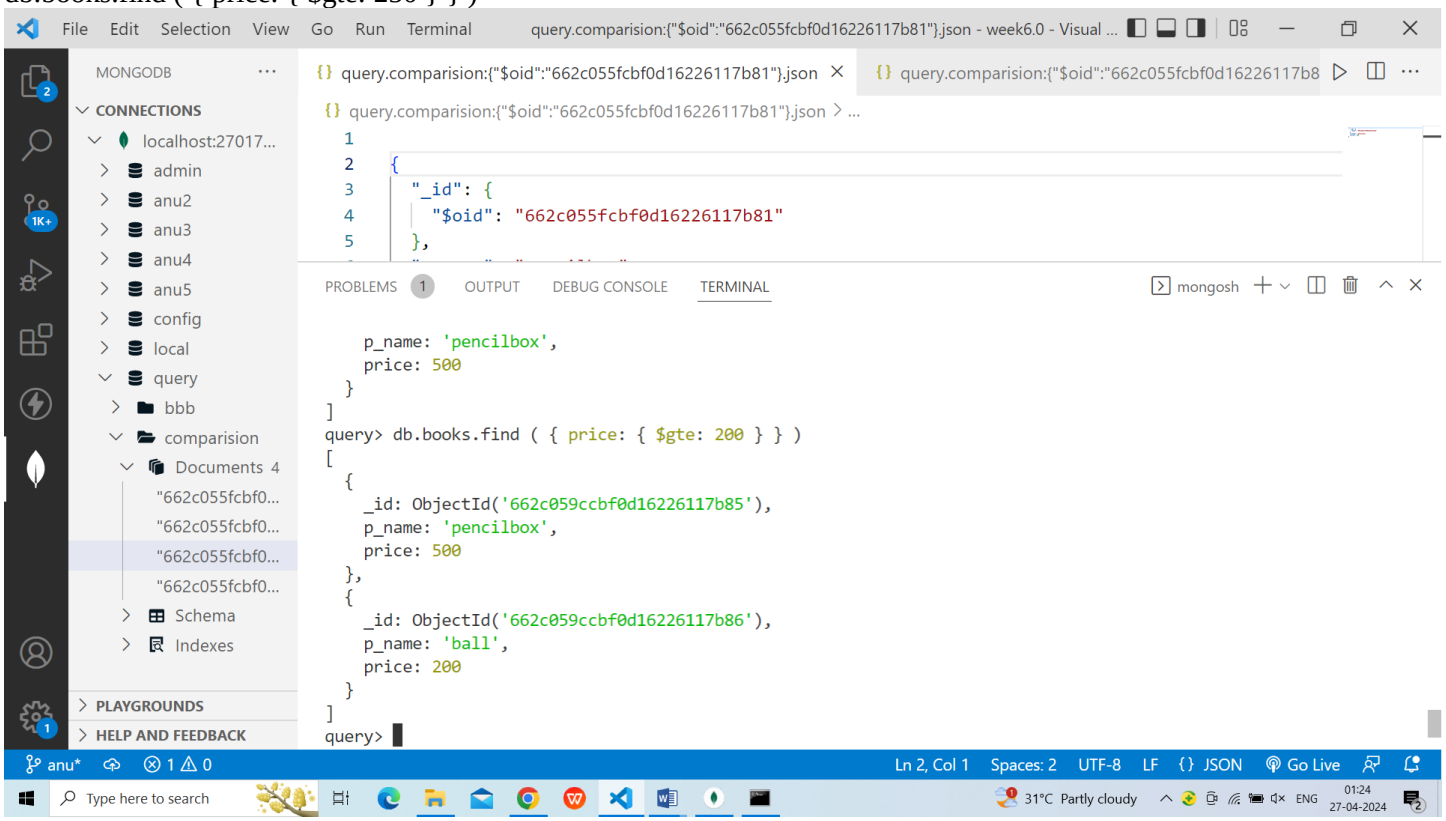
The $gte choose the documents where the field value is greater than or equal to a specified value.

Syntax:

1. { field: { $gte: value } }

Example:

1. db.books.find ({ price: { $gte: 250 } })



4.$in

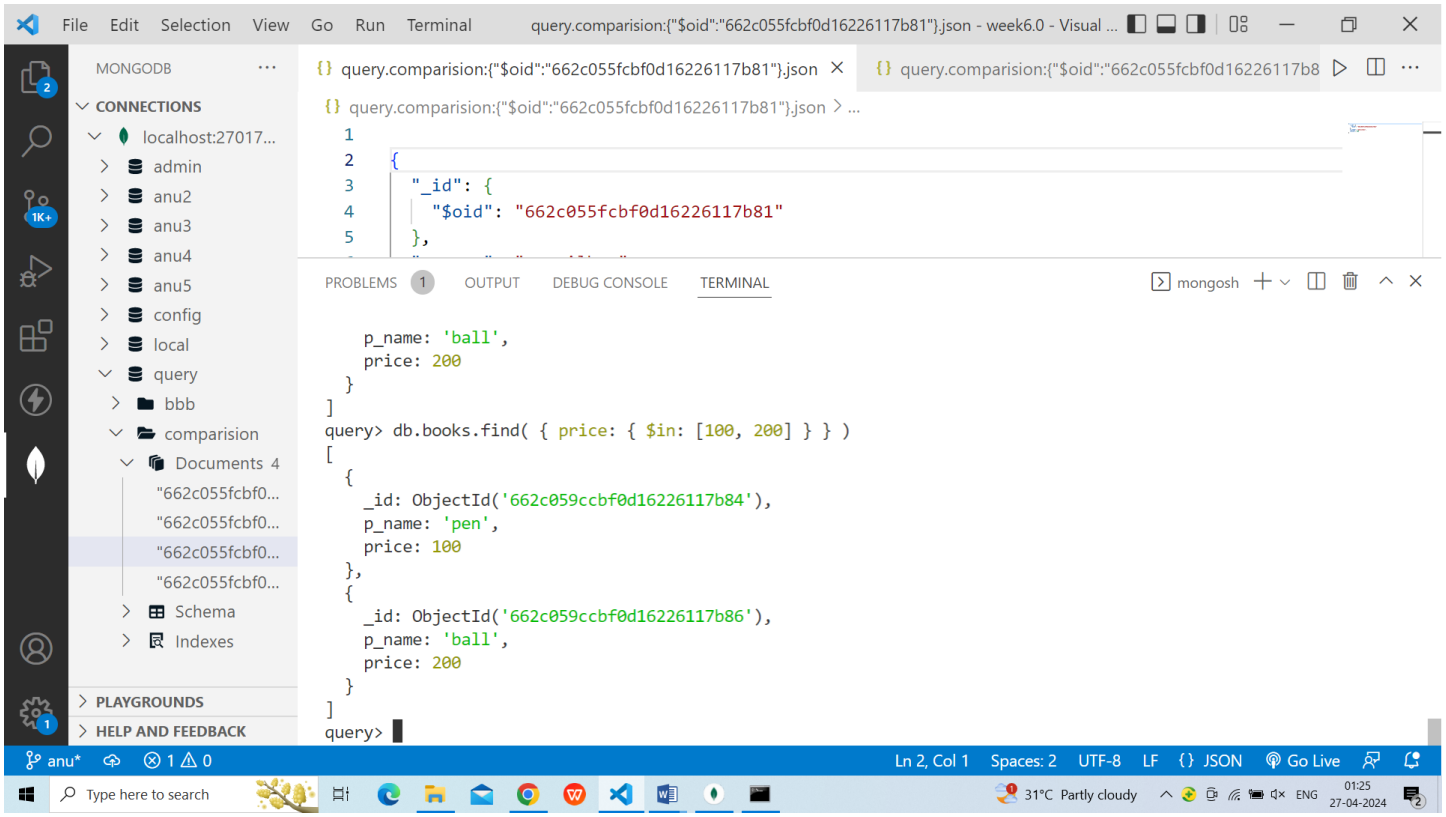
The $in operator choose the documents where the value of a field equals any value in the specified array.

Syntax:

1. { filed: { $in: [<value1>, <value2>, ……] } }

Example:

1. db.books.find({ price: { $in: [100, 200] } })



5.$lt

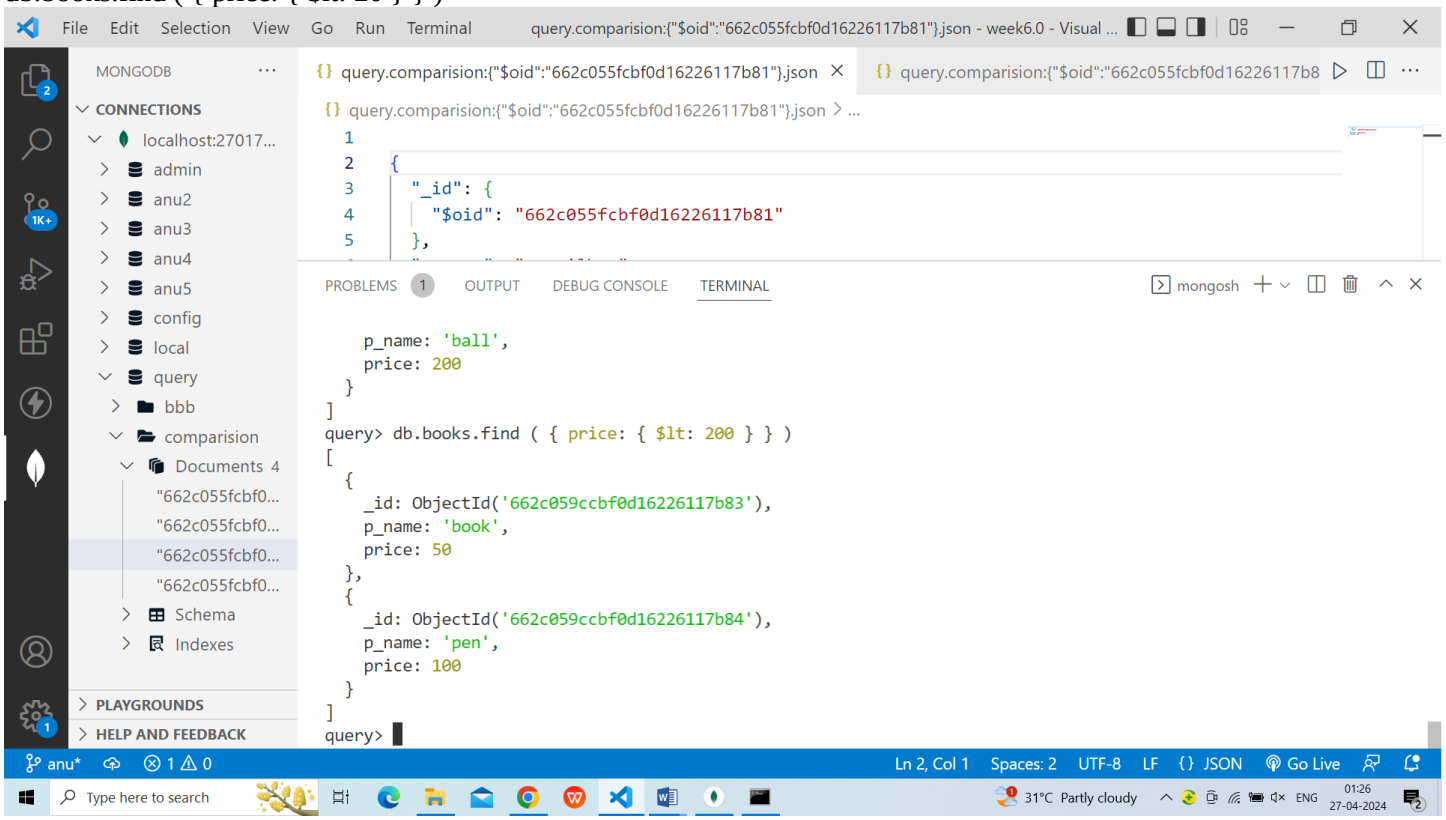
The $lt operator chooses the documents where the value of the field is less than the specified value.

Syntax:

1. { field: { $lt: value } }

Example:

1. db.books.find ({ price: { $lt: 20 } })



6. $lte

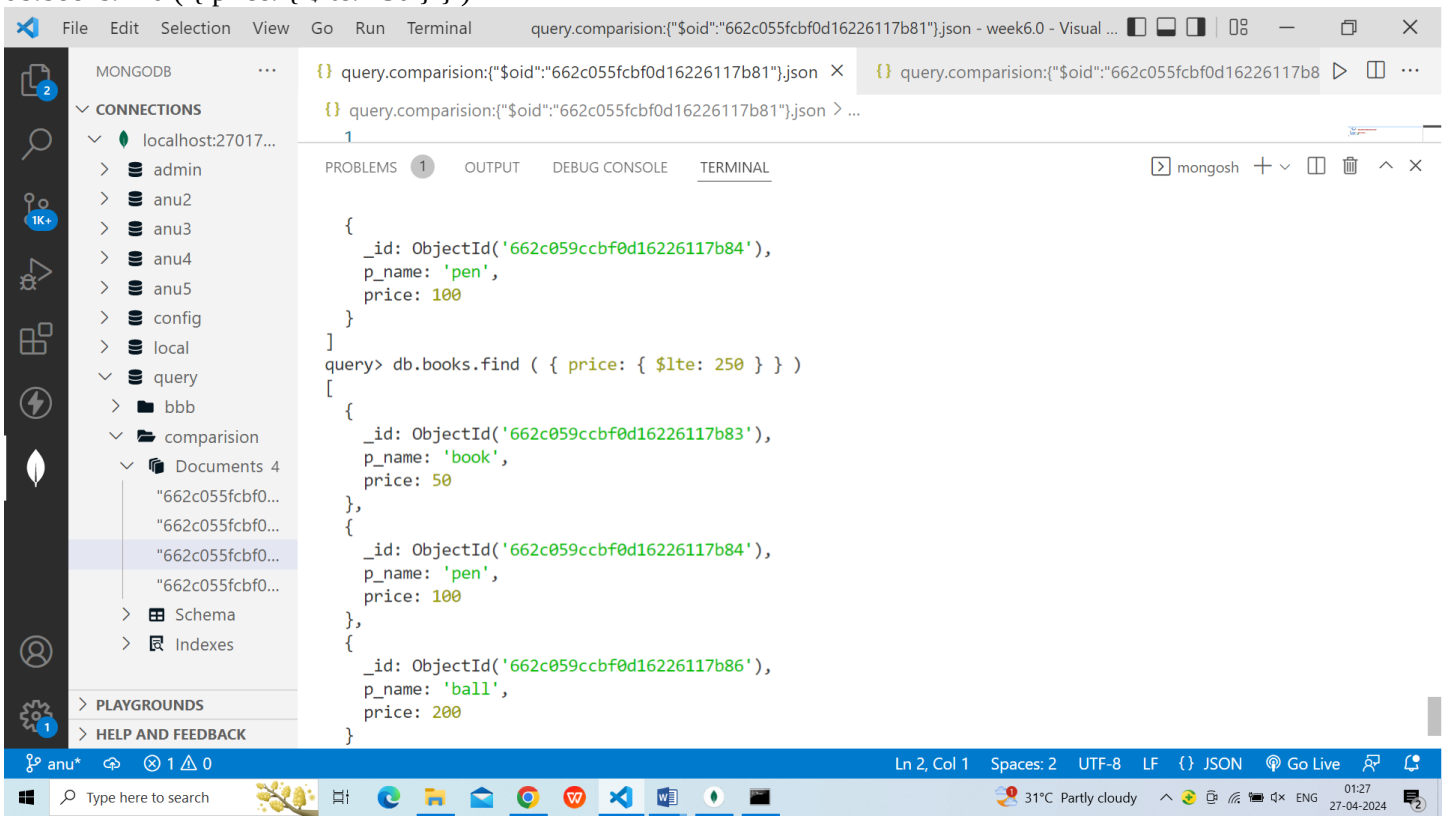
The $lte operator chooses the documents where the field value is less than or equal to a specified value.

Syntax:

1. { field: { $lte: value } }

Example:

1. db.books.find ({ price: { $lte: 250 } })



7. $ne

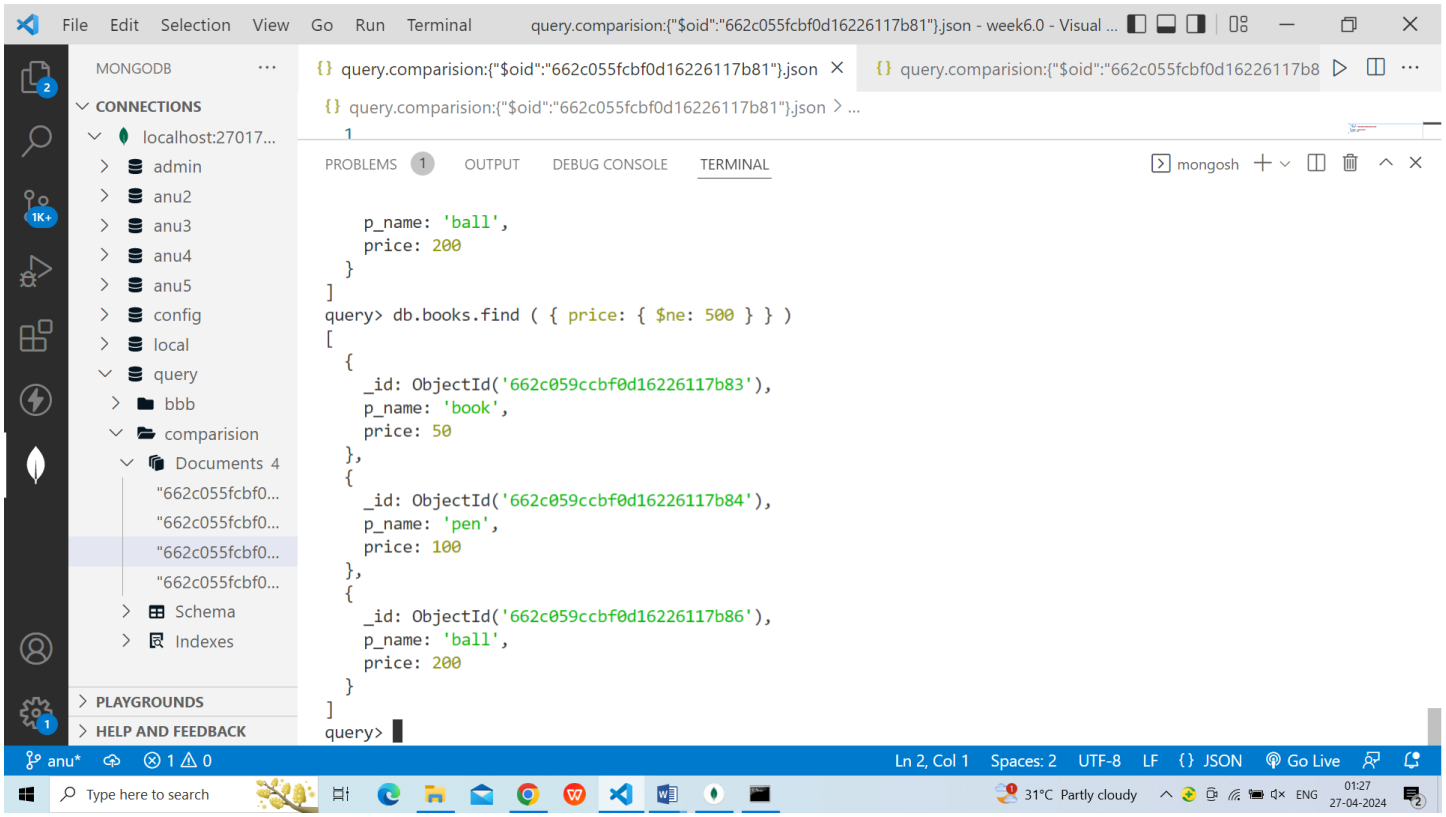
The $ne operator chooses the documents where the field value is not equal to the specified value.

Syntax:

1. { <field>: { $ne: <value> } }

Example:

1. db.books.find ({ price: { $ne: 500 } })



8. $nin

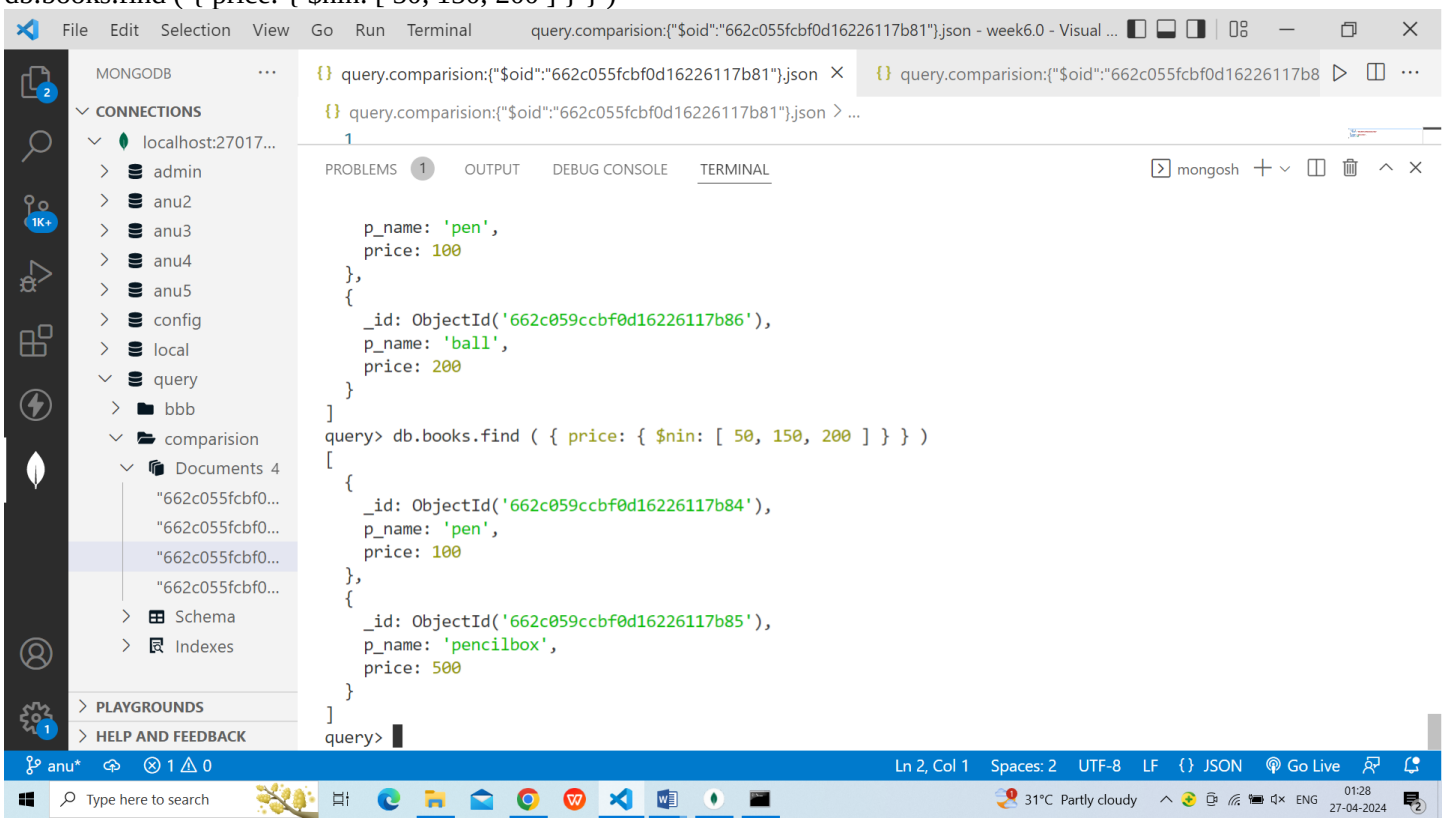
The $nin operator chooses the documents where the field value is not in the specified array or does not exist.

Syntax:

1. { field : { $nin: [<value1>, <value2>, ...] } }

Example:

1. db.books.find ({ price: { $nin: [50, 150, 200] } })



2.MongoDB Logical Operator

Logical Operators

The logical operators in MongoDB are used to filter data based on expressions that evaluate to true or false.
The Logical operators in MongoDB are shown in the table below:

Logical Operator	Description	Syntax
$and	Returns all the documents that satisfy all the conditions.	{ $and: [{ <expression1> }, { <expression2> } , ... , { <expressionN> }] }
$not	Inverts the effect of the query expression and returns documents that do not match the query expression.	{ field: { $not: { <operator-expression> } } }
$or	Returns the documents from the query that match either one of the conditions in the query.	{ $or: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }
$nor	Returns the documents that fail to match both conditions.	{ $nor: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }

$and

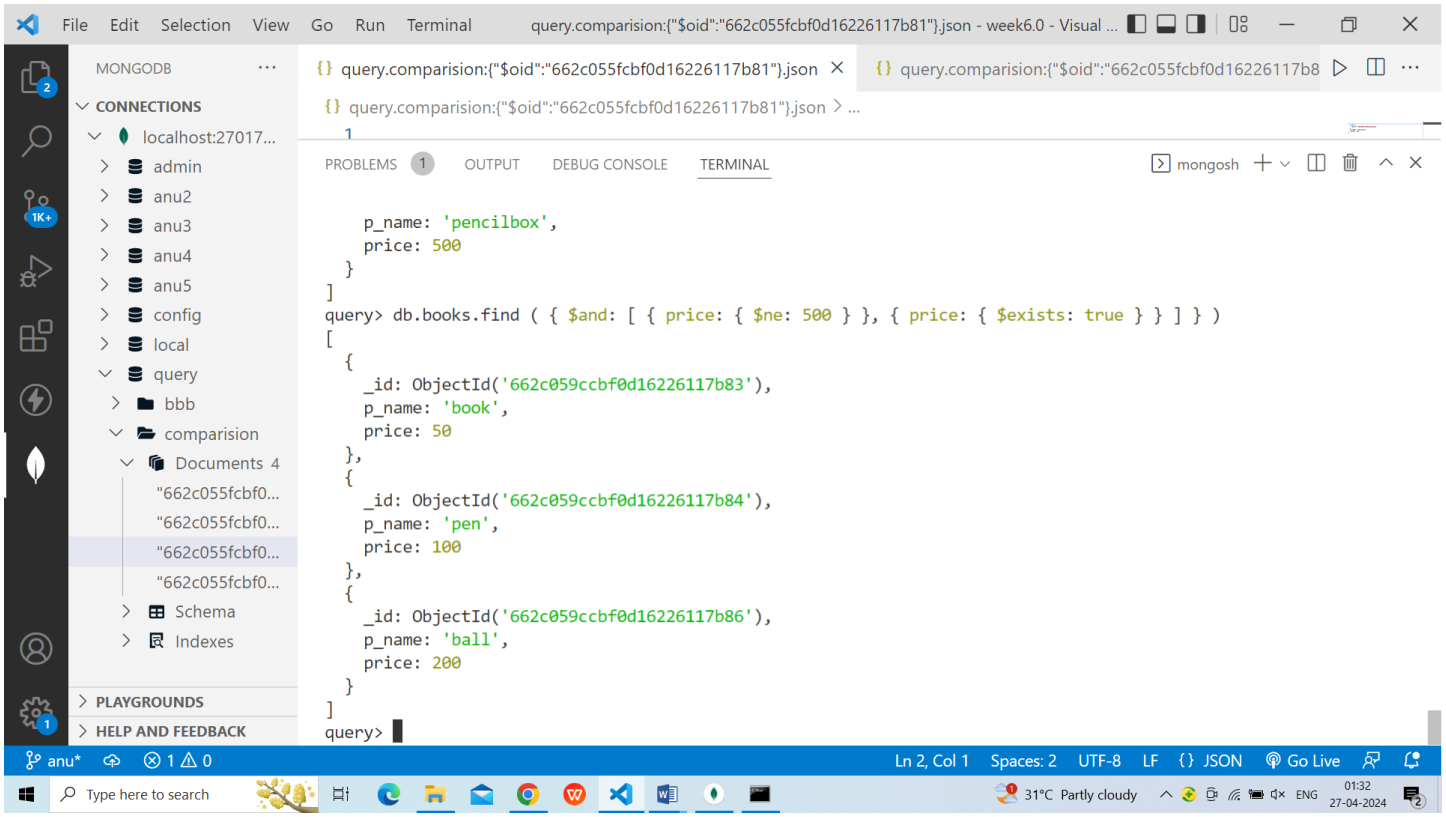
The $and operator works as a logical AND operation on an array. The array should be of one or more expressions and chooses the documents that satisfy all the expressions in the array.

Syntax:

1. { $and: [{ <exp1> }, { <exp2> }, ...]}

Example:

1. db.books.find ({ $and: [{ price: { $ne: 500 } }, { price: { $exists: **true** } }] })



$not

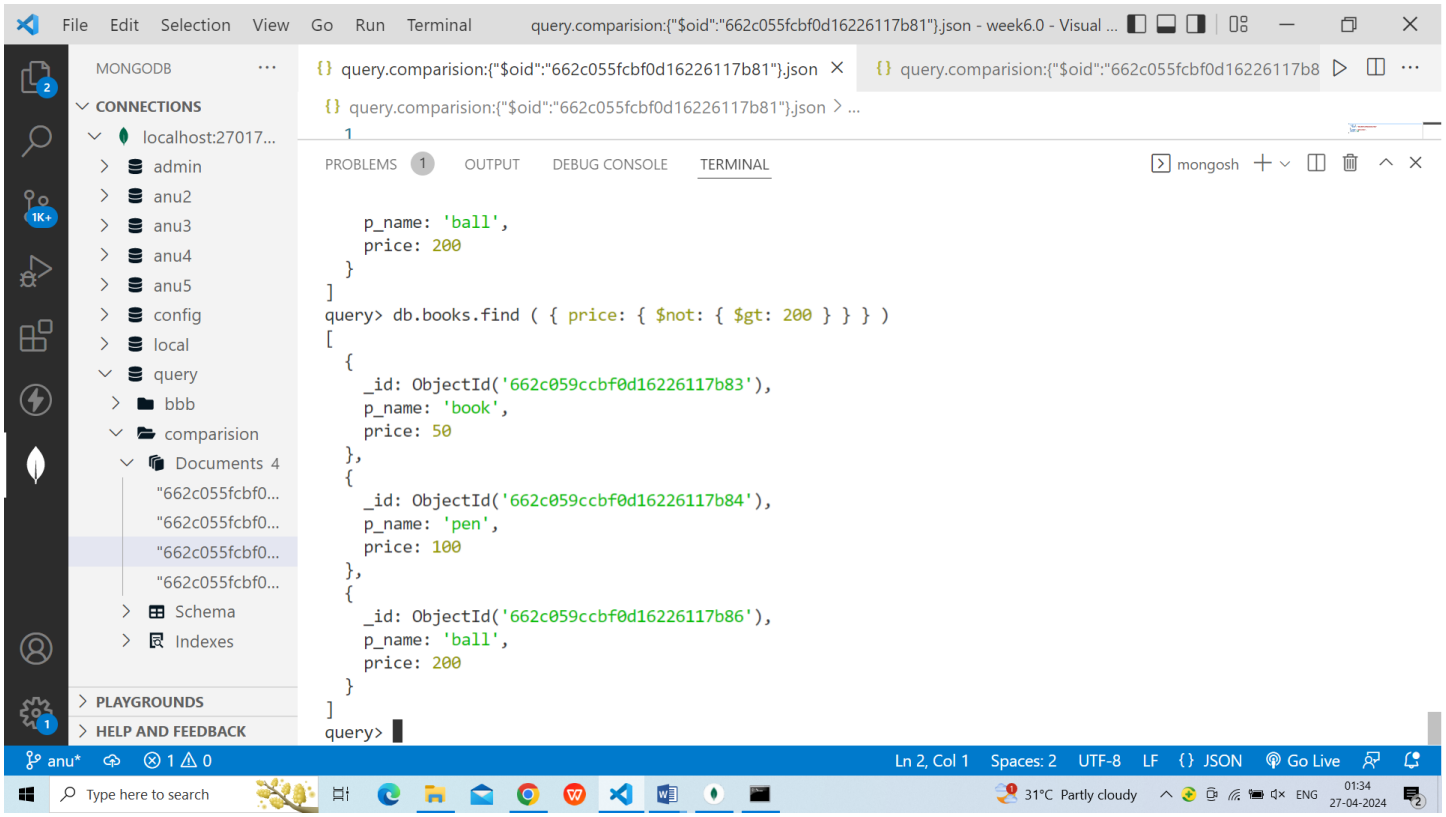
The $not operator works as a logical NOT on the specified expression and chooses the documents that are not related to the expression.

Syntax:

1. { field: { $not: { <operator-expression> } } }

Example:

1. db.books.find ({ price: { $not: { $gt: 200 } } })



$nor

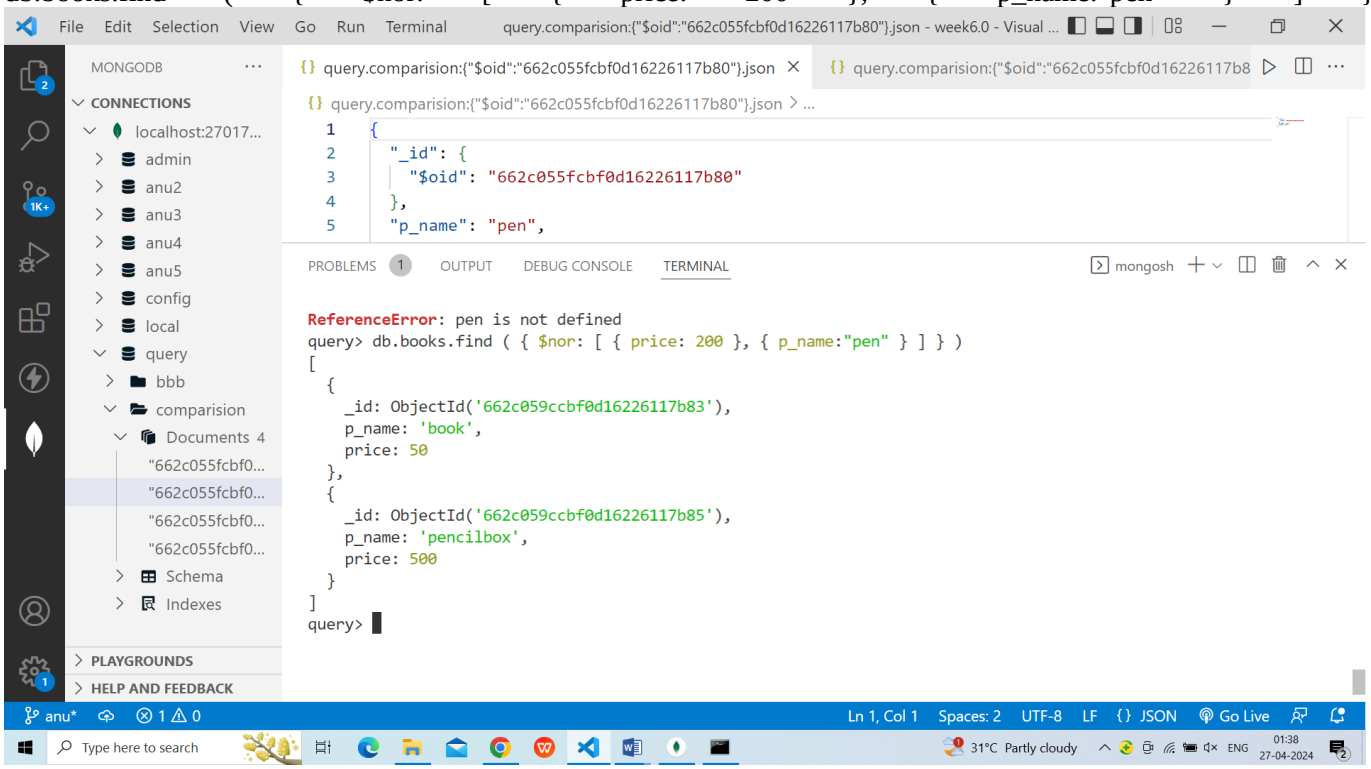
The $nor operator works as logical NOR on an array of one or more query expression and chooses the documents that fail all the query expression in the array.

Syntax:

1. { $nor: [{ <expression1> } , { <expresion2> } , ...] }

Example:

db.books.find ({ $nor: [{ price: 200 }, { p_name:"pen" }] })



$or

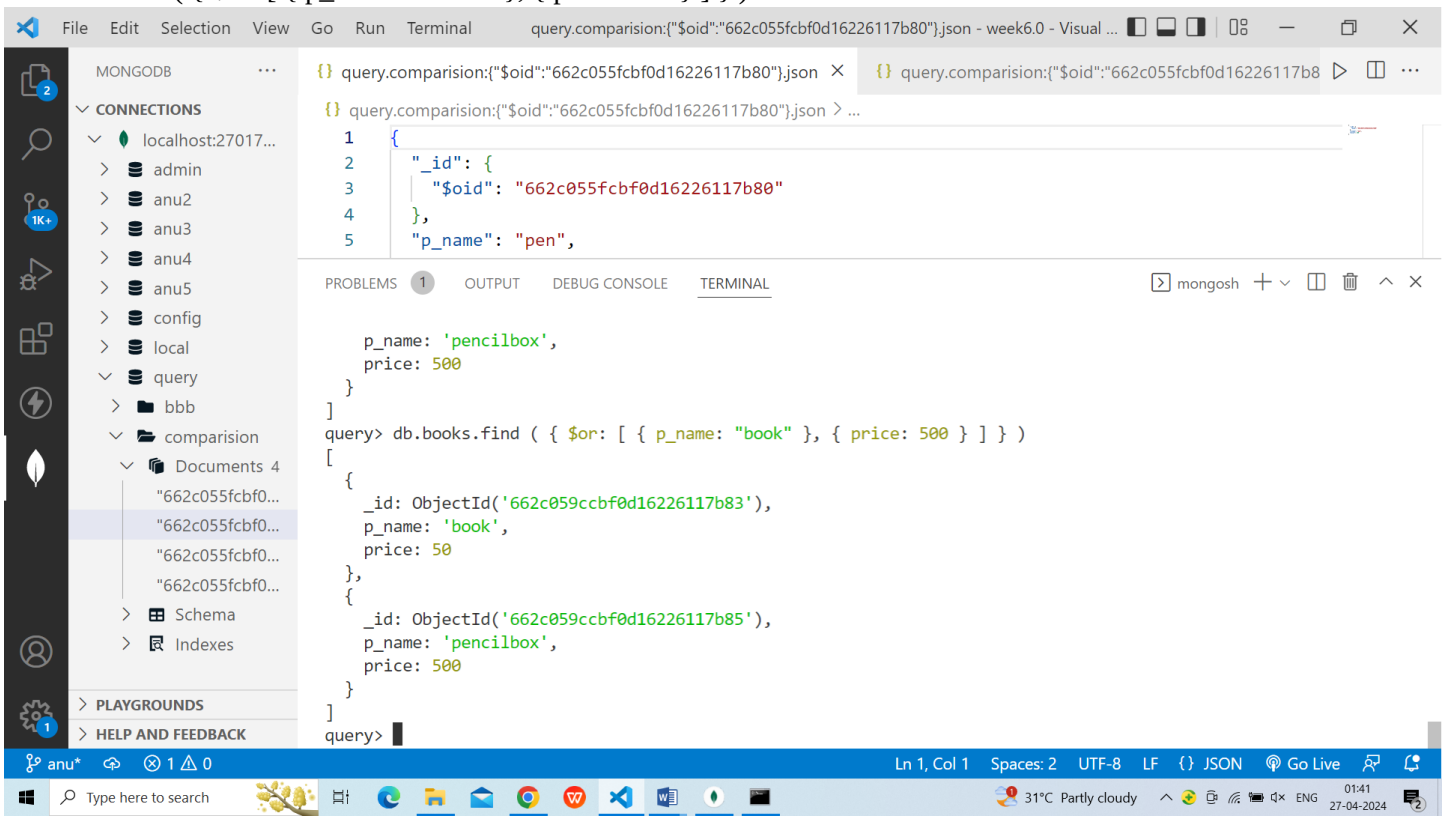
It works as a logical OR operation on an array of two or more expressions and chooses documents that meet the expectation at least one of the expressions.

Syntax:

1. { $or: [{ <exp_1> }, { <exp_2> }, ... , { <exp_n> }] }

Example:

db.books.find ({ $or: [{ p_name: "book" }, { price: 500 }] })



3.Array Operator

Name	Description
$all	Matches arrays that contain all elements specified in the query.
$elemMatch	Selects documents if element in the array field matches all the specified $elemMatch conditio
$size	Selects documents if the array field is a specified size.

$all

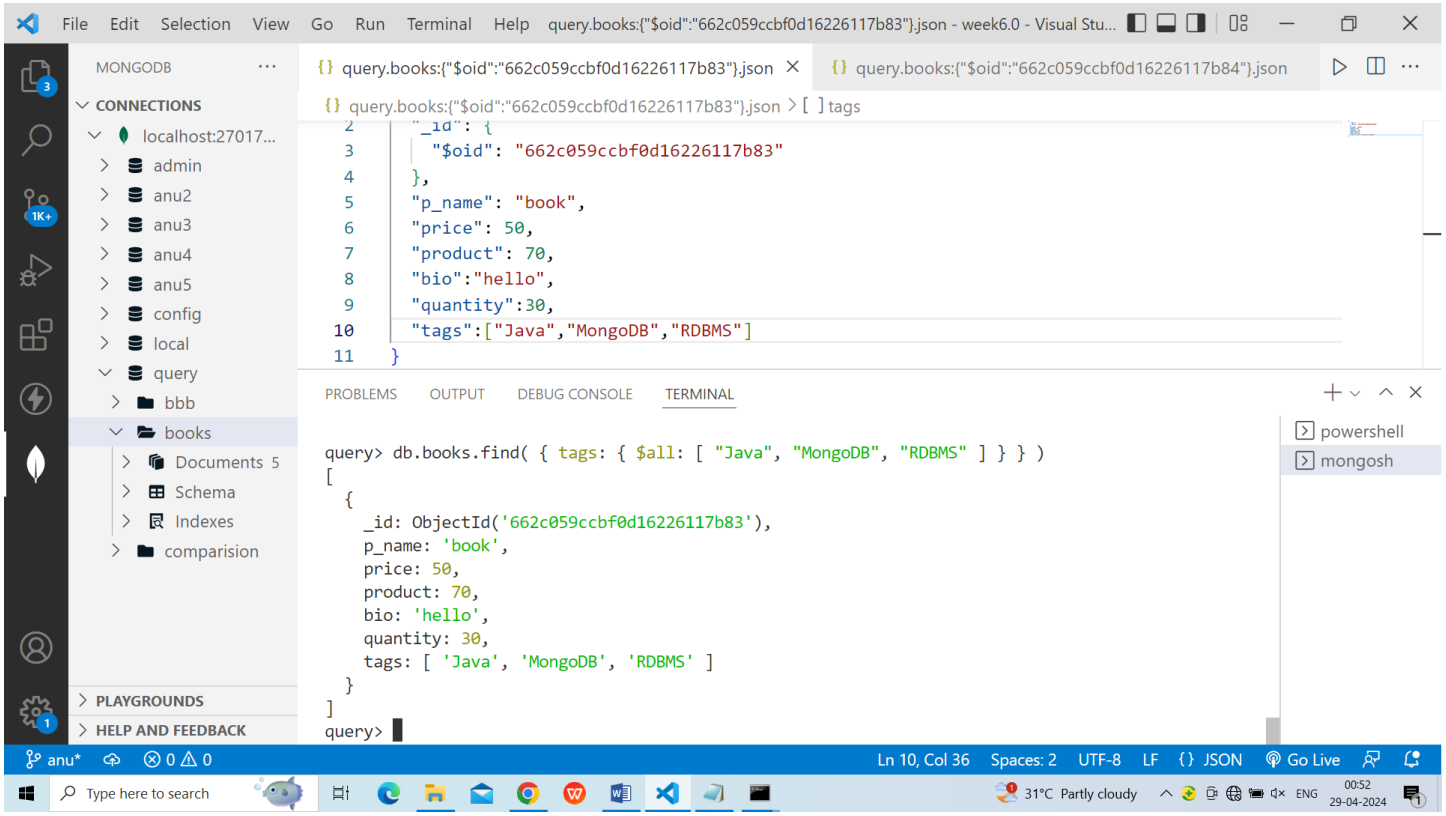
It chooses the document where the value of a field is an array that contains all the specified elements.

Syntax:

1. { <field>: { $all: [<value1> , <value2> ...] } }

Example:

1. db.books.find({ tags: { $all: ["Java", "MongoDB", "RDBMS"] } })



$elemMatch

The operator relates documents that contain an array field with at least one element that matches with all the given query criteria.

Syntax:

1. { <field>: { $elemMatch: { <query1>, <query2>, ... } } }

Example of Using $elemMatch Operator

Let's first make some updates in our demo collection. Here we will Insert some data into the count_no database

Query:

```
db.count_no.insertOne( {"name" : "Harsha", "Age": 24,"Likes" : 2,"Colors":["Red","Green","Blue"] } );
```

Output:

```
]
Data_base> db.count_no.insertOne({"name":"Harsha","Age":24,"Likes":2,"Colors":["Red","Green","Blue"]});
{
  acknowledged: true,
  insertedId: ObjectId("6589856d543f99be41e7a5ab")
}
Data_base> db.count_no.find({},{_id:0});
[
  { name: 'Krishna', Age: 22, Likes: 1 },
  { name: 'Shiva', Age: 25, Likes: 2 },
  { name: 'Rama', Age: 23, Likes: 2 },
  { name: 'Hanuman', Age: 23, Likes: 3 },
  {
    name: 'Harsha',
    Age: 24,
    Likes: 2,
    Colors: [ 'Red', 'Green', 'Blue' ]
  }
]
```

Inserting the array of elements .

Now, using the $**elemMatch** operator in MongoDB let's match the Red colors from a set of Colors.

Query:

```
db.count_no.find( {"Colors": {$elemMatch: {$eq: "Red" } } },{_id: 0 } );
```

Output:

```
Data_base> db.count_no.find({"Colors":{$elemMatch:{$eq:"Red"}}},{_id:0});
[
  {
    name: 'Harsha',
    Age: 24,
    Likes: 2,
    Colors: [ 'Red', 'Green', 'Blue' ]
  }
]
Data_base>
```

Using the elemMatch Operator.

Explanation: The $elemMatch operator is used with the field **Colors** which is of type array. In the above query, it returns the documents that have the field Colors and if any of the values in the Colors field has "**Red**" in it.

Example:

1. db.books.find({ price: { $elemMatch: { $gte: 500, $lt: 400 } } })

$size

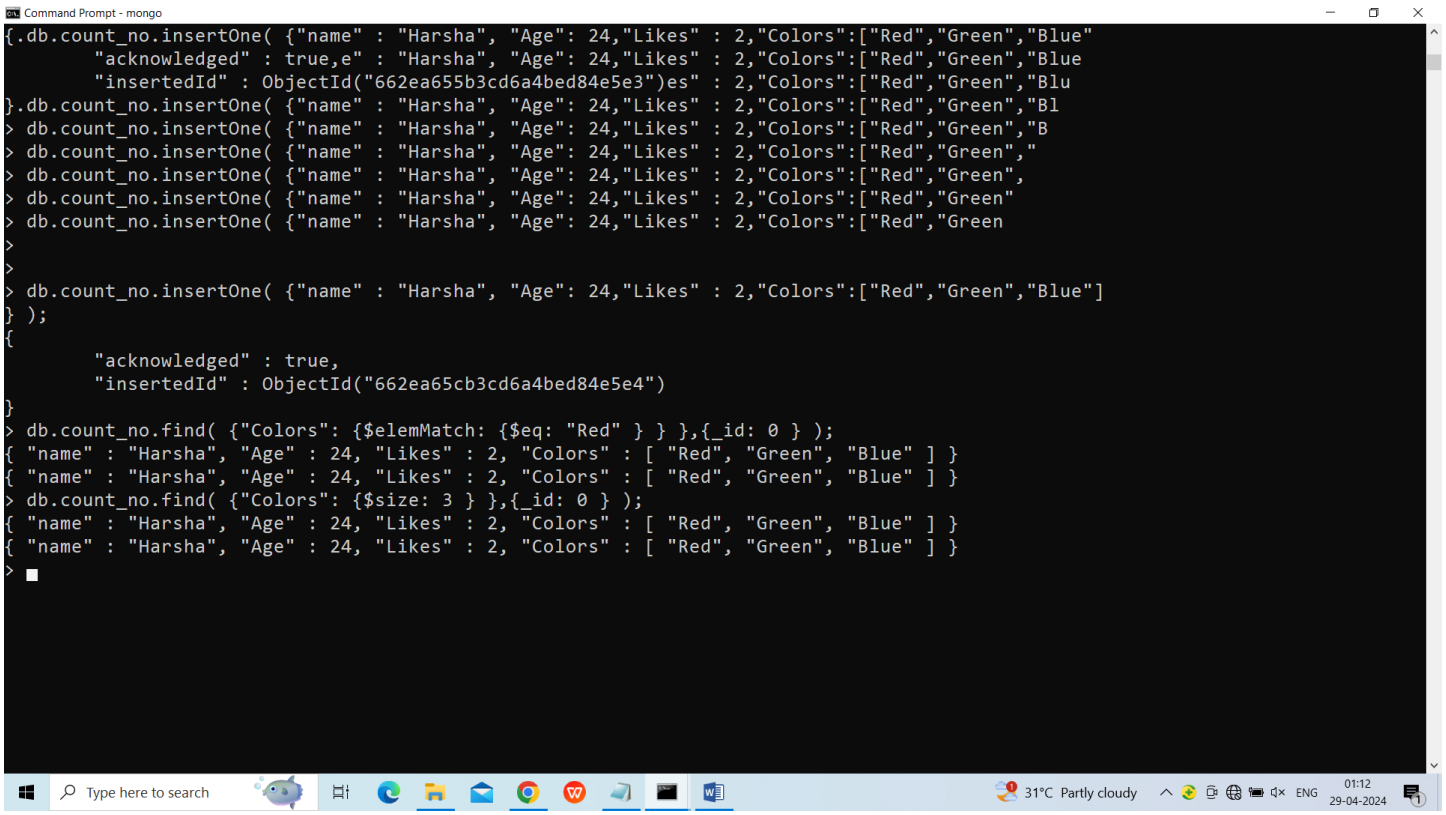
It selects any array with the number of the element specified by the argument.

Syntax:

1. db.collection.find({ field: { $**size**: 2 } });

2. db.count_no.find({"Colors": {$size: 3 } },{_id: 0 });

```
Data_base> db.count_no.find({"Colors":{$elemMatch:{$eq:"Red"}}},{_id:0});
[
  {
    name: 'Harsha',
    Age: 24,
    Likes: 2,
    Colors: [ 'Red', 'Green', 'Blue' ]
  }
]
Data_base>
```



4.MongoDB Evaluation Operator

The evaluation operators in the MongoDB are used to return the documents based on the result of the given expression.
Some of the evaluation operators present in the MongoDB are:

Evaluation Operator	Description	Syntax
$mod operator	The $mod operator in MongoDB performs a modulo operation on the value of a field and selects documents where the modulo equals a specified value. It only works with numerical fields.	{ field: { $mod: [divisor, remainder] } }
$expr operator	The $expr operator in MongoDB allows aggregation expressions to be used as query conditions. It returns documents that satisfy the conditions of the query.	{ $expr: { <aggregation expression> } }
$where operator	The $where operator in MongoDB uses JavaScript expression or function to perform queries. It evaluates the function for every document in the database and returns the documents that match the condition.	{ $where: <JavaScript expression

Evaluation

Name	Description
$expr	Allows use of aggregation expressions within the query language.
$jsonSchema	Validate documents against the given JSON Schema.
$mod	Performs a modulo operation on the value of a field and selects documents with a specified result.
$regex	Selects documents where values match a specified regular expression.
$text	Performs text search.
$where	Matches documents that satisfy a JavaScript expression.

$expr

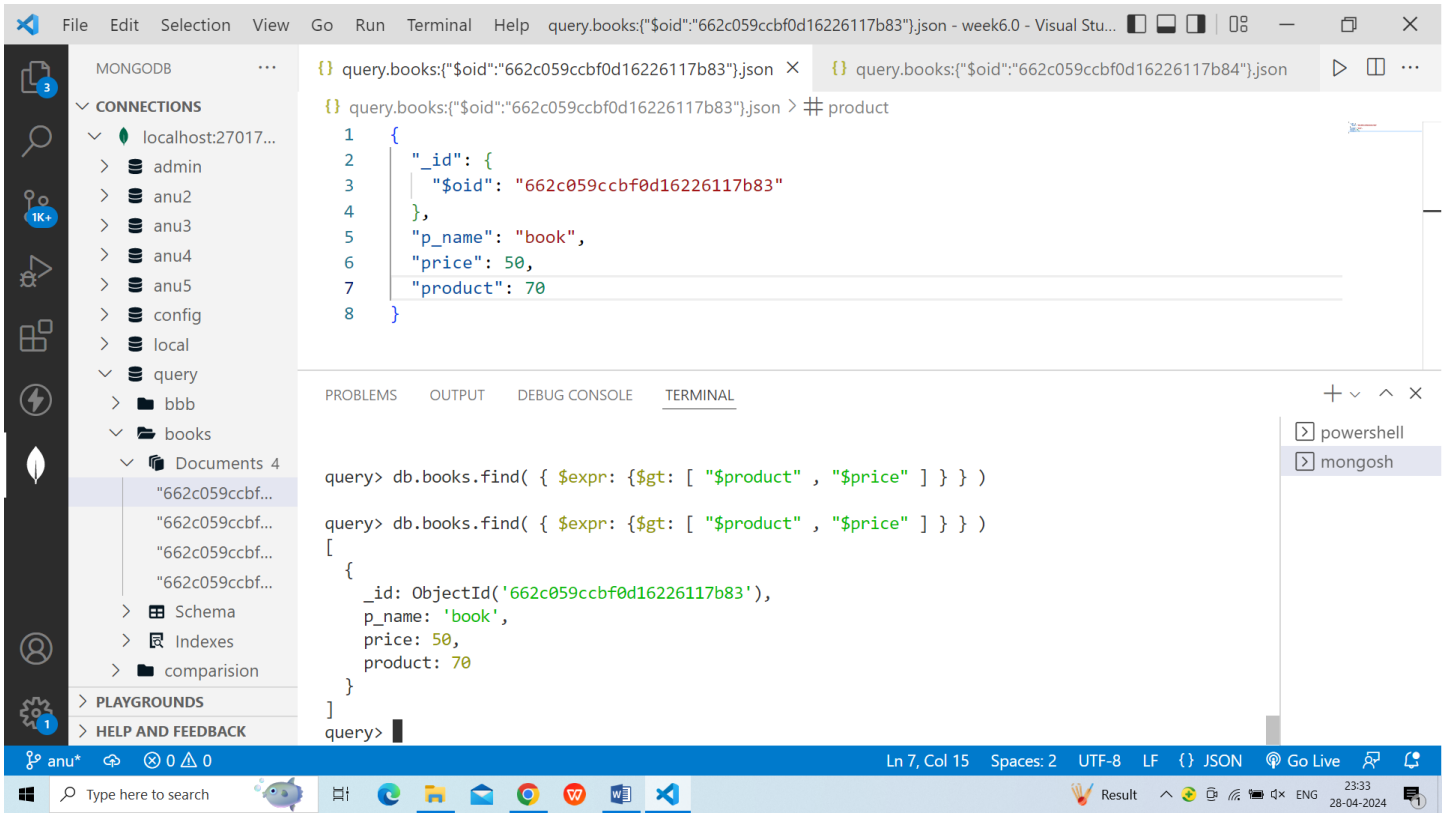
The expr operator allows the use of aggregation expressions within the query language.

Syntax:

1. { $expr: { <expression> } }

Example:

1. db.store.find({ $expr: {$gt: ["$product" , "$price"] } })



$jsonSchema

It matches the documents that satisfy the specified JSON Schema.

db.createCollection("students12",{

validator:{

$jsonSchema: {

required: ["name", "major", "gpa", "address"],

properties: {

name: {

bsonType: "string",

description: "must be a string and is required"

},

address: {

bsonType: "object",

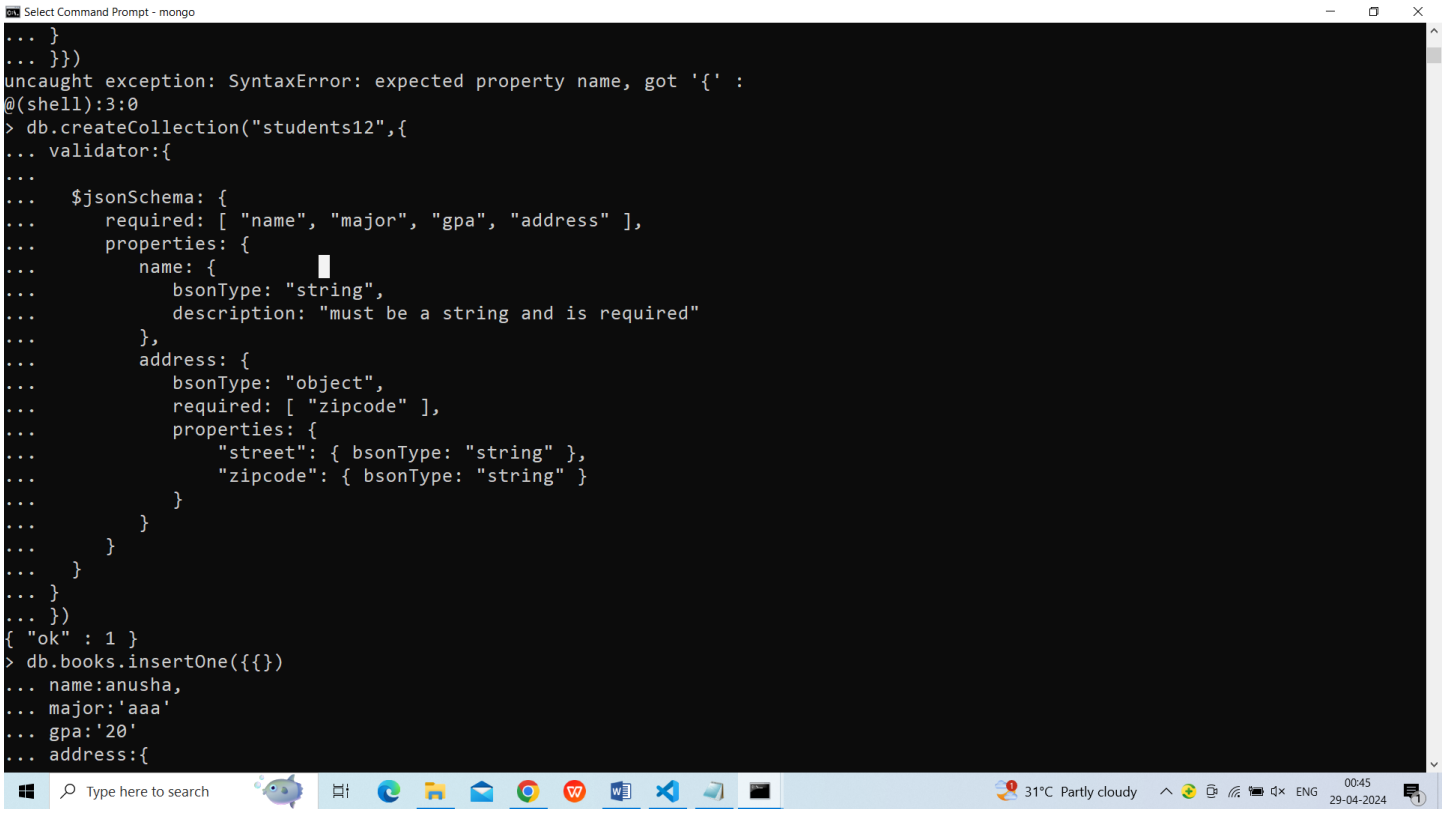
required: ["zipcode"],

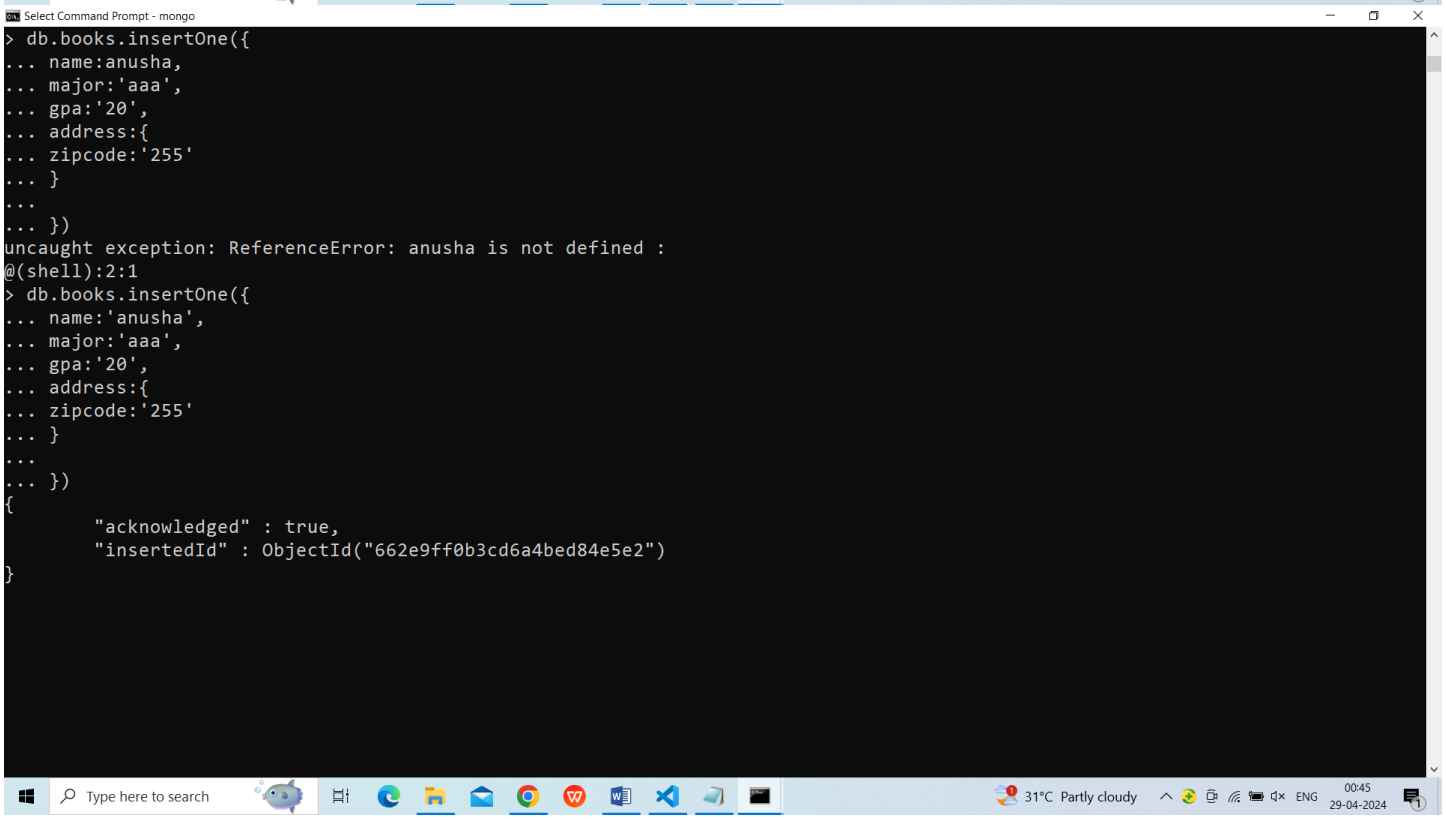
properties: {

```
            "street": { bsonType: "string" },
            "zipcode": { bsonType: "string" }
          }
        }
      }
    }
})
//////
Student:{
name:'anusha',
major:'aaa',
gpa:'20',
address:{
zipcode:'255'
}
```

}**Syntax:**

1. { $jsonSchema: <JSON **schema** object> }





$mod

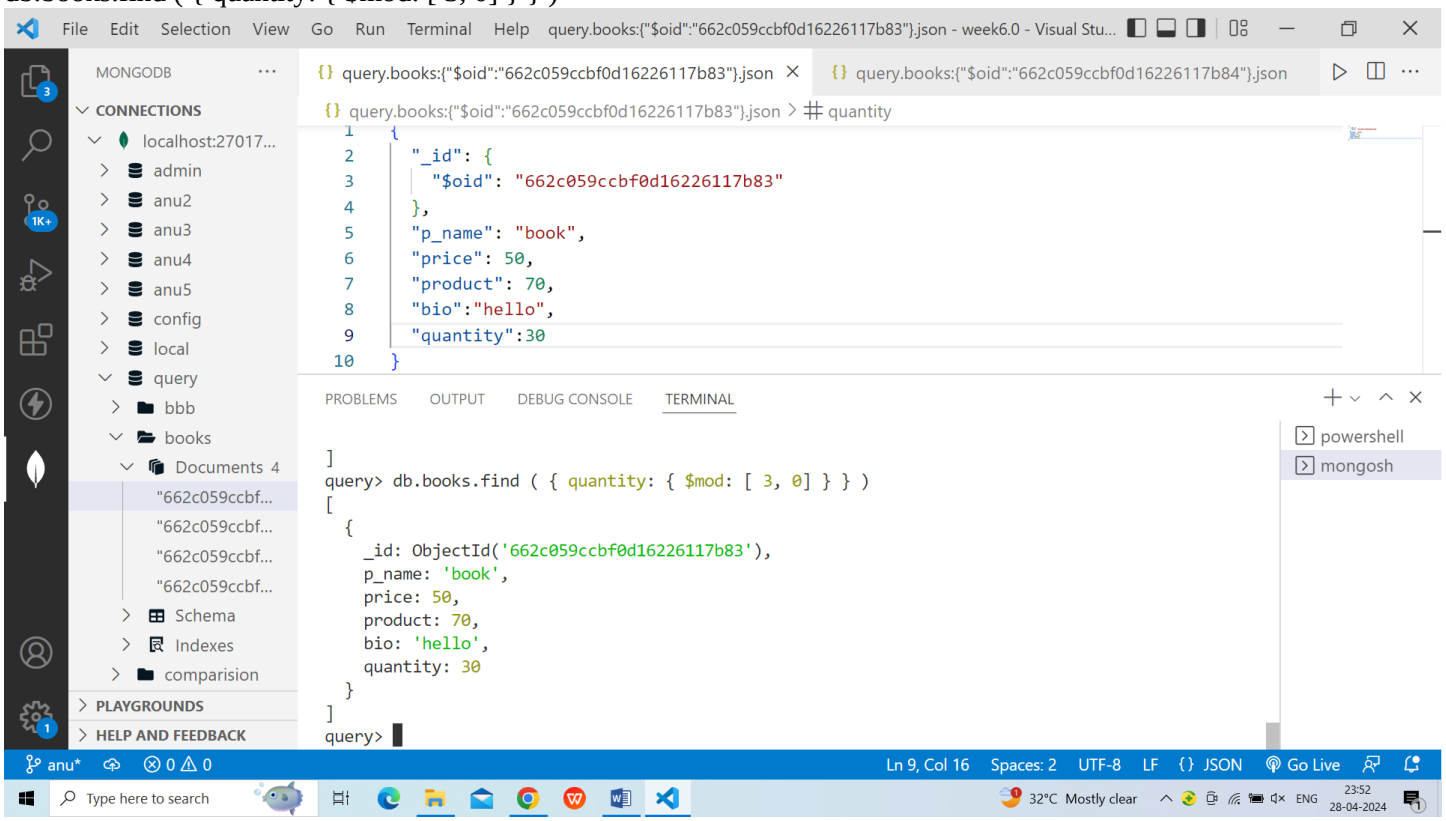
The mod operator selects the document where the value of a field is divided by a divisor has the specified remainder.

Syntax:

1. { field: { $mod: [divisor, remainder] } }

Example:

1. db.books.find ({ quantity: { $mod: [3, 0] } })



$regex

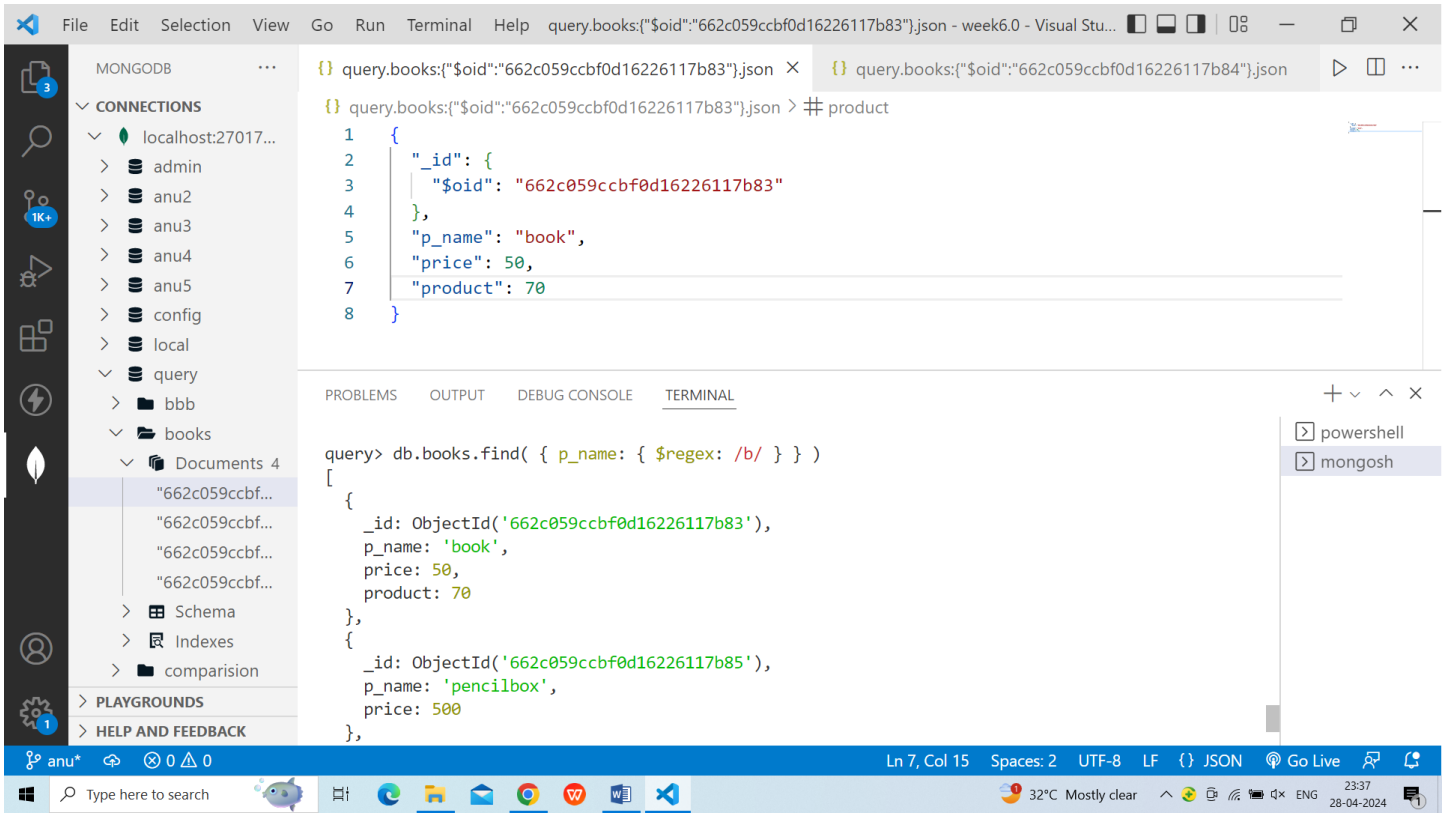
It provides regular expression abilities for pattern matching strings in queries. The MongoDB uses regular expressions that are compatible with Perl.

Syntax:

1. { <field>: /pattern/<options> }

Example:

db.books.find({ p_name: { $regex: /b/ } })



$text

The $text operator searches a text on the content of the field, indexed with a text index.

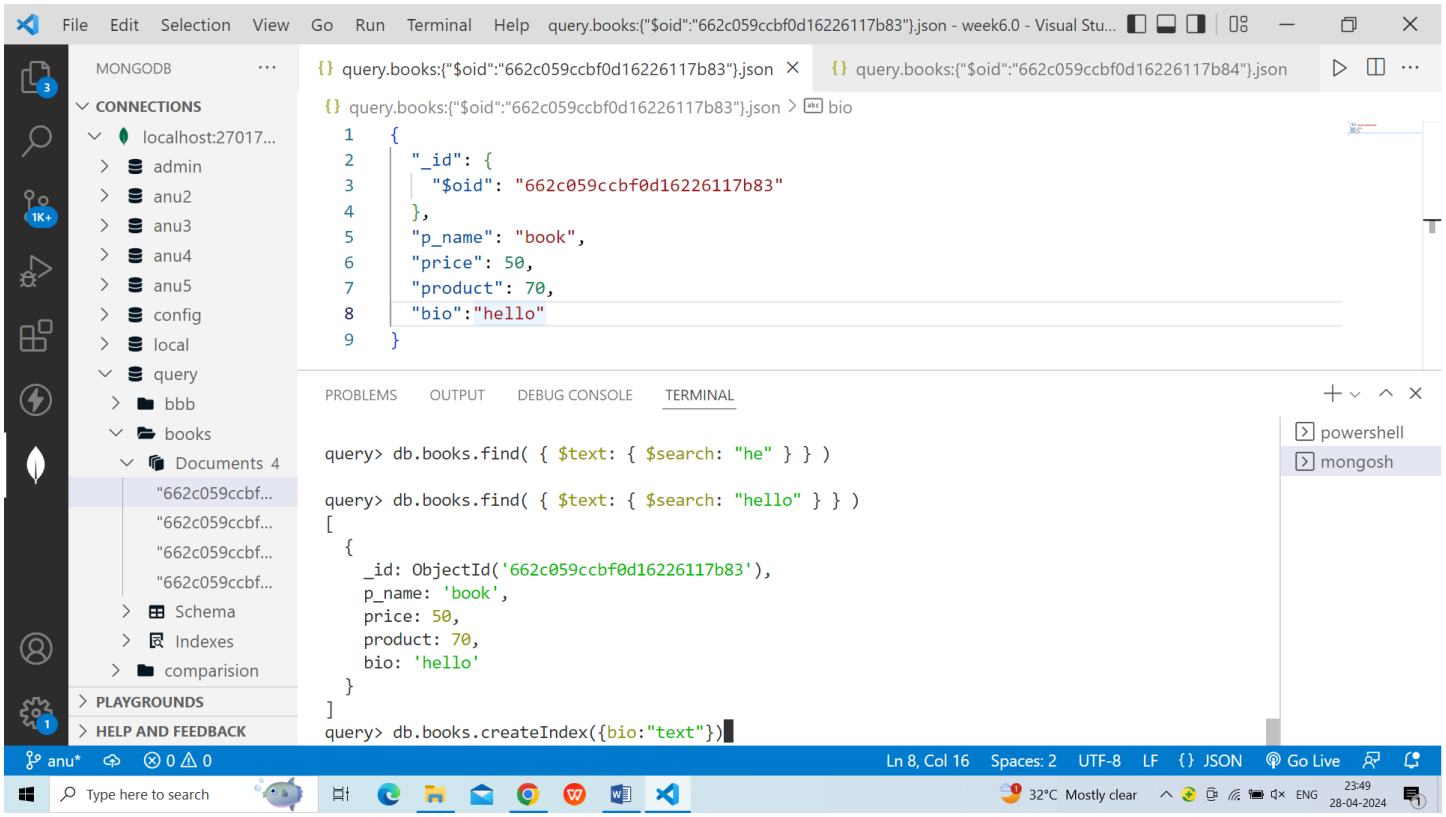
```
db.books.createIndex({bio:"text"})
```

Syntax:

```
1.          {
2.  $text:
3.    {
4.      $search: <string>,
5.      $language: <string>,
6.      $caseSensitive: <boolean>,
7.      $diacriticSensitive: <boolean>
8.    }
9. }
```

Example:

```
1. db.books.find( { $text: { $search: "hello" } } )
```

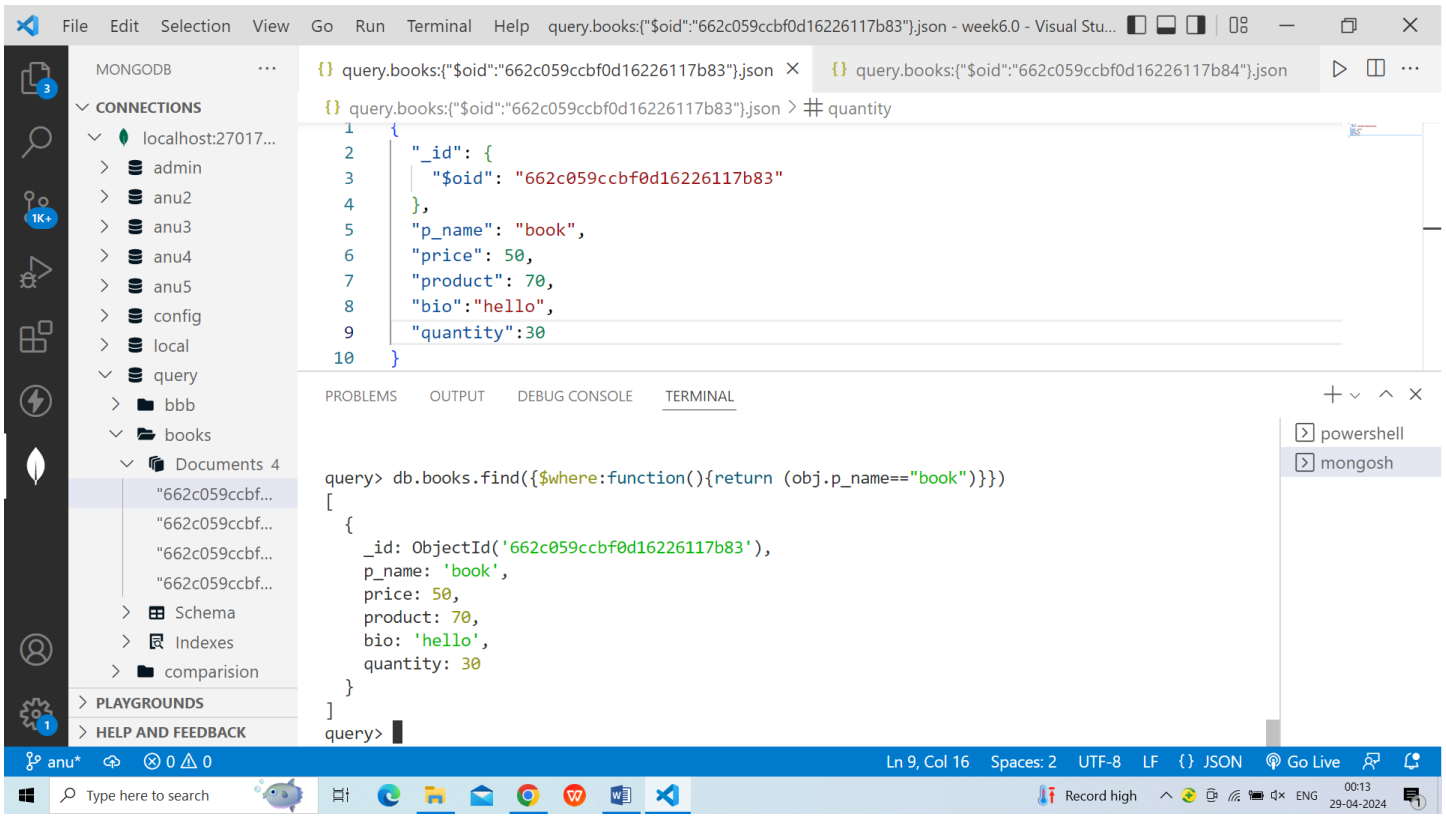


$where

The "where" operator is used for passing either a string containing a JavaScript expression or a full JavaScript function to the query system.

Example:

db.books.find({$where:function(){return (obj.p_name=="book")}})



5. **Element Operators**

The element operators in the MongoDB return the documents in the collection which returns true if the keys match the fields and datatypes.
There are mainly two Element operators in MongoDB:

Element Operator	Description	Syntax
$exists	Checks if a specified field exists in the documents.	{ field: { $exists: <boolean> } }
$type	Verifies the data type of a specified field in the documents.	{ field: { $type: <BSON type> } }

MongoDB Element Operator

$exists

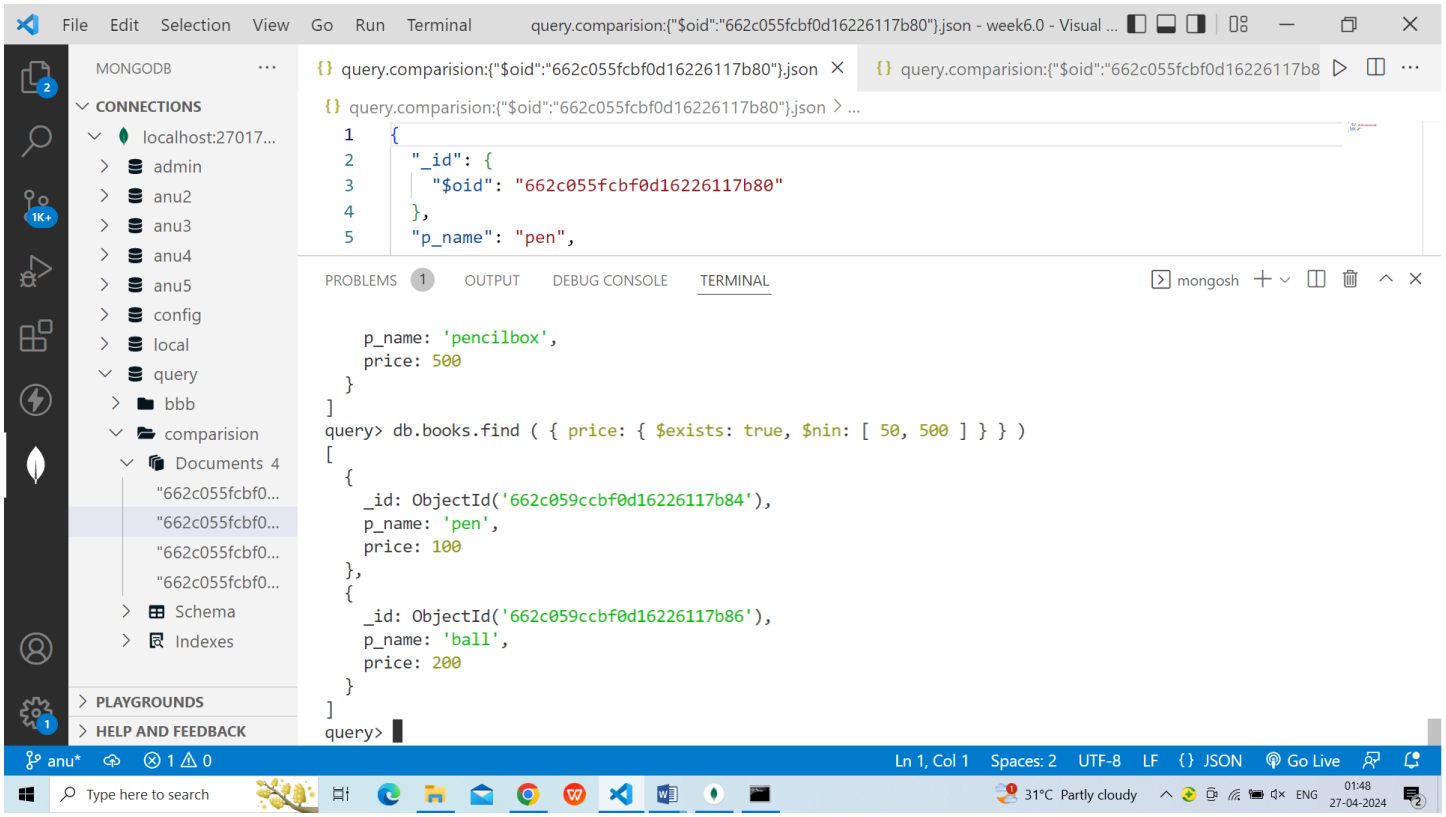
The exists operator matches the documents that contain the field when Boolean is true. It also matches the document where the field value is null.

Syntax:

1. { field: { $exists: <boolean> } }

Example:

db.books.find ({ price: { $exists: true, $nin: [50, 500] } })



$type

The type operator chooses documents where the value of the field is an instance of the specified BSON type.

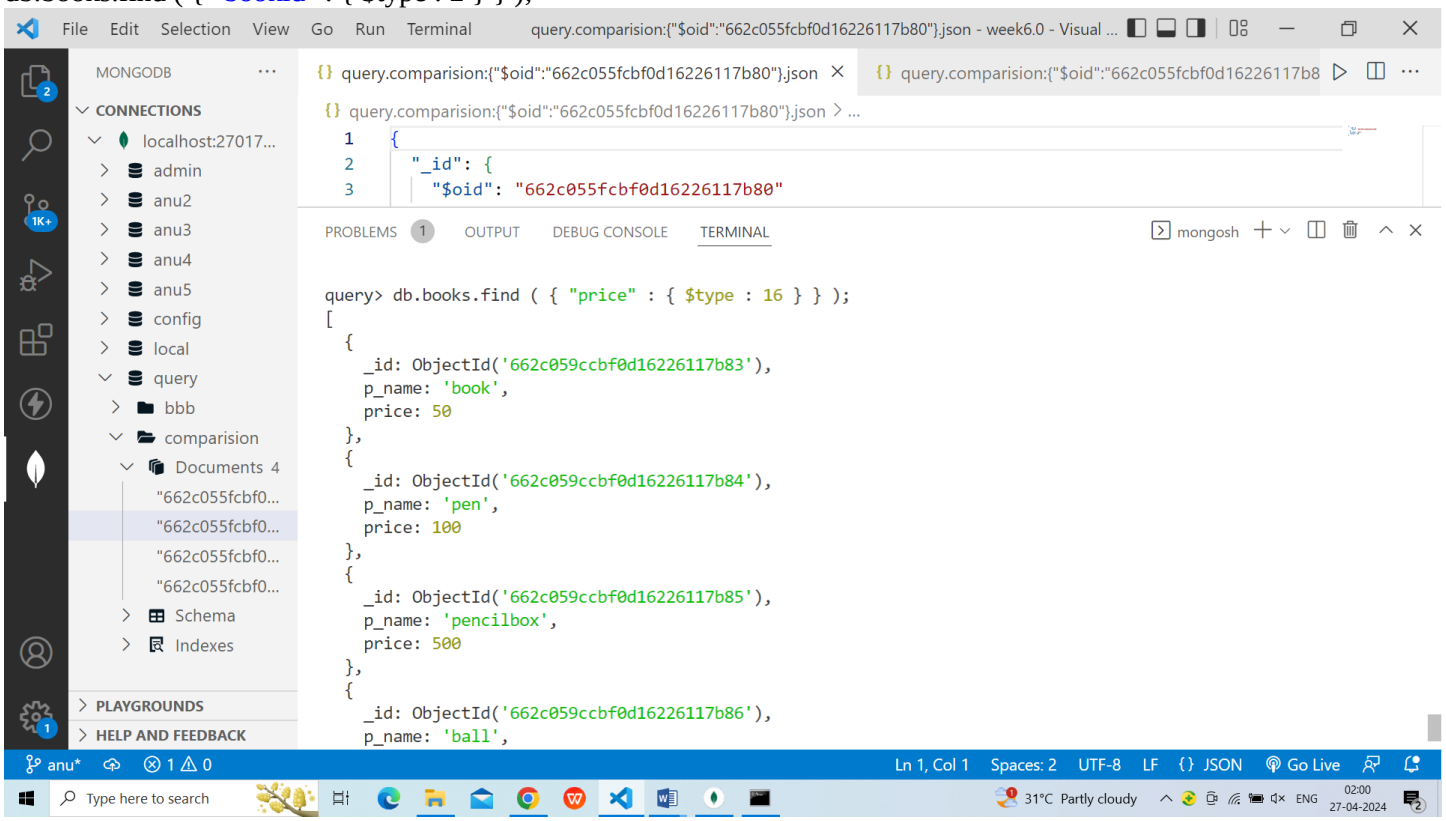
Type	Number	Alias	Notes
Double	1	“double”	
String	2	“string”	
Object	3	“object”	
Array	4	“array”	
Binary Data	5	“binData”	
Undefined	6	“undefined”	Deprecated
ObjectId	7	“objectId”	
Boolean	8	“bool”	
Date	9	“date”	
Null	10	“null”	
Regular Expression	11	“regex”	
DBPointer	12	“dbPointer”	Deprecated
JavaScript	13	“javascript”	
Symbol	14	“symbol”	Deprecated
Javascript (with scope)	15	“javascriptWithScope”	
32-bit integer	16	“int”	
Timestamp	17	“timestamp”	
64-bit Integer	18	“long”	
Decimal128	19	“decimal	New in Version 3.4
Min Key	-1	“minKey”	
Max Key	127	“maxKey”	

Syntax:

1. { field: { $type: <BSON type> } }

Example:

1. db.books.find ({ "bookid" : { $type : 2 } });



6. **Bitwise Operators**

The Bitwise operators in the MongoDB return the documents in the MongoDB mainly on the fields that have **numeric values based on their bits** similar to other programming languages.

Bitwise Operator	Description	Syntax
$bitsAllClear	Returns documents where all bits in the specified field are 0.	{ field: { $bitsAllClear: <bitmask> } }
$bitsAllSet	Returns documents where all bits in the specified field are 1.	{ field: { $bitsAllSet: <bitmask> } }
$bitsAnySet	Returns documents where at least one bit in the specified field is set (1).	{ field: { $bitsAnySet: <bitmask> } }
$bitsAnyClear	Returns documents where at least one bit in the specified field is clear (0).	{ field: { $bitsAnyClear: <bitmask> } }

In the below example, we also specified the positions we wanted.

Example of Using $bitsAllSet Operator

Let's find the **Person** whose **Age** has bit **1** from position **0 to 4**.

Query:

```
db.count_no.find({"Age":{$bitsAllSet: [0,4] } },{_id:0 } );
```

Output:

```
Data_base> db.count_no.find({"Age":{$bitsAllSet:[0,4]}},{_id:0});
[
  { name: 'Shiva', Age: 25, Likes: 2 },
  { name: 'Rama', Age: 23, Likes: 2 },
  { name: 'Hanuman', Age: 23, Likes: 3 }
]
Data_base>
```

$bitsAllSet in MongoDB

Explanation: In the above query, we have used $**bitsAllSet** and it returns documents whose **bits position from 0 to 4 are only ones.** It works only with the **numeric values**. The numeric values will be converted into the bits and the bits numbering takes place from the right.

7. Geospatial Operators

The Geospatial operators in the MongoDB are used mainly with the terms that relate to the data which mainly focuses on the directions such as **latitude** or **longitudes**.

The Geospatial operators in the MongoDB are:

Geospatial Operator	Description	Syntax
$near	Finds geospatial objects near a point. Requires a geospatial index.	{ $near: { geometry: <point_geometry>, maxDistance: <distance> (optional) } }
$center	(For $geoWithin with planar geometry) Specifies a circle around a center point	{ $geoWithin: { $center: [<longitude>, <latitude>], radius: <distance> } }
$maxDistance	Limits results of $near and $nearSphere queries to a maximum distance from the point.	{ $near: { geometry: <point_geometry>, maxDistance: <distance> } }
$minDistance	Limits results of $near and $nearSphere queries to a minimum distance from the point.	{ $near: { geometry: <point_geometry>, minDistance: <distance> } }

8. Comment Operators

The $comment operator in MongoDB is used to **write the comments along with the query in the MongoDB** which is used to easily understand the data.

Comment Operator Example

Let’s apply some comments in the queries using the **$comment** Operator**.**

Query:

```
db.collection_name.find( { $comment : comment })
```

Output:

```
Data_base> db.count_no.find({$comment:"This is comment"},{_id:0});
[
  { name: 'Krishna', Age: 22, Likes: 1 },
  { name: 'Shiva', Age: 25, Likes: 2 },
  { name: 'Rama', Age: 23, Likes: 2 },
  { name: 'Hanuman', Age: 23, Likes: 3 },
  {
    name: 'Harsha',
    Age: 24,
    Likes: 2,
    Colors: [ 'Red', 'Green', 'Blue' ]
  }
]
Data_base>
```

$Comment operator in MongoDB

Explanation: In the above query we used the $comment operator to mention the comment. We have used “**This is a comment**” with $comment to specify the comment. The comment operator in the MongoDB is used to represent the comment and it increases the understandibility of the code.

MongoDB Projection

MongoDB provides a special feature that is known as **Projection**. It allows you to select only the necessary data rather than selecting whole data from the document. For example, a document contains 5 fields, i.e.,

```
{
name: "Roma",
age: 30,
branch: EEE,
department: "HR",
salary: 20000
}
```

But we only want to display the *name* and the *age* of the employee rather than displaying whole details. Now, here we use projection to display the name and age of the employee.
One can use projection with db.collection.find() method. In this method, the second parameter is the projection parameter, which is used to specify which fields are returned in the matching documents.

Syntax:

```
db.collection.find({}, {field1: value2, field2: value2, ..})
```

- If the value of the field is set to 1 or true, then it means the field will include in the return document.
- If the value of the field is set to 0 or false, then it means the field will not include in the return document.
- You are allowed to use projection operators.
- There is no need to set _id field to 1 to return _id field, the find() method always return _id unless you set a _id field to 0.

Examples:

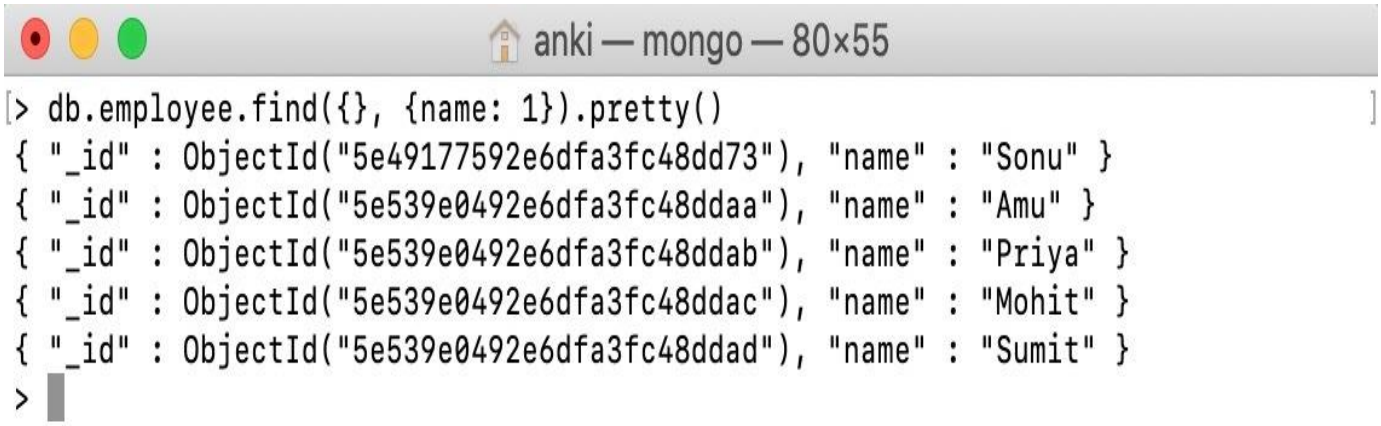
In the following examples, we are working with:

Database: GeeksforGeeks
Collection: employee
Document: five documents that contain the details of the employees in the form of field-value pairs.

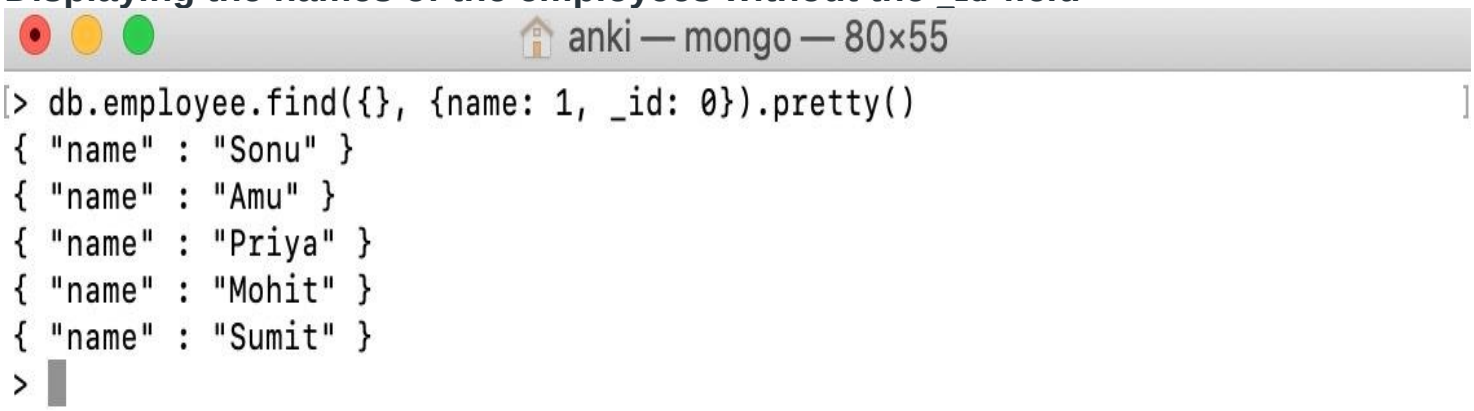
anki — mongo — 80×55

```
> use GeeksforGeeks
switched to db GeeksforGeeks
> db.employee.find().pretty()
{
        "_id" : ObjectId("5e49177592e6dfa3fc48dd73"),
        "name" : "Sonu",
        "age" : 26,
        "branch" : "CSE",
        "department" : "HR",
        "salary" : 44000,
        "joiningYear" : 2018
}
{
        "_id" : ObjectId("5e539e0492e6dfa3fc48ddaa"),
        "name" : "Amu",
        "age" : 24,
        "branch" : "ECE",
        "department" : "HR",
        "joiningYear" : 2017,
        "salary" : 25000
}
{
        "_id" : ObjectId("5e539e0492e6dfa3fc48ddab"),
        "name" : "Priya",
        "age" : 24,
        "branch" : "CSE",
        "department" : "Development",
        "joiningYear" : 2017,
        "salary" : 30000
}
{
        "_id" : ObjectId("5e539e0492e6dfa3fc48ddac"),
        "name" : "Mohit",
        "age" : 26,
        "branch" : "CSE",
        "department" : "Development",
        "joiningYear" : 2018,
        "salary" : 30000
}
{
        "_id" : ObjectId("5e539e0492e6dfa3fc48ddad"),
        "name" : "Sumit",
        "age" : 26,
        "branch" : "ECE",
        "department" : "HR",
        "joiningYear" : 2019,
        "salary" : 25000
}
>
```

Displaying the names of the employees –

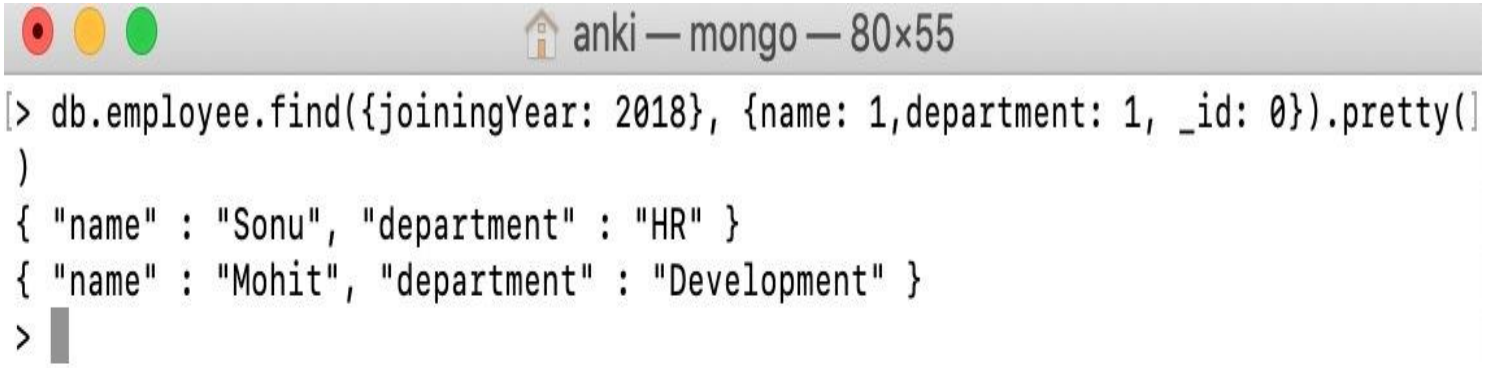


Displaying the names of the employees without the `_id` field –



Displaying the name and the department of the employees without the `_id` field –

Displaying the names and the department of the employees whose joining year is 2018 –



Projection Operators

Name	Description
`$`	Projects the first element in an array that matches the query condition.
`$elemMatch`	Projects the first element in an array that matches the specified `$elemMatch` condition.
`$meta`	Projects the document's score assigned during the `$text` operation.
	NOTE `$text` provides text query capabilities for self-managed (non-Atlas) deployments. For data hosted on MongoDB Atlas, MongoDB offers an improved full-text query solution, Atlas Search.
`$slice`	Limits the number of elements projected from an array. Supports skip and limit slices.

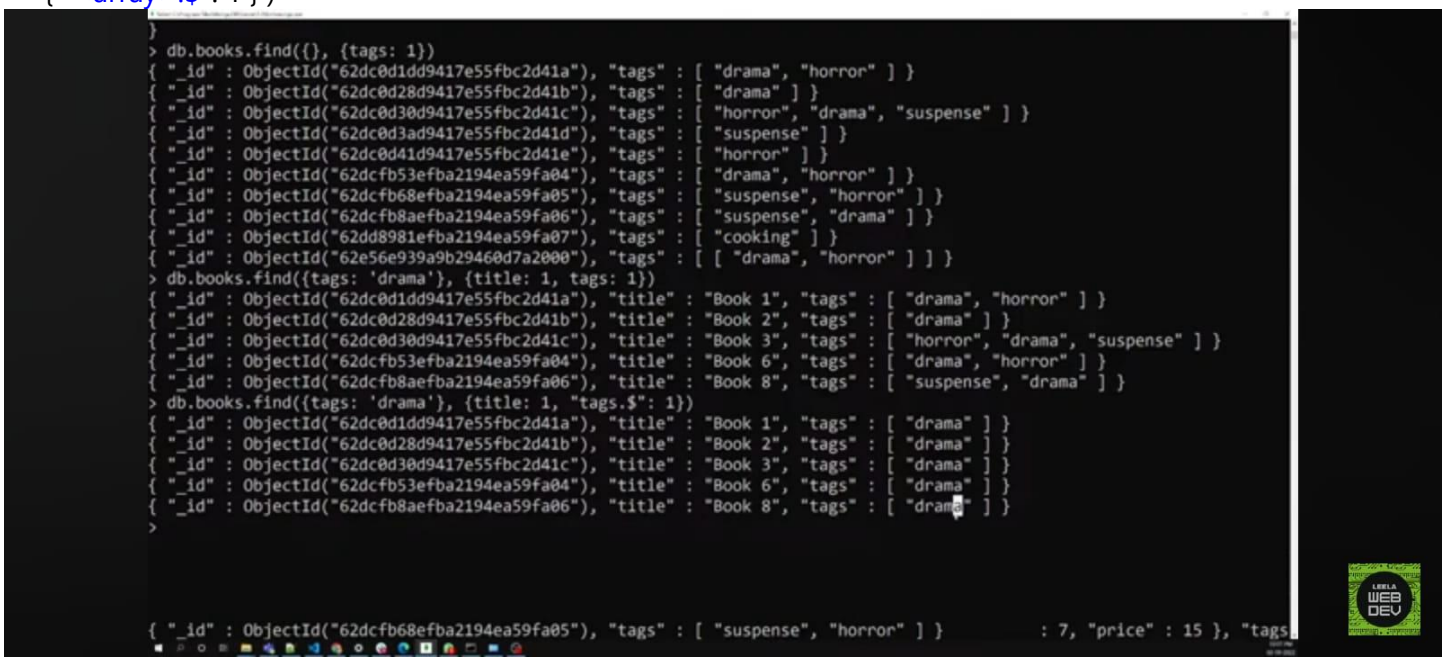
MongoDB Projection Operator

$

The $ operator limits the contents of an array from the query results to contain only the first element matching the query document.

Syntax:

1. db.books.find({ <array>: <value> ... },
2. { "<array>.$": 1 })
3. db.books.find({ <array.field>: <value> ...},
4. { "<array>.$": 1 })

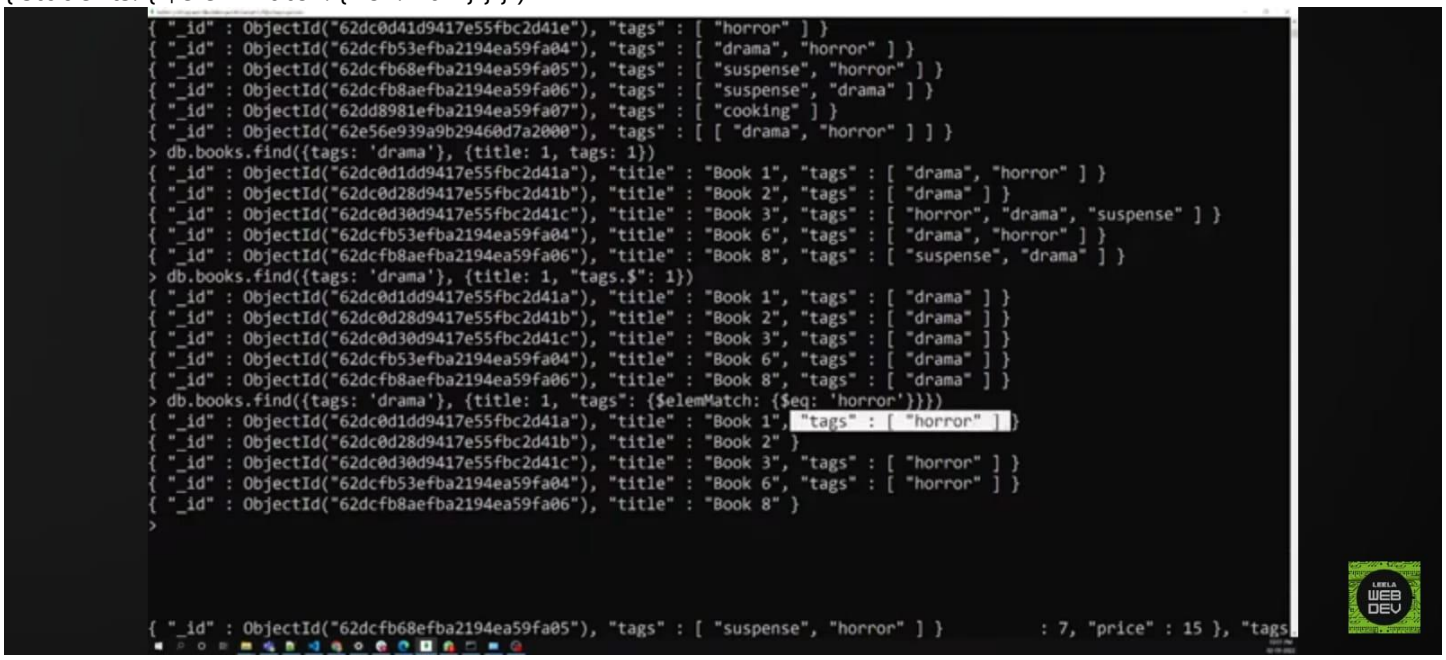


$elemMatch

The content of the array field made limited using this operator from the query result to contain only the first element matching the element $elemMatch condition.

Syntax:

1. db.library.find({ bookcode: "63109" },
2. { students: { $elemMatch: { roll: 102 } } })



$meta

The meta operator returns the result for each matching document where the metadata associated with the query.

Syntax:

1. { $meta: <metaDataKeyword> }

Example:

1. db.books.find(
2. <query>,
3. { score: { $meta: "textScore" } }

$slice

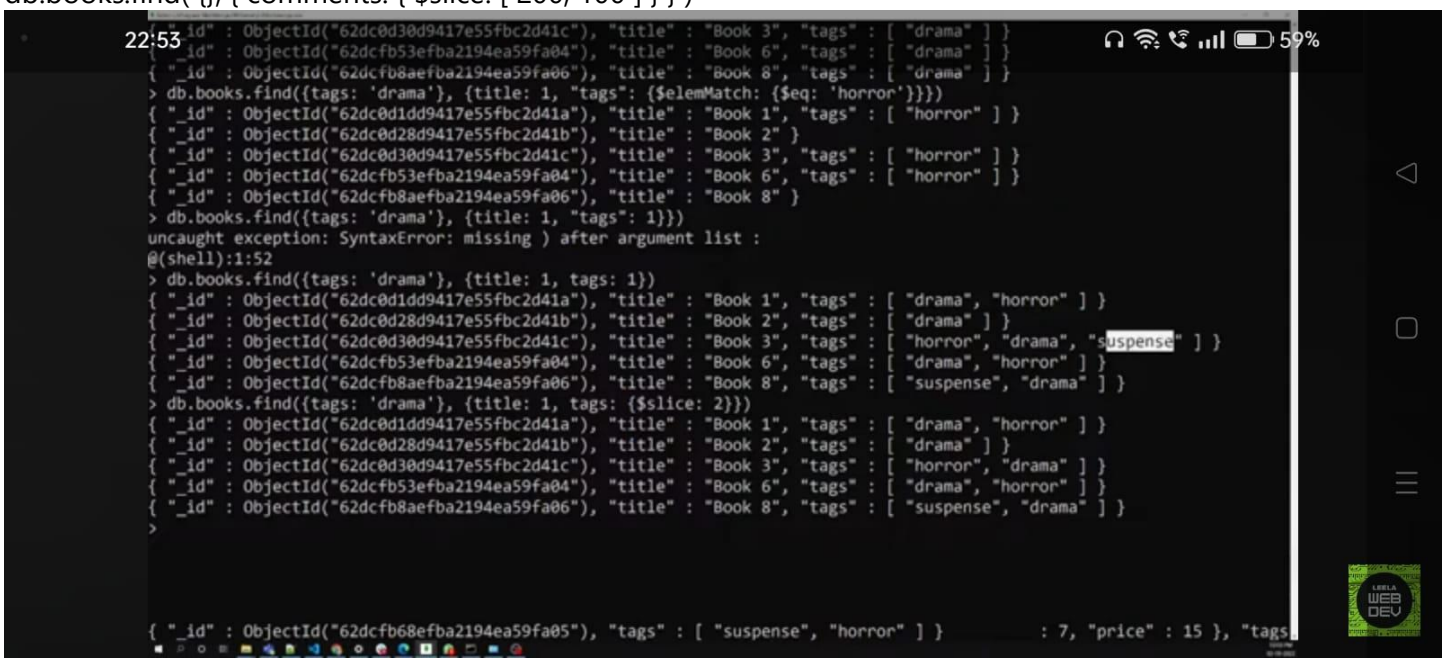
It controls the number of values in an array that a query returns.

Syntax:

1. db.books.find({ field: value }, { array: {$slice: count } });

Example:

1. db.books.find({}, { comments: { $slice: [200, 100] } })



4. Aggregation Pipeline

a. What is Aggregation in MongoDB?

Aggregation is a way of processing a large number of documents in a collection by means of passing them through different stages. The stages make up what is known as a pipeline. The stages in a pipeline can filter, sort, group, reshape and modify documents that pass through the pipeline.

One of the most common use cases of Aggregation is to calculate aggregate values for groups of documents. This is similar to the basic aggregation available in SQL with the GROUP BY clause and COUNT, SUM and AVG functions. MongoDB Aggregation goes further though and can also perform relational-like joins, reshape documents, create new and update existing collections, and so on.

There are what are called **single purpose methods** like `estimatedDocumentCount()`, `count()`, and `distinct()` which are appended to a `find()` query making them quick to use but limited in scope.

- Each stage of the pipeline transforms the documents as they pass through it and allowing for operations like **filtering**, **grouping**, **sorting**, **reshaping** and performing calculations on the data.

b. MongoDB aggregate pipeline syntax

This is an example of how to build an aggregation query:

`db.collectionName.aggregate(pipeline, options)`,

- where *collectionName* – is the name of a collection,
- *pipeline* – is an array that contains the aggregation stages,
- *options* – optional parameters for the aggregation

This is an example of the aggregation pipeline syntax:

```
pipeline = [

    { $match : { … } },

    { $group : { … } },

    { $sort : { … } }

    ]
```

c. Single-purpose aggregation

- It is used when we need simple access to document like counting the number of documents or for finding all distinct values in a document.
- It simply provides the access to the common aggregation process using the **count()**, **distinct()** and **estimatedDocumentCount()** methods so due to which it lacks the flexibility and capabilities of the pipeline.

Example of Single-purpose aggregation

Let's consider a single-purpose aggregation example where we find the total number of users in each city from the `users` collection.

```
db.users.aggregate([
  { $group: { _id: "$city", totalUsers: { $sum: 1 } } }
])
```

Output:

```
[
  { _id: 'Los Angeles', totalUsers: 1 },
  { _id: 'New York', totalUsers: 1 },
  { _id: 'Chicago', totalUsers: 1 }
]
```

In this example, the aggregation pipeline first groups the documents by the `city` field and then uses the `$sum` accumulator to count the number of documents (users) in each city.
The result will be a list of documents, each containing the city (`_id`) and the total number of users (`totalUsers`) in that city.

d. How to use MongoDB to Aggregate Data?

To use MongoDB for aggregating data, follow below steps:

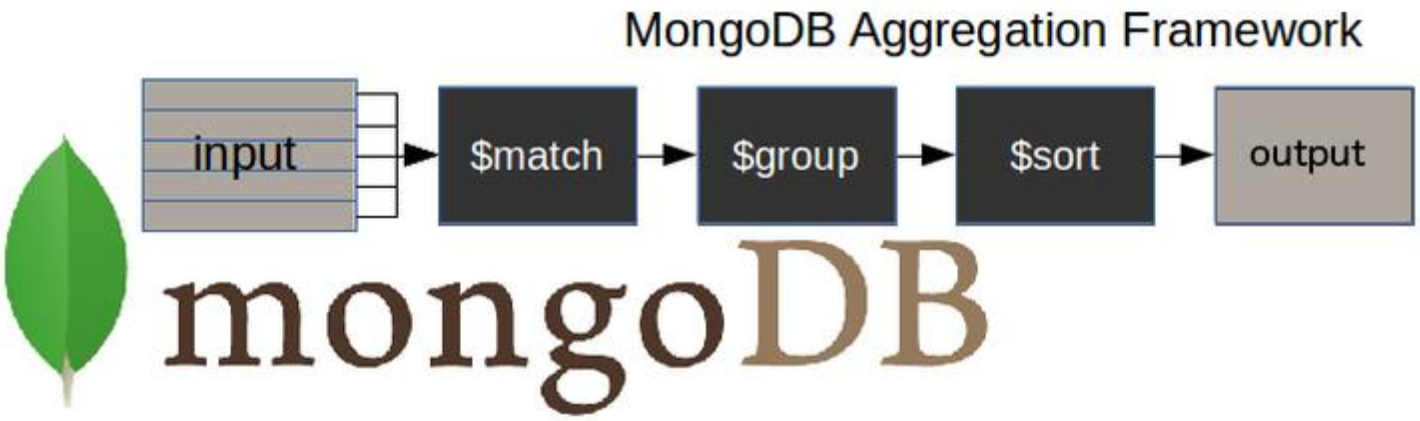
1. **Connect to MongoDB**: Ensure you are connected to your MongoDB instance.
2. **Choose the Collection**: Select the collection you want to perform aggregation on, such as students.
3. **Define the Aggregation Pipeline**: Create an array of stages, like $group to group documents and perform operations (e.g., calculate the average grade).
4. **Run the Aggregation Pipeline**: Use the aggregate method on the collection with your defined pipeline.

Example:

```
db.students.aggregate([
  {
    $group: {
      _id: null,
      averageGrade: { $avg: "$grade" }
    }
  }
])
```

This calculates the average grade of all students in the `students` collection.

e. Mongodb Aggregation Pipeline

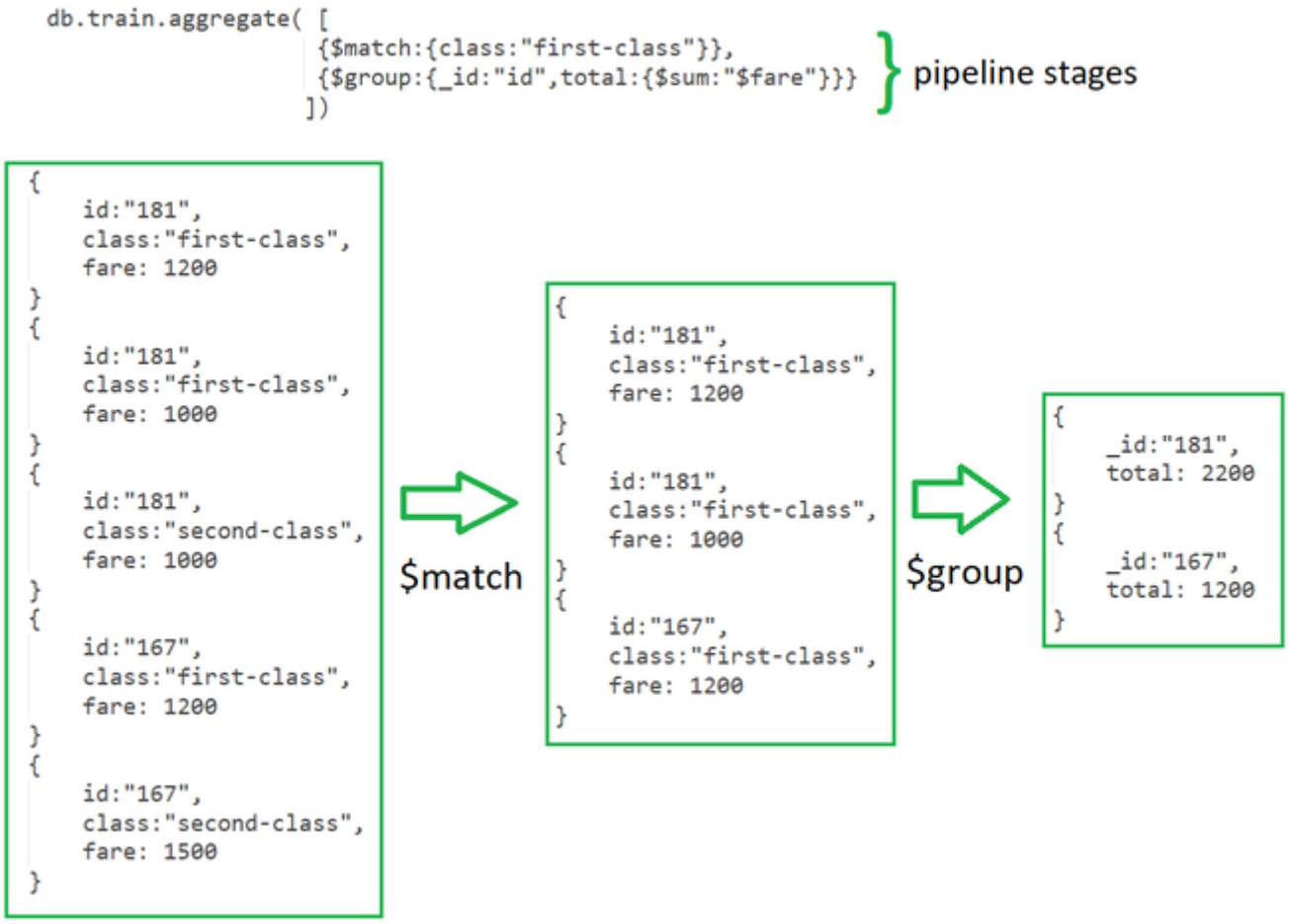


- **Mongodb Aggregation Pipeline** consist of stages and each stage transforms the document. It is a **multi-stage pipeline** and in each state and the documents are taken as input to produce the resultant set of documents.
- In the next stage (ID available) the resultant documents are taken as input to produce output, this process continues till the last stage.
- The **basic pipeline stages** are defined below:
  1. filters that will operate like queries.

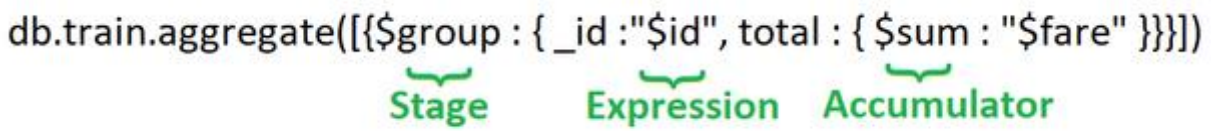
2. the document transformation that modifies the resultant document.
3. provide pipeline provides tools for **grouping and sorting documents.**

- Aggregation pipeline can also be used in **sharded collection**.

Example:



Explanation:
In the above example of a collection of "train fares". **$match** stage filters the documents by the value in class field i.e. class: "first-class" in the first stage and passes the document to the second stage.
In the Second Stage, the **$group stage** groups the documents by the id field to calculate the sum of fare for each unique id.
Here, the **aggregate() function** is used to perform aggregation. It can have three operators **stages** , **expression** and **accumulator.** These operators work together to achieve final desired outcome.



f. Aggregation Pipeline Method

To understand Aggregation Pipeline Method Let's imagine a collection named **users** with some documents for our examples.

```
{
 "_id": ObjectId("60a3c7e96e06f64fb5ac0700"),
 "name": "Alice",
 "age": 30,
 "email": "alice@example.com",
```

```
  "city": "New York"
}
{
  "_id": ObjectId("60a3c7e96e06f64fb5ac0701"),
  "name": "Bob",
  "age": 35,
  "email": "bob@example.com",
  "city": "Los Angeles"
}
{
  "_id": ObjectId("60a3c7e96e06f64fb5ac0702"),
  "name": "Charlie",
  "age": 25,
  "email": "charlie@example.com",
  "city": "Chicago"
}
```

0. **$group:** It Groups documents by the `city` field and calculates the average age using the **$avg** accumulator.

```
db.users.aggregate([
  { $group: { _id: "$city", averageAge: { $avg: "$age" } } }
])
```

Output:

```
[
  { _id: 'New York', averageAge: 30 },
  { _id: 'Chicago', averageAge: 25 },
  { _id: 'Los Angeles', averageAge: 35 }
]
```

1. **$project**: Include or exclude fields from the output documents.

```
db.users.aggregate([
  { $project: { name: 1, city: 1, _id: 0 } }
])
```

Output:

```
[
  { name: 'Alice', city: 'New York' },
  { name: 'Bob', city: 'Los Angeles' },
  { name: 'Charlie', city: 'Chicago' }
]
```

2. **$match**: Filter documents to pass only those that match the specified condition(s).

```
db.users.aggregate([
  { $match: { age: { $gt: 30 } } }
])
```

Output:

```
[
  {
    _id: ObjectId('60a3c7e96e06f64fb5ac0701'),
    name: 'Bob',
    age: 35,
    email: 'bob@example.com',
    city: 'Los Angeles'
```

```
  }
]
```

3. **$sort**: It Order the documents.

```
db.users.aggregate([
  { $sort: { age: 1 } }
])
```

Output:

```
[
  {
    _id: ObjectId('60a3c7e96e06f64fb5ac0702'),
    name: 'Charlie',
    age: 25,
    email: 'charlie@example.com',
    city: 'Chicago'
  },
  {
    _id: ObjectId('60a3c7e96e06f64fb5ac0700'),
    name: 'Alice',
    age: 30,
    email: 'alice@example.com',
    city: 'New York'
  },
  {
    _id: ObjectId('60a3c7e96e06f64fb5ac0701'),
    name: 'Bob',
    age: 35,
    email: 'bob@example.com',
    city: 'Los Angeles'
  }
]
```

4. **$limit**: Limit the number of documents passed to the next stage.

```
db.users.aggregate([
  { $limit: 2 }
])
```

Output:

```
[
  {
    _id: ObjectId('60a3c7e96e06f64fb5ac0700'),
    name: 'Alice',
    age: 30,
    email: 'alice@example.com',
    city: 'New York'
  },
  {
    _id: ObjectId('60a3c7e96e06f64fb5ac0701'),
    name: 'Bob',
    age: 35,
    email: 'bob@example.com',
    city: 'Los Angeles'
  }
]
```

g. How Fast is MongoDB Aggregation?

- The speed of MongoDB aggregation depends on various factors such as the complexity of the aggregation pipeline, the size of the data set, the hardware specifications of the MongoDB server and the efficiency of the indexes.
- In general, MongoDB's aggregation framework is designed to efficiently process large volumes of data and complex aggregation operations. When used correctly it can provide fast and scalable aggregation capabilities.
- So with any database operation, the performance can vary based on the specific use case and configuration. It is important to optimize our aggregation queries and use indexes where appropriate and ensure that our MongoDB server is properly configured for optimal performance.

How to Insert a Document into a MongoDB Collection using Node.js?

MongoDB, a popular NoSQL database, offers flexibility and scalability for handling data. If you're developing a Node.js application and need to interact with MongoDB, one of the fundamental operations you'll perform is inserting a document into a collection. This article provides a step-by-step guide on how to accomplish this using Node.js.

Prerequisites:

- NPM
- NodeJS
- MongoDB

The steps to insert documents in MongoDB collection are given below

Table of Content

- NodeJS and MongoDB Connection
- Create a Collection in MongoDb using Node Js
- Insert a Single Document
- Insert Many Document
- Handling Insertion Results
- Read Documents from the collection

Steps to Setup the Project

Step 1: Create a nodeJS application by using this command

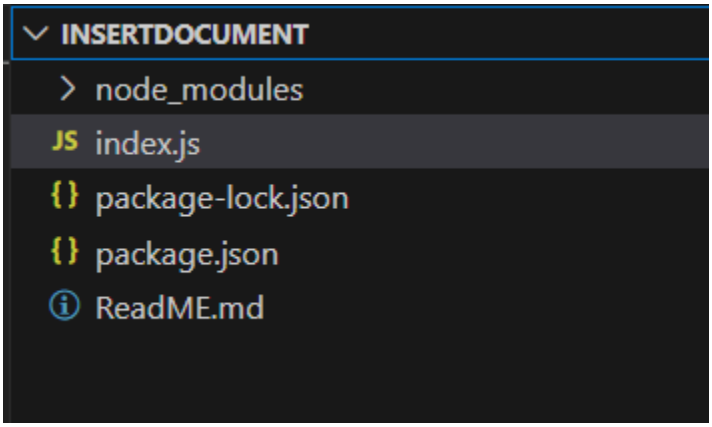
```
npm init
or
npm init -y
```

- **npm init** command asks some setup questions that are important for the project
- **npm init -y** command is used to set all the answers of the setup questions as **yes**.

Step 2: Install the necessary packages/libraries in your project using the following commands.

```
npm install mongodb
```

Project Structure:



Project Structure

The updated dependencies in package.json file will look like:

```
"dependencies": {
   "mongodb": "^6.6.1"
  }
```

NodeJS and MongoDB Connection

Once the MongoDB is installed we can use MongoDB database with the Nodejs Project.Initially we need to specify the database name ,connection URL and the instance of MongoDBClient.

```
const { MongoClient } = require('mongodb');
// or as an ecmascript module:
// import { MongoClient } from 'mongodb'

// Connection URL
const url = 'mongodb://localhost:27017';
const client = new MongoClient(url);

const dbName = 'project_name';    // Database Name

async function main() {

  await client.connect();
  console.log('Connected successfully to server');

  const db = client.db(dbName);
  const collection = db.collection('collection_name');

//Can Add the CRUD operations
    }

main() .then(console.log)
            .catch(console.error)
             .finally(() => client.close());
```

- **MongoClient** class provided method, to connect MongoDB and Nodejs.
- **client** is the instance of MongoDb and Node Js connection.
- **client.connect()** is used to connect to MongoDB database ,it **awaits** until the the connection is established.

Create a Collection in MongoDb using Node Js

In this operation we create a collection inside a database.Intilally we specify the database in which collections is to be created.

```
//Sepcify Database
const dbName = 'database_name';

const db = client.db(dbName);

//Create Collection
const collection = db.collection('collection_name');
```

- **client** is the instance of the connection which provides the **db**() method to create a new Database.
- **collection**() method is used to set the instance of the collection .

Insert a Single Document

To insert a document into the collection **insertOne**() method is used.

```
const insertDoc = await collection.insertOne({
   filed1: value1,
   field2: value2,
);
```

```
//Insert into collection
console.log('Inserted documents =>', insertDoc);
```

Insert Many Document

To insert a document into the collection insertMany() method is used.

```
const doc_array = [
   { document1 },
   { document2 },
   { document3 },
];

//Insert into collection
const insertDoc =
   await collection.insertMany(doc_array);
console.log('Inserted documents =>', insertDoc);
```

Handling Insertion Results

In a project we have different tasks which needs to be executed in specific order.In the MongoDB and Node Js project we must ensure that connection is set.While performing insertion of documents , we perform asynchronous insertion so that execution is not interrupted.We use try-catch block to handle errors while setting up connection, inserting document or while performing any other operation. If an error occurs during execution ,catch block handles it or provide the details about the error ,which helps to resolve the error.

```
try {
   const dbName = 'database_name';
   await client.connect();
   const collection = db.collection('collection_name');

   const doc_array = [
      { document1 },
      { document2 },
      { document3 },
   ];

//Insert into collection
const insertDoc = await collection.insertMany(doc_array);

   console.log('Inserted documents =>', insertDoc);
} catch (error) {
   console.error('Error:', error);
}
```

- Initally connection is established .AS the connection is established insertMany() method or insertOne() method is used to insert the document in the collection.
- **insertDoc** stores the result of the insertion which is further logged.

Read Documents from the collection

We can read the documents inside the collection using the find() method.

```
const doc = await collection.find({}).toArray();
console.log('Found documents =>', doc    );
```

find() method is used to along with empty {} are used to read all the documents in the collection.Which are further converted into the array using the toArray() method.

Closing the Connection

```
finally{
client.close()
}
```

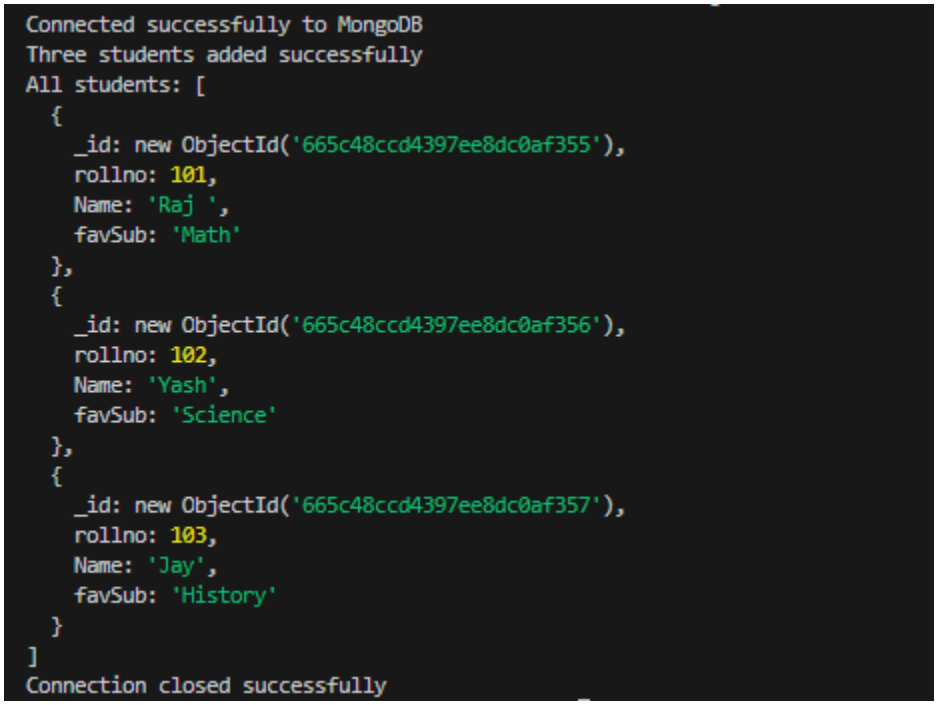
- Once the promise is resolved or rejected , code in finally block is executed. The **close()** method is used to close the connection.
- Connection is closed irrespective of the error .It is generally used to cleanup and release the resource.

Example: Implementation to show Insertion of documents into a MongoDB collection using Node.js

JavaScript

```
const { MongoClient } = require("mongodb");

async function main() {
    const url = "mongodb://127.0.0.1:27017";
    const dbName = "GeeksforGeeks";
    const studentsData = [
        { rollno: 101, Name: "Raj ", favSub: "Math" },
        { rollno: 102, Name: "Yash", favSub: "Science" },
        { rollno: 103, Name: "Jay", favSub: "History" },
    ];

    let client = null;

    try {
        // Connect to MongoDB
        client = await MongoClient.connect(url);
        console.log("Connected successfully to MongoDB");

        const db = client.db(dbName);
        const collection = db.collection("students");

        // Add students to the database
        await collection.insertMany(studentsData);
        console.log("Three students added successfully");

        // Query all students from the database
        const students = await collection.find().toArray();
        console.log("All students:", students);
    } catch (err) {
        console.error("Error:", err);
    } finally {
        // Close the connection
        if (client) {
            client.close();
            console.log("Connection closed successfully");
        }
    }
}

main();
```

Output:



Insert Document in MongoDB

Explanation :

In the above example, Initially **MongoClient** class is imported which is used to connect MongoDB and Nodejs .**client** is the instance of MongoDb and Node Js connection. which is used to name the database .As database is set **,collection()** method sets the instance of the collection .Three documents are inserted in the **students** collection using **insertMany**() method .Error during the execution are handled using the try catch block ,finally connection is closed using the **close() method**