

# MEAN Full Stack

---

**Amir Hussain**  
**Cyber security trainer**  
**&**  
**Full Stack Developer**





## CHAPTER-2nd

### Introduction



## NoSQL database

NoSQL Database is used to refer a non-SQL or non-relational database.

It provides a mechanism for storage and retrieval of data other than tabular relations model used in relational databases.

NoSQL database doesn't use tables for storing data. It is generally used to store big data and real-time web applications.

NoSQL databases are a class of database management systems that differ from traditional relational databases in their data models, schemas, and scalability options



# History behind the creation of NoSQL Databases

In the early 1970, Flat File Systems are used. Data were stored in flat files and the biggest problems with flat files are each company implement their own flat files and there are no standards. It is very difficult to store data in the files, retrieve data from files because there is no standard way to store data.

Then the relational database was created by E.F. Codd and these databases answered the question of having no standard way to store data. But later relational database also get a problem that it could not handle big data, due to this problem there was a need of database which can handle every types of problems then NoSQL database was developed.





# Characteristics of NoSQL Databases

**Flexibility:** NoSQL databases are schema-less, allowing for flexible data models.

**Scalability:** Designed to scale horizontally to handle large amounts of data and traffic.

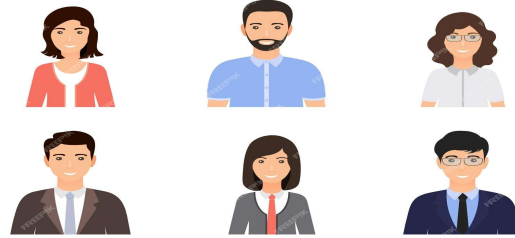
**Variety of Data Models:** Support for various data models like document-oriented, key-value, column-family, and graph databases.

**High Performance:** Optimized for read and write-intensive workloads.



# Types of NoSQL Databases

- Document-Oriented: e.g. MongoDB
- Key-Value Stores: e.g. Redis
- Column-Family Stores: e.g. Apache Cassandra
- Graph Databases: e.g. Neo4j
- Object-Oriented Databases



## Use Cases

Real-time Big Data Applications

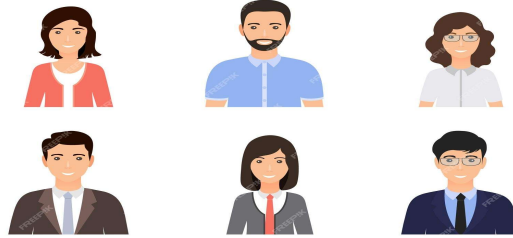
Content Management Systems (CMS)

Mobile App Development

Web App

IoT (Internet of Things) Applications

Social Media Analytics



# MongoDB

- **Definition:** MongoDB is a popular open-source NoSQL document-oriented database designed for flexibility, scalability, and ease of development.
- **Overview:** It stores data in flexible, JSON-like documents, known as BSON (Binary JSON)





## Key Features

NoSQL: Not restricted to the traditional relational database model.

Schema-less: Allows dynamic and flexible data schema.

Horizontal Scalability: Easily scales horizontally to handle large datasets.

Rich Query Language: Supports a variety of queries and indexing options.

High Performance: Optimized for read and write-intensive workloads.



# Document-Oriented Data Model

**Overview:** MongoDB uses a document-oriented data model, where data is stored in BSON documents.

**Documents:** JSON-like structures with key-value pairs.

**Collections:** A group of MongoDB documents, equivalent to a table in a relational database.

**Definition:** BSON (Binary JSON) is a binary-encoded serialization of JSON-like documents.

**Structure:** Similar to JSON but includes additional data types like Date, Binary, and ObjectId.

**Efficient Storage:** Compact and efficient for storage and transmission.

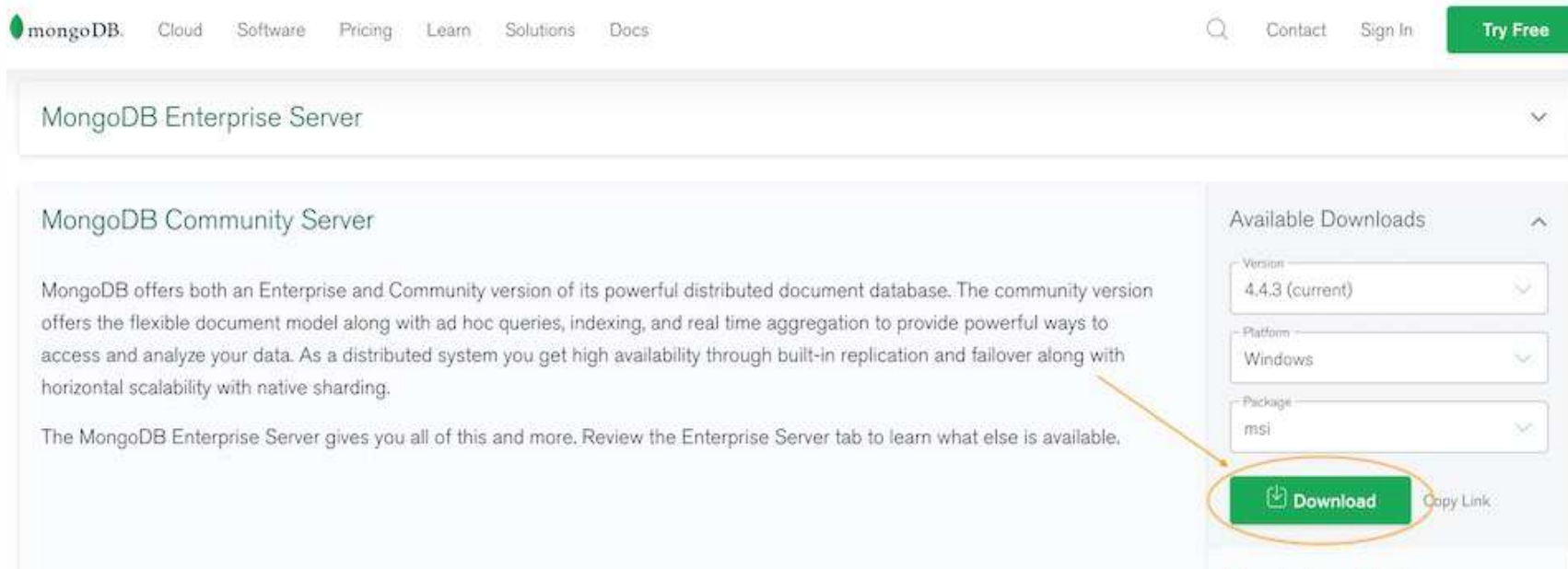


# MongoDB Data type

Data Types	Description
String	String is the most commonly used datatype. It is used to store data. A string must be UTF 8 valid in mongodb.
Integer	Integer is used to store the numeric value. It can be 32 bit or 64 bit depending on the server you are using.
Boolean	This datatype is used to store boolean values. It just shows YES/NO values.
Double	Double datatype stores floating point values.
Min/Max Keys	This datatype compare a value against the lowest and highest bson elements.
Arrays	This datatype is used to store a list or multiple values into a single key.
Object	Object datatype is used for embedded documents.
Null	It is used to store null values.
Symbol	It is generally used for languages that use a specific type.
Date	This datatype stores the current date or time in unix time format. It makes you possible to specify your own date time by creating object of date and pass the value of date, month, and year into it.

# Installation and configuration of MongoDB

**Step 1:** Go to [MongoDB Download Center](#) to download MongoDB Community Server. Here, You can select any version, Windows, and package according to your requirement

A screenshot of the MongoDB Download Center website. The page has a navigation bar with links for Cloud, Software, Pricing, Learn, Solutions, and Docs. A search bar and links for Contact, Sign In, and Try Free are also present. The main content area shows the MongoDB Enterprise Server tab selected. Below it, the MongoDB Community Server section is visible, featuring a description of the community version and a list of available downloads. The available downloads section includes dropdown menus for Version (4.4.3 (current)), Platform (Windows), and Package (msi). A green Download button is highlighted with a red circle and an arrow pointing to it from the text description. A Copy Link button is also visible next to the Download button.

mongoDB. Cloud Software Pricing Learn Solutions Docs

Search Contact Sign In Try Free

MongoDB Enterprise Server

MongoDB Community Server

MongoDB offers both an Enterprise and Community version of its powerful distributed document database. The community version offers the flexible document model along with ad hoc queries, indexing, and real time aggregation to provide powerful ways to access and analyze your data. As a distributed system you get high availability through built-in replication and failover along with horizontal scalability with native sharding.

The MongoDB Enterprise Server gives you all of this and more. Review the Enterprise Server tab to learn what else is available.

Available Downloads

Version: 4.4.3 (current)

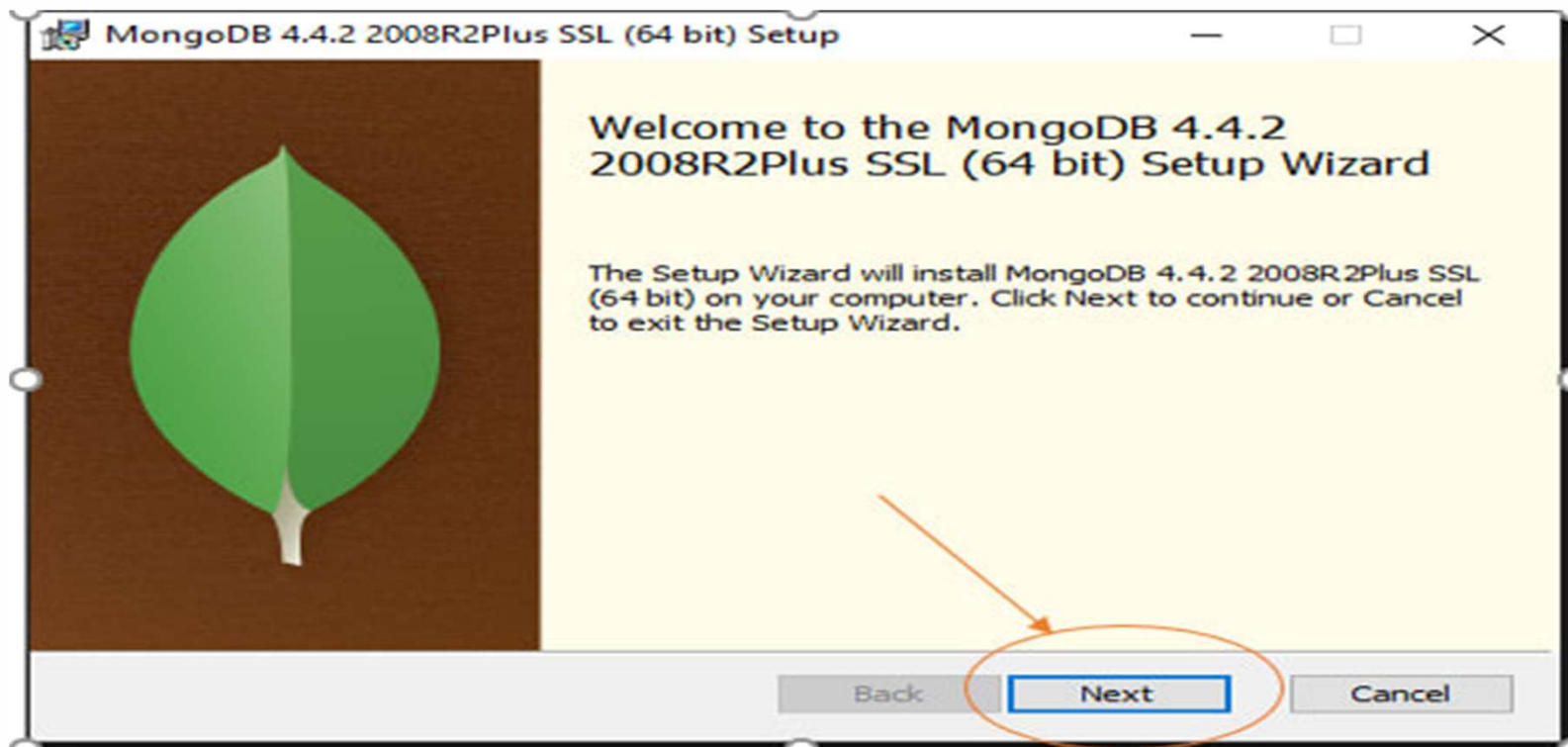
Platform: Windows

Package: msi

Download Copy Link

## Contn...

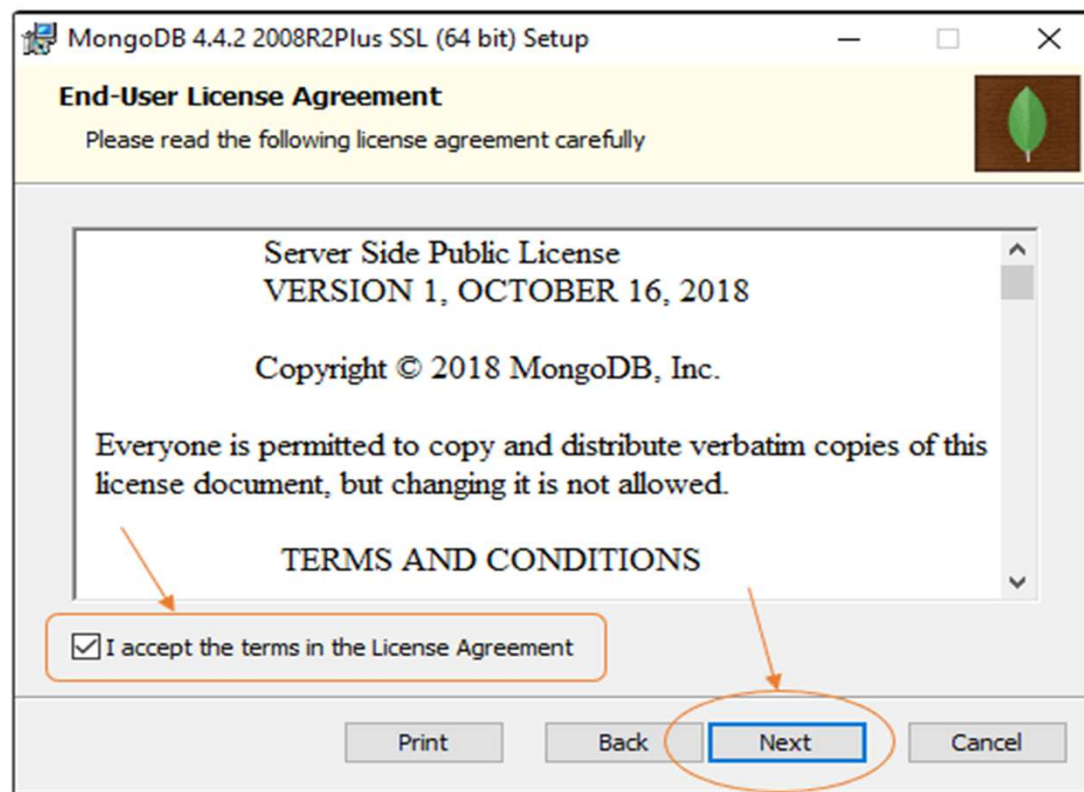
**Step 2:** When the download is complete open the msi file and click the *next button* in the startup screen:





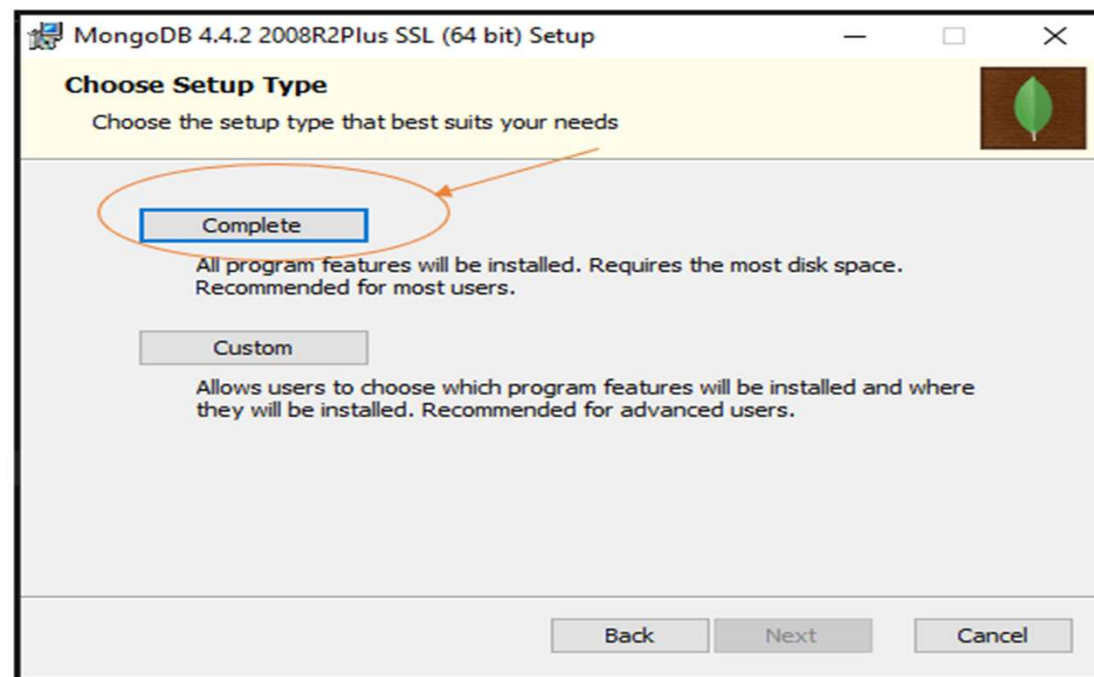
## Contn...

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



## Contn...

**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*:



## Contn...

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

MongoDB 4.4.2 2008R2Plus SSL (64 bit) Service Customization

### Service Configuration

Specify optional settings to configure MongoDB as a service.

☒ Install MongoDB as a Service

☒ Run service as Network Service user

☐ Run service as a local or domain user:

Account Domain:

Account Name:

Account Password:

Service Name:

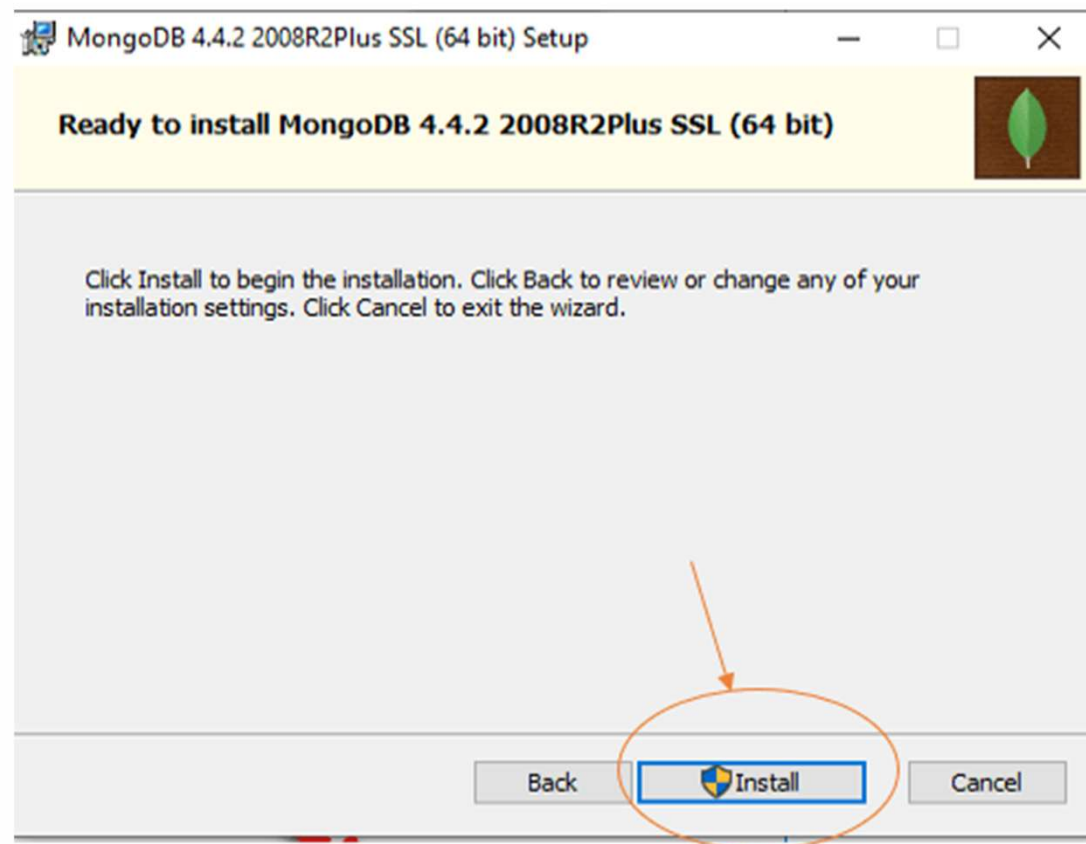
Data Directory:

Log Directory:

< Back   Next >   Cancel

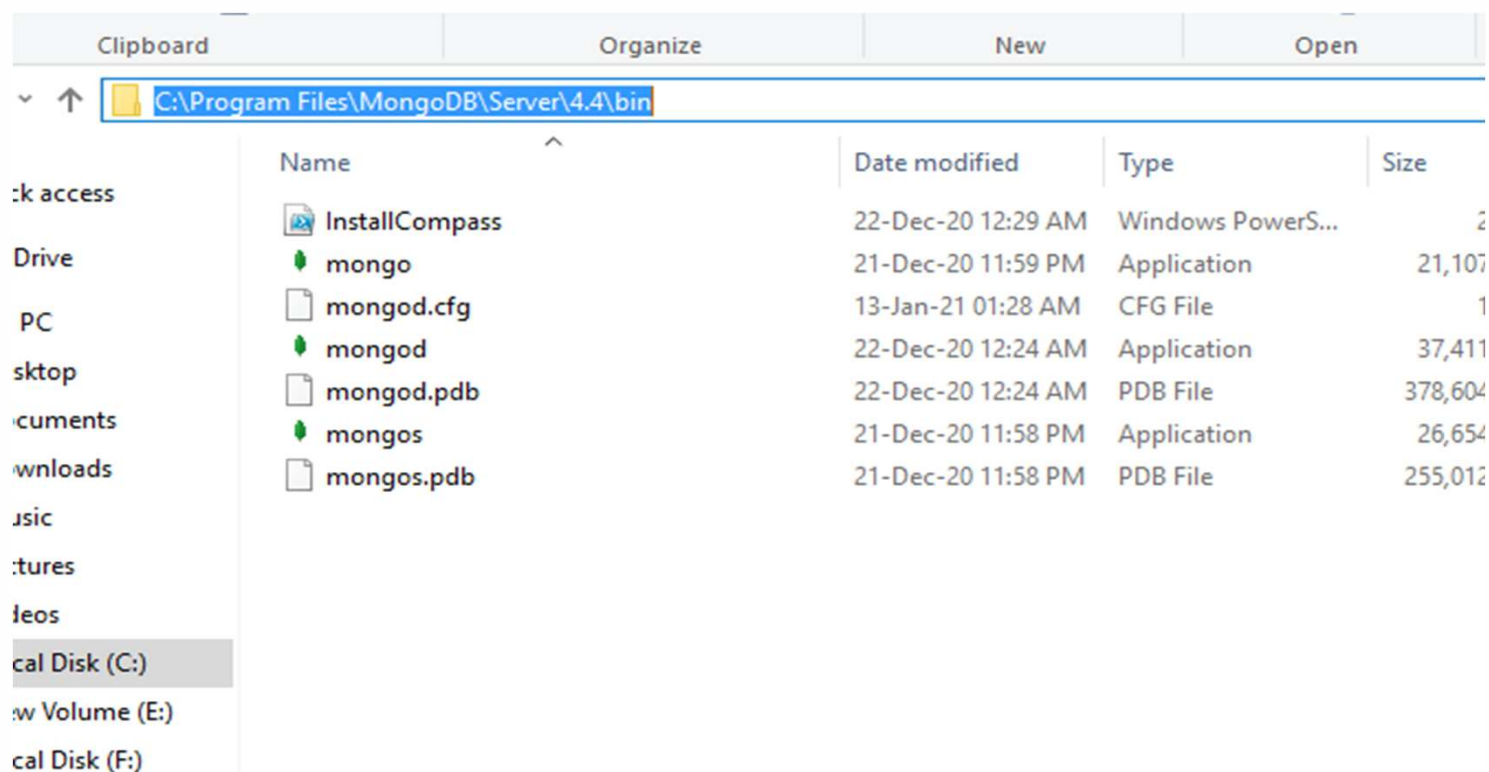
## Contn...

**Step 6:** Click the *Install* button to start the installation process and then click Finish:



## Contn...

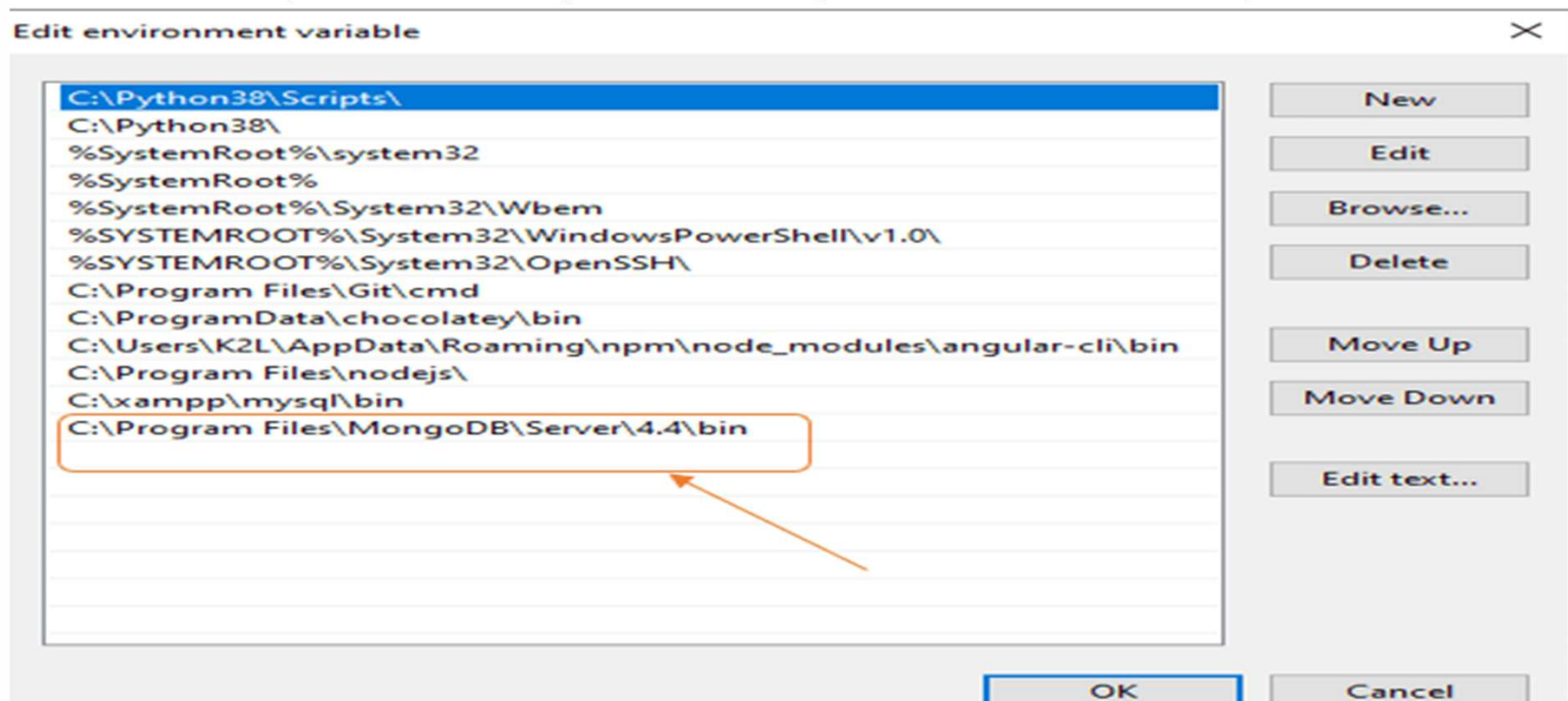
**Step 7:** Now we go to the location where MongoDB installed in step 5 in your system and copy the bin path:





## Contn...

**Step 8:** 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:





## Contn...

**Step 9:** Now, Open C drive and create a folder named “data” inside this folder create another folder named “db”. After creating these folders. open the command prompt and run the following command:

**Command: mongod**

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

## Contn...

**Step 10:** 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 be able to connect with the mongo shell.

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



## CRUD in MongoDB

CREATE, READ, UPDATE, and DELETE actions are referred to as CRUD operations in MongoDB. These are the basic operations used to alter data in databases. In MongoDB, the term "CRUD operations" refers to the standard set of database operations used to work with collections of data.

**Create:** To add new documents to a collection, use the Create operation.

**Read:** Data from a collection is retrieved using the Read operation.

**Update:** The Update operation is used to edit existing documents in a collection.

**Delete:** A collection of documents can be deleted using the Delete procedure.



## Contn...

To create a new database, you can simply run any command against a non-existing database, and MongoDB will automatically create it for you.

If Database already exist then it use same existing database

Let us create a database name test

**use test;** (execute this command to create database)





## Contn...

### Creating a New Collection:

To create a new collection, you can use the "createCollection" method.

**Syntax: db.createCollection(name, options)**

eg:

```
db.createCollection("dummy")
```

## Contn...

### Create Operation

There are two ways to create new documents to a collection in MongoDB:

#### 1. **insertOne():**

Adding a single document to a collection is done using this method. A document to be added is the only argument it accepts, and it returns a result object with details about the insertion.

**Syntax: db.collectionName.insertOne()**

**eg:**

```
db.dummy.insertOne({  
  name: "RAM",  
  age: 25,  
  ...  
});
```

## Contn...

### 2. insertMany()

This method is used to insert multiple documents into a collection at once. It takes an array of documents as its argument and returns a result object that contains information about the insertion.

**Syntax: db.collectionName.insertMany();**

**eg:**

```
db.users.insertMany([
  {
    name: "Ram",
    age: 27,
  },
  {
    name: "Sham",
    age: 30,
  },
]);
```



## Contn...

### Read Operations

In MongoDB, read operations are used to retrieve data from the database.

#### 1. find()

The find() method is used to retrieve data from a collection. It returns a cursor that can be iterated to access all the documents that match the specified query criteria.

**Syntax:** `db.collectionName.find(query, projection)`

**Eg:**

1. `db.dummy.find()`
2. `db.dummy.find({ age: { $gt: 29 } }, { name: 1, age: 1 })`

This command will return all the documents in the “dummy” collection where the age is greater than 29, and only return the “name” and “age” fields.



## Contn...

### 2. findOne()

The findOne() method returns a single document object, or null if no document is found. You can pass a query object to this method to filter the results.

**Syntax:** `db.collectionName.findOne()`

Eg:

```
db.dummy.findOne({ name: "Ram" })
```





## Contn...

### 1. **limit()** Method:

Limits the number of documents returned by a MongoDB query.

**query: db.students.find().limit(5);**

Limits the result set to the first 5 documents returned by the query.

### 2. **skip()** Method:

Skips a specified number of documents in a MongoDB query result set.

**query: db.students.find().skip(10);**

Skips the first 10 documents in the result set, returning the subsequent documents.



## Contn...

### Update Operations

In MongoDB, the "update" operation is used to modify existing documents in a collection.

#### 1. updateOne()

The updateOne() method is used to update a single document that matches a specified filter.

**Syntax: db.collectionName.updateOne(filter, update, options)**

**Eg:**

```
db.dummy.updateOne({ name: "Ram" }, { $set: { age: "30" } })
```



## Contn...

**Following are a few of the many available operations:**

**\$set:** Sets the value of a field in a document. If the field does not exist, the set will create it.

**\$unset:** Removes a field from a document.

**\$inc:** Increments the value of a field in a document by a specified amount.

**\$push:** Adds an element to the end of an array field in a document. If the field does not exist, push will create it as an array with the specified element.

**\$pull:** Removes all occurrences of a specified value from an array field in a document.



## Contn...

### 2. updateMany

The updateMany() method is used to update multiple documents that match a specified filter.

**Syntax:**

**db.collectionName.updateMany(filter, update, options)**

**Eg:**

**db.dummy.updateMany({ age: { \$lt: 30 } }, { \$set: { age: 28} })**

This command will update the status of all documents in the “dummy” collection where the age is less than 30 to age 28.



## Contn...

### 3. replaceOne():

**Replaces a single document with a new document.**

```
db.students.replaceOne(  
  { name: "Bob" },  
  { name: "Bob", age: 29, grade: "A-" }  
);
```



## Contn...

### Delete Operations

In MongoDB, the "delete" operation is used to remove documents from a collection.

**There are several ways to perform a delete operation, including the following:**

#### 1. deleteOne()

The deleteOne() method is used to remove a single document that matches a specified filter.

**Syntax: db.collectionName.deleteOne(filter, options)**

Eg:

```
db.dummy.deleteOne({ name: "ram" })
```

This command will remove the first document in the "dummy" collection where the name is "ram"



## Contn...

### 2. deleteMany()

The deleteMany() method is used to remove multiple documents that match a specified filter.

**Syntax:**

**db.collectionName.deleteMany(filter, options)**

**Eg:**

**db.dummy.deleteMany({ age: { \$lt: 30 } })**

This command will remove all documents in the "users" collection where the age is less than 30





## Contn...

### 3. drop()

The drop() method is used to remove an entire collection.

**Syntax:**

**db.collectionName.drop()**

**Eg:**

**db.dummy.drop()**

This command will remove the dummy collection.



# Indexing and querying in MongoDB

Indexes support the efficient resolution of queries. Without indexes, MongoDB must scan every document of a collection to select those documents that match the query statement. This scan is highly inefficient and require MongoDB to process a large volume of data.

Indexes are special data structures, that store a small portion of the data set in an easy-to-traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field as specified in the index.

To create an index, you need to use `createIndex()` method of MongoDB.

Syntax: **`db.collectionName.createIndex({KEY:1})`**



## Contn...

Here key is the name of the field on which you want to create index and 1 is for ascending order. To create index in descending order you need to use -1.

**Eg; db.dummy.createIndex({"title":1})**

In **createIndex()** method you can pass multiple fields, to create index on multiple fields.

**Eg; db.dummy.createIndex({"title":1, "description":1})**

## Contn...

### The dropIndex() method

You can drop a particular index using the dropIndex() method of MongoDB.

Syntax: **Eg; db.collectionName.dropIndex({"KEY":1})**

**Eg: Eg; db.dummy.createIndex({"title":1})**



# Schema design and data modeling

## MongoDB Schema Design

Now, MongoDB schema design works a lot differently than relational schema design. With MongoDB schema design, there is:

No formal process

No algorithms

No rules



## Schema design and data modeling

When you are designing your MongoDB schema design, the only thing that matters is that you design a schema that will work well for your application. Two different apps that use the same exact data might have very different schemas if the applications are used differently. When designing a schema, we want to take into consideration the following:

- Store the data

- Provide good query performance

- Require reasonable amount of hardware



# Schema design and data modeling

```
{
  _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: TEST,
      dateCreated: DATE_TIME,
      like: LIKES
    }
  ]
}
```





## Schema design and data modeling

You can see that instead of splitting our data up into separate collections or documents, we take advantage of MongoDB's document based design to embed data into arrays and objects within the User object. Now we can make one simple query to pull all that data together for our application.



# Schema design and data modeling

MongoDB schema design actually comes down to only two choices for every piece of data. You can either embed that data directly or reference another piece of data using the \$lookup operator (similar to a JOIN). Let's look at the pros and cons of using each option in your schema.

## 1.Embedding

You can retrieve all relevant information in a single query.

Avoid implementing joins in application code or using \$lookup.

Update related information as a single atomic operation.



# Schema design and data modeling

## 2. Referencing

The other option for designing our schema is referencing another document using a document's unique ID and connecting them together using the \$lookup operator. Referencing works similarly as the JOIN operator in an SQL query. It allows us to split up data to make more efficient and scalable queries, yet maintain relationships between data.

By splitting up data, you will have smaller documents.

Infrequently accessed information not needed on every query.

Reduce the amount of duplication of data. However, it's important to note that data duplication should not be avoided if it results in a better schema.



# Schema design and data modeling

## Type of Relationships

### 1. One-to-one

Using embedded documents we can create one-to-one relationships between the data so, that we can easily retrieve data using few read operations

```
// Student document {   StudentName: Ishan,  
                        StudentId: 123_2022,  
                        Branch:CSE }
```

```
// Address document  
{   StudentName: Ishan,  
    PremanentAddress: XXXXXXXX,  
    City: Jaipur,  
    PinCode:302022 }
```



## Schema design and data modeling

If the address data is frequently used, the user creates a query using Student Name to retrieve the data from the address document. However, because two documents contain the same field (i.e., StudentName), the user must write a few more queries to retrieve the required information. This data retrieval procedure is time-consuming. As a result, the address document was embedded in the student document.

```
{  
  StudentName: Ishan,  
  StudentId: 123_2020,  
  Branch:CSE  
  PermanentAddress:{  
    PremanentAddress:XXXXXXX,  
    City: Jaipur,  
    PinCode:302022    } }  
}
```



# Schema design and data modeling

## 2. One-to-Many

We can create one-to-many relationships between data using embedded documents, allowing us to retrieve data quickly with few read operations. With the help of an example, we will now discuss the one-to-many relationship with embedded documents.

# Schema design and data modeling

## 2. One-to-Many

// Student document

```
{  StudentName: Ishan,  
  StudentId: 123_2022,  
  Branch:CSE  
}
```

// Permanent Address document

```
{  StudentName: Ishan,  
  PermanentAddress: XXXXXXXX,  
  City: Jaipur,  
  PinCode:302022 }
```

// Current Address document

```
{  
  StudentName: Ishan,  
  CurrentAddress: XXXXXXXX,  
  City: Kota,  
  PinCode:324001 }
```





# Schema design and data modeling

Instead of writing three documents, we can now combine them into one.

// Student document

```
{  StudentName: ishan,
  StudentId: k_hut_2022,
  Branch:CSE
  Address: [
    {
      StudentName: Ishan,
      PermanentAddress: XXXXXXXX,
      City: Jaipur,
      PinCode:302022
    }, {
      StudentName: Ishan,
      CurrentAddress: XXXXXXXX,
      City: kota,
      PinCode:324001    }  ] }
```



# Schema design and data modeling

## **One-to-Many relationships with the document reference**

The document reference model can also be used to create a one-to-many relationship. We keep the documents separate in this model, but one document contains the references to the others.

## **A Many-to-Many relationship**

Many to Many relationships are a type of mongodb relationship in which any two entities within a document can have multiple relationships.

In this relationship, we can consider a case of Online courses website where there are many courses and also many users. Each course is purchased by many users and each user may purchase many courses.



# MongoDB Query Operators

## Comparison

The following operators can be used in queries to compare values:

\$eq	Matches values that are equal to the given value.
\$gt	Matches if values are greater than the given value.
\$lt	Matches if values are less than the given value.
\$gte	Matches if values are greater or equal to the given value.
\$lte	Matches if values are less or equal to the given value.
\$in	Matches any of the values in an array.
\$ne	Matches values that are not equal to the given value.
\$nin	Matches none of the values specified in an array



# MongoDB Query Operators

## Comparison

The following operators can be used in queries to compare values:

\$eq	Matches values that are equal to the given value.
\$gt	Matches if values are greater than the given value.
\$lt	Matches if values are less than the given value.
\$gte	Matches if values are greater or equal to the given value.
\$lte	Matches if values are less or equal to the given value.
\$in	Matches any of the values in an array.
\$ne	Matches values that are not equal to the given value.
\$nin	Matches none of the values specified in an array



# MongoDB Query Operators

## Logical

The following operators can logically compare multiple queries.

\$and	Joins two or more queries with a logical AND and returns the documents that match all the conditions.
\$or	Join two or more queries with a logical OR and return the documents that match either query.
\$nor	The opposite of the OR operator. The logical NOR operator will join two or more queries and return documents that do not match the given query conditions.
\$not	Returns the documents that do not match the given query expression.



# MongoDB Aggrigation

In MongoDB, aggregation operations process the data records/documents and return computed results. It collects values from various documents and groups them together and then performs different types of operations on that grouped data like sum, average, minimum, maximum, etc to return a computed result.

MongoDB provides three ways to perform aggregation

Aggregation pipeline

Map-reduce function

Single-purpose aggregation



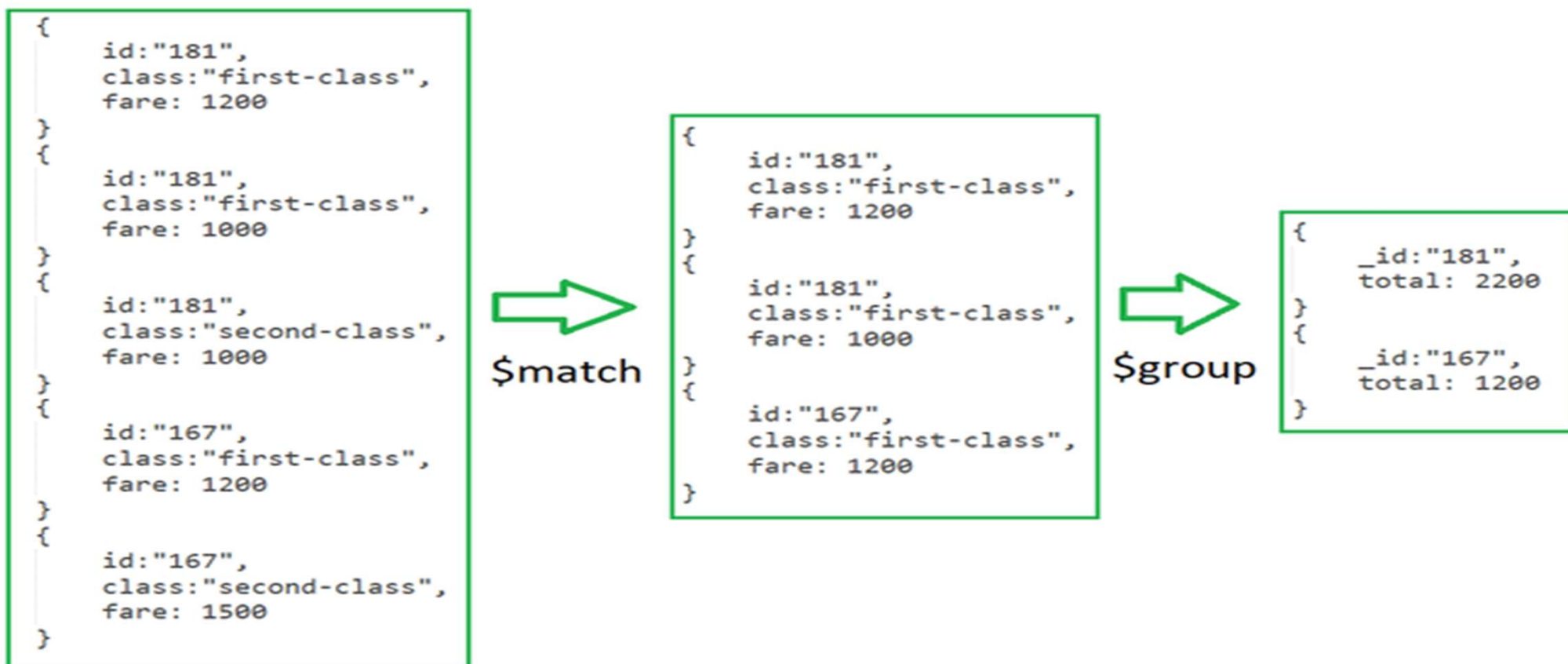
# Aggrigation Pipeline

In MongoDB, the aggregation pipeline consists of stages and each stage transforms the document. Or in other words, the aggregation pipeline is a multi-stage pipeline, so in each state, the documents taken as input and produce the resultant set of documents now in the next stage the resultant documents taken as input and produce output, this process is going on till the last stage. The basic pipeline stages provide filters that will perform like queries and the document transformation modifies the resultant document and the other pipeline provides tools for grouping and sorting documents.



# Aggrigation Pipeline

```
db.train.aggregate( [
  {$match:{class:"first-class"}},
  {$group:{_id:"id",total:{$sum:"$fare"}}} ] pipeline stages
)
```





## Aggrigation Pipeline

In the previous example of a collection of train fares in the first stage. Here, the \$match stage filters the documents by the value in class field i.e. class: “first-class” 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.



# Aggrigation Pipeline

**Stages:** Each stage starts from stage operators which are:

1. `$match`: It is used for filtering the documents can reduce the amount of documents that are given as input to the next stage.
2. `$project`: It is used to select some specific fields from a collection.
3. `$group`: It is used to group documents based on some value.
4. `$sort`: It is used to sort the document that is rearranging them



## Aggrigation Pipeline

5. \$skip: It is used to skip n number of documents and passes the remaining documents
6. \$limit: It is used to pass first n number of documents thus limiting them.
7. \$unwind: It is used to unwind documents that are using arrays i.e. it deconstructs an array field in the documents to return documents for each element.
8. \$out: It is used to write resulting documents to a new collection.



# Aggrigation Pipeline

**Expressions:** It refers to the name of the field in input documents for e.g. { \$group : { \_id : “\$id“, total: { \$sum: ”\$fare“ } } } here \$id and \$fare are expressions.

Accumulators: These are basically used in the group stage

1. sum: It sums numeric values for the documents in each group
2. count: It counts total numbers of documents



## Aggrigation Pipeline

3. avg: It calculates the average of all given values from all documents
4. min: It gets the minimum value from all the documents
5. max: It gets the maximum value from all the documents
6. first: It gets the first document from the grouping
7. last: It gets the last document from the grouping



# Aggrigation Pipeline

```
db.dummy.aggregate([  
  // Stage 1: Only find documents that have more than 1 like  
  {  
    $match: { likes: { $gt: 1 } }  
  },  
  // Stage 2: Group documents by category and sum each categories likes  
  {  
    $group: { _id: "$category", totalLikes: { $sum: "$likes" } }  
  }  
])
```

# Aggrigation Pipeline

```
db.restaurants.aggregate([
  {
    $project: {
      "name": 1,
      "cuisine": 1,
      "address": 1
    }
  },
  {
    $limit: 5
  }
])
```





# Aggrigation Pipeline

```
db.listingsAndReviews.aggregate([
  {
    $sort: { "accommodates": -1 }
  },
  {
    $project: {
      "name": 1,
      "accommodates": 1
    }
  },
  {
    $limit: 5
  }
])
```



# Aggrigation Pipeline

```
db.listingsAndReviews.aggregate([
  { $match : { property_type : "House" } },
  { $limit: 2 },
  { $project: {
    "name": 1,
    "bedrooms": 1,
    "price": 1
  }}
])
```



# Aggrigation Pipeline

**\$addField:** This will return the documents along with a new field, avgGrade, which will contain the average of each restaurants grades.score.

```
db.restaurants.aggregate([
  {
    $addFields: {
      avgGrade: { $avg: "$grades.score" }
    }
  },
  {
    $project: {
      "name": 1,
      "avgGrade": 1
    }
  }
])
```



## Aggrigation Pipeline

**\$lookup:** This aggregation stage performs a left outer join to a collection in the same database.

**from:** The collection to use for lookup in the same database

**localField:** The field in the primary collection that can be used as a unique identifier in the from collection.

**foreignField:** The field in the from collection that can be used as a unique identifier in the primary collection.

**as:** The name of the new field that will contain the matching documents from the from collection.

# Aggrigation Pipeline

```
db.comments.aggregate([
  {
    $lookup: {
      from: "movies",
      localField: "movie_id",
      foreignField: "_id",
      as: "movie_details",
    },
  },
  {
    $limit: 1
  }
])
```



# Aggrigation Pipeline

## 1. **findOneAndUpdate():**

Finds a single document that matches a specified query criteria, updates it, and returns the original or modified document.

example:

```
const result = db.students.findOneAndUpdate(  
  { name: "Alice" }, // Query criteria  
  { $set: { grade: "A+" } }, // Update operation  
  { returnDocument: "after" } // Options to return the modified document  
);  
  
printjson(result.value);
```



# Aggrigation Pipeline

## 1. **findOneAndUpdate():**

Finds a single document that matches a specified query criteria, updates it, and returns the original or modified document.

example:

```
const result = db.students.findOneAndUpdate(  
  { name: "Alice" }, // Query criteria  
  { $set: { grade: "A+" } }, // Update operation  
  { returnDocument: "after" } // Options to return the modified document  
);  
  
printjson(result.value);
```



# Aggrigation Pipeline

## 2. findOneAndDelete():

Finds a single document that matches a specified query criteria, deletes it, and returns the original or deleted document.

example

```
const result = db.students.findOneAndDelete(  
  { name: "Bob" } // Query criteria  
);
```





# Aggrigation Pipeline

## 3. findOneAndReplace():

Finds a single document that matches a specified query criteria, replaces it with a new document, and returns the original or replaced document.

```
const result = db.students.findOneAndReplace(  
  { name: "Charlie" }, // Query criteria  
  { name: "Charlie", age: 24, grade: "A-" }, // Replacement document  
  { returnDocument: "before" } // Options to return the original document  
);
```

# × DIGITAL LEARNING CONTENT



# Parul<sup>®</sup> University



[www.paruluniversity.ac.in](http://www.paruluniversity.ac.in)