ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

(DATA SCIENCE)

(ACADEMIC YEAR 2024-25)

LABORATORY MANUAL

COURSE: MONGODB LABORATORY

Course CODE: BDSL456B

SEMESTER: IV-2022 CBCS Scheme

Prepared By **Approved By**

Dr.Neethi M V
Faculty In-charge

Dr. Anitha D B
HOD, CSE-DSE

INSTITUTIONAL MISSION AND VISION

Objectives

- To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To cultivate strong community relationships and involve the students and the staff in local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING AND ENGINEERING

(DATA SCIENCE &ENGINEERING)

Vision of The Department

- To impart technical education in the field of data science of excellent quality with a high level of professional competence, social responsibility, and global awareness among the students

Mission

- To impart technical education that is up to date, relevant and makes students competitive and employable at global level
- To provide technical education with a high sense of discipline, social relevance in an intellectually, ethically and socially challenging environment for better tomorrow
- Educate to the global standards with a benchmark of excellence and to kindle the spirit of innovation.

Program Outcomes(PO)

- **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

- **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

- **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

- **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

- **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice
- **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

- **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

- **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

- **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

- **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

- PSO1: Develop relevant programming skills to become a successful data scientist
- PSO2: Apply data science concepts and algorithms to solve real world problems of the society
- PSO3: Apply data science techniques in the various domains like agriculture, education healthcare for better society

Program Educational Objectives (PEOs):

PEO1: Develop cutting-edge skills in data science and its related technologies, such as machine learning, predictive analytic, and data engineering.

PEO2: Design and develop data-driven solutions to real-world problems in a business, research, or social environment.

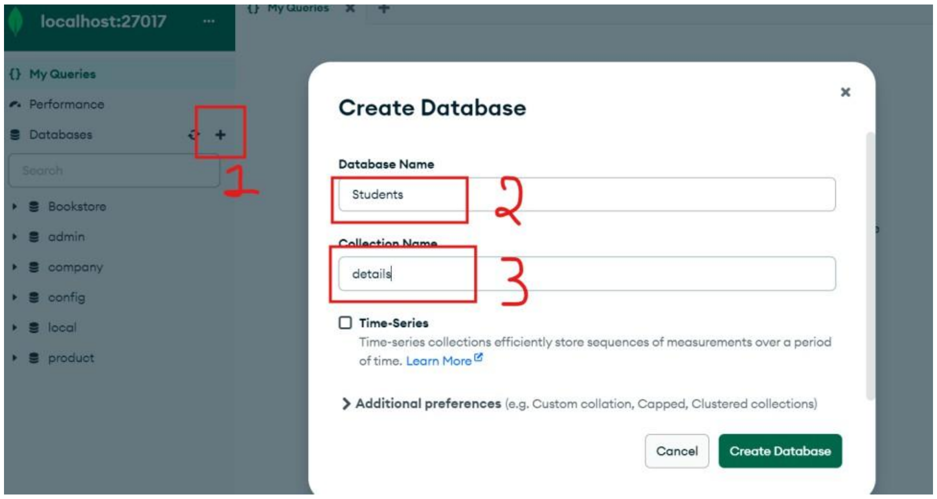
PEO3: Apply data engineering and data visualization techniques to discover, investigate, and interpret data.

PEO4: Demonstrate ethical and responsible data practices in problem solving

PEO5: Integrate fields within computer science, optimization, and statistics to develop better solutions

Program 1

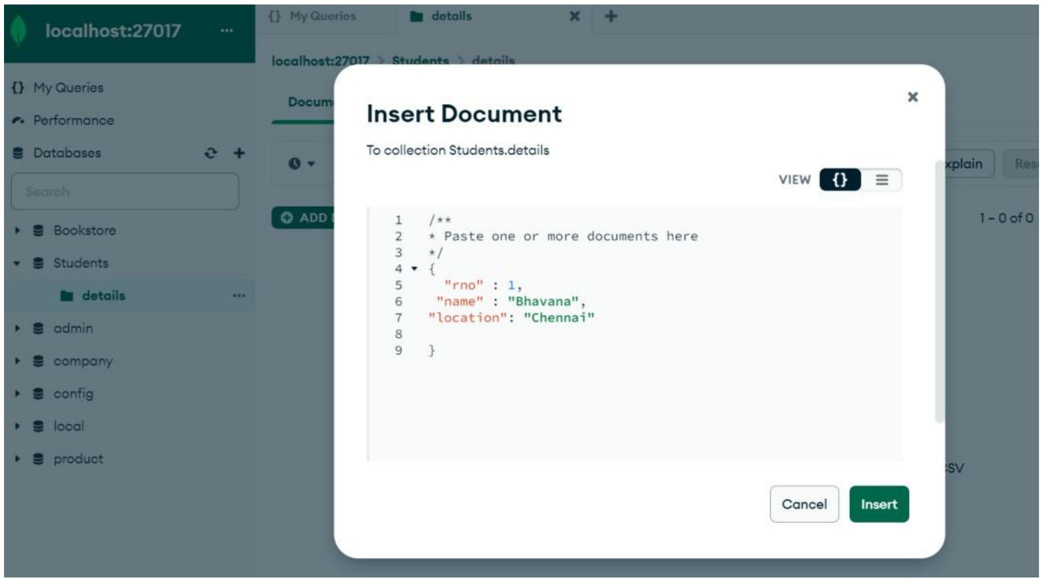
a. Illustration of Where Clause, AND, OR operations in MongoDB.
Create a database of **Students** and collection **details** in Mongo DB IDE**.**


localhost:27017
My Queries
Performance
Databases
Search
Bookstore
admin
company
config
local
product
1
2
3
Create Database
Time-Series
Time-series collections efficiently store sequences of measurements over a period of time. Learn More
Additional preferences (e.g. Custom collation, Capped, Clustered collections)
Cancel
Create Database

Database Name: Students
Collection Name: details

Add the following documents in the **details collection** in MongoDB IDE.

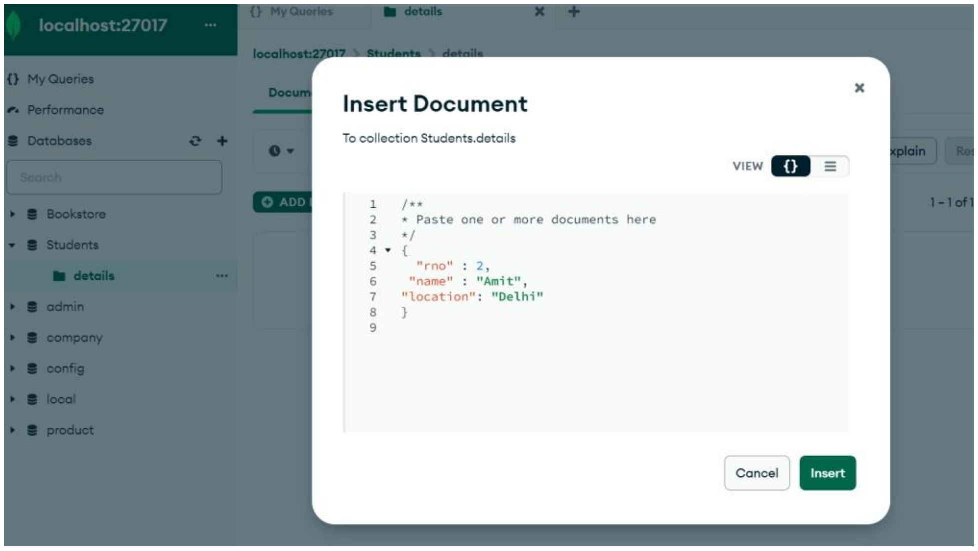
{
"rno" : 1,
"name" : "Bhavana",
"location": "Chennai"
}



```
/**
* Paste one or more documents here
*/
{
   "rno" : 1,
  "name" : "Bhavana",
 "location": "Chennai"

 }
```

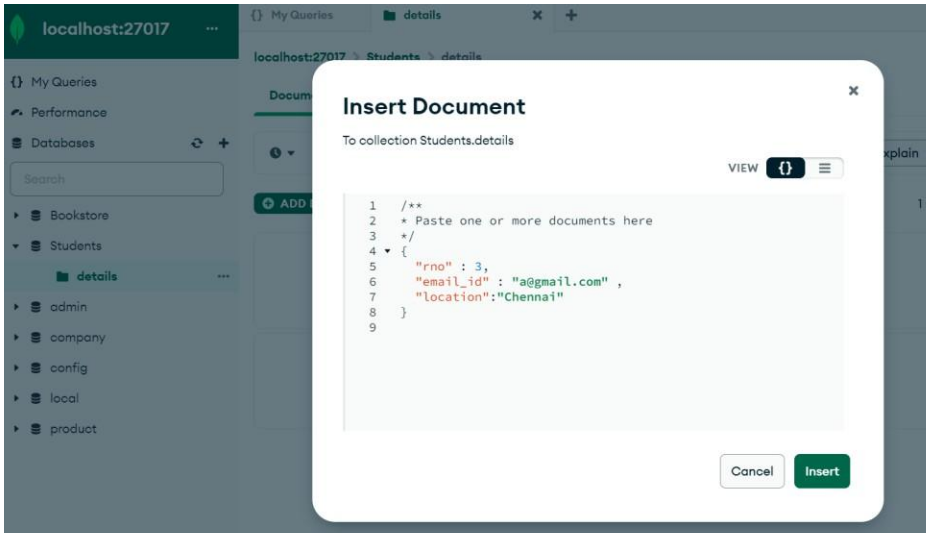
{
"rno" : 2,
"name" : "Amit",
"location": "Delhi"

}



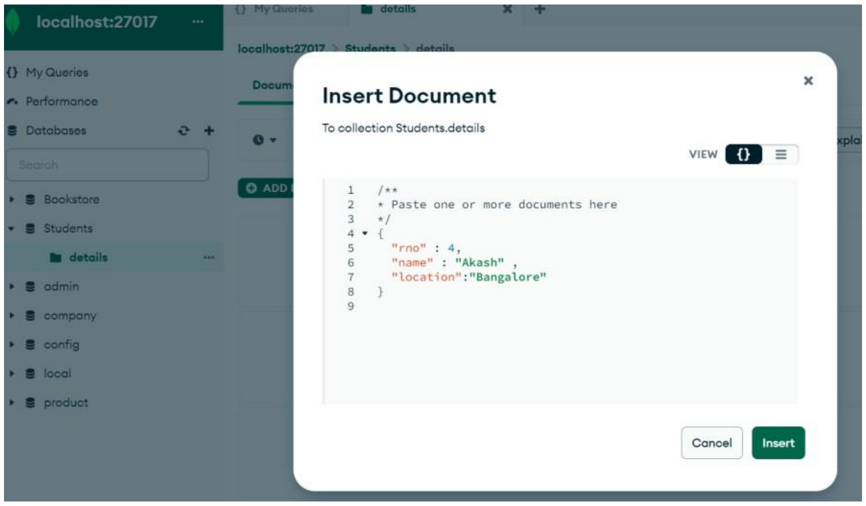
```
/**
* Paste one or more documents here
*/
{
  "rno" : 2,
 "name" : "Amit",
"location": "Delhi"
}
```

{
"rno" : 3,
"email_id" : "a@gmail.com" ,
"location":"Chennai"
}



```
/**
* Paste one or more documents here
*/
{
  "rno" : 3,
  "email_id" : "a@gmail.com" ,
  "location":"Chennai"
}
```

{
"rno" : 4,
"name" : "Akash" ,
"location":"Bangalore"
}



```
/**
* Paste one or more documents here
*/
{
  "rno" : 4,
  "name" : "Akash" ,
  "location":"Bangalore"
}
```

----

1. **Where Clause in MongoDB:** In MongoDB, the find() method is used to query documents from a collection. The find() method can accept a query document as a parameter which acts as a "WHERE" clause.

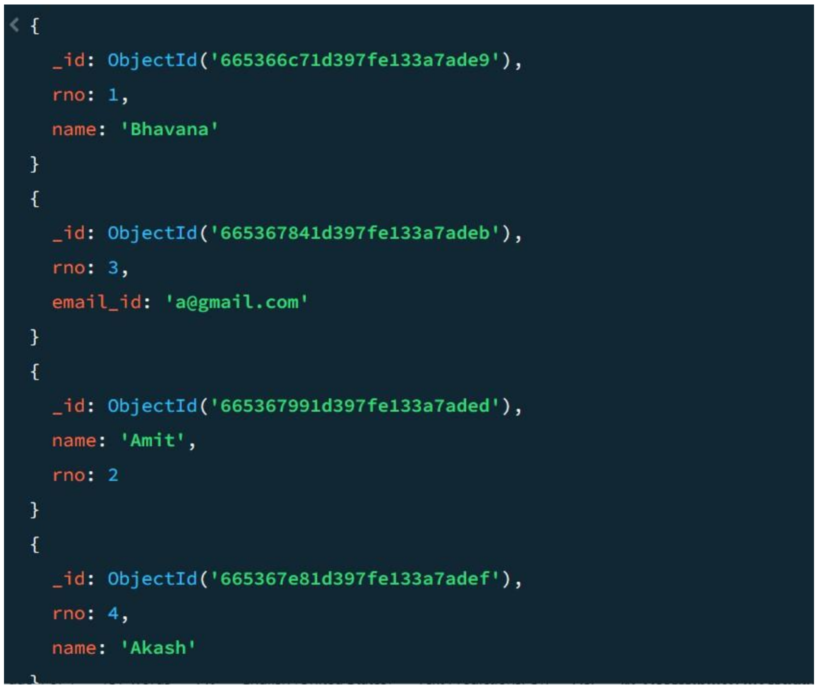
Syntax: db.collection.find({ field: value })

In Mongodb shell, execute the following code:

> use Students

> db.details.find()

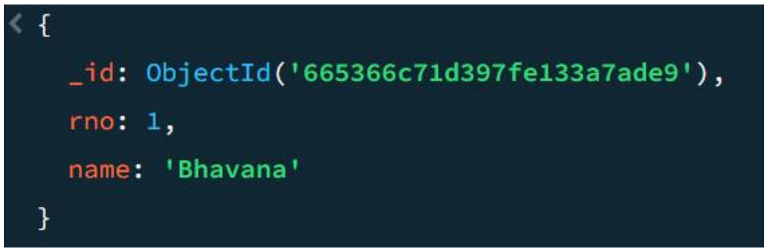
Output:



```
< {
    _id: ObjectId('665366c71d397fe133a7ade9'),
    rno: 1,
    name: 'Bhavana'
  }
  {
    _id: ObjectId('665367841d397fe133a7adeb'),
    rno: 3,
    email_id: 'a@gmail.com'
  }
  {
    _id: ObjectId('665367991d397fe133a7aded'),
    name: 'Amit',
    rno: 2
  }
  {
    _id: ObjectId('665367e81d397fe133a7adef'),
    rno: 4,
    name: 'Akash'
```

//findOne to show only first record
> db. details.findOne()

Output:



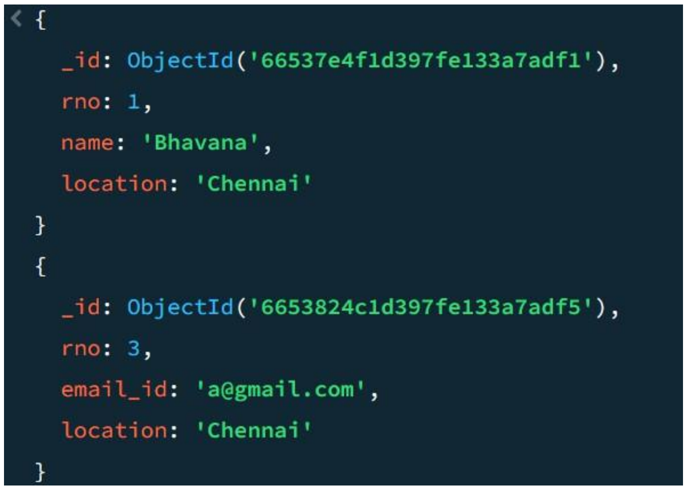
```
{
  _id: ObjectId('665366c71d397fe133a7ade9'),
  rno: 1,
  name: 'Bhavana'
}
```

----

2. **AND Operation in MongoDB:** MongoDB provides the $and operator to perform logical AND operation between multiple conditions in a query.

Syntax: db.collection.find({ $and: [{ field1: value1 }, { field2: value2 }] })

>db.details.find({$and: [{"location": "Chennai"}] })
Output:



```
{
  _id: ObjectId('66537e4f1d397fe133a7adf1'),
  rno: 1,
  name: 'Bhavana',
  location: 'Chennai'
}
{
  _id: ObjectId('6653824c1d397fe133a7adf5'),
  rno: 3,
  email_id: 'a@gmail.com',
  location: 'Chennai'
}
```

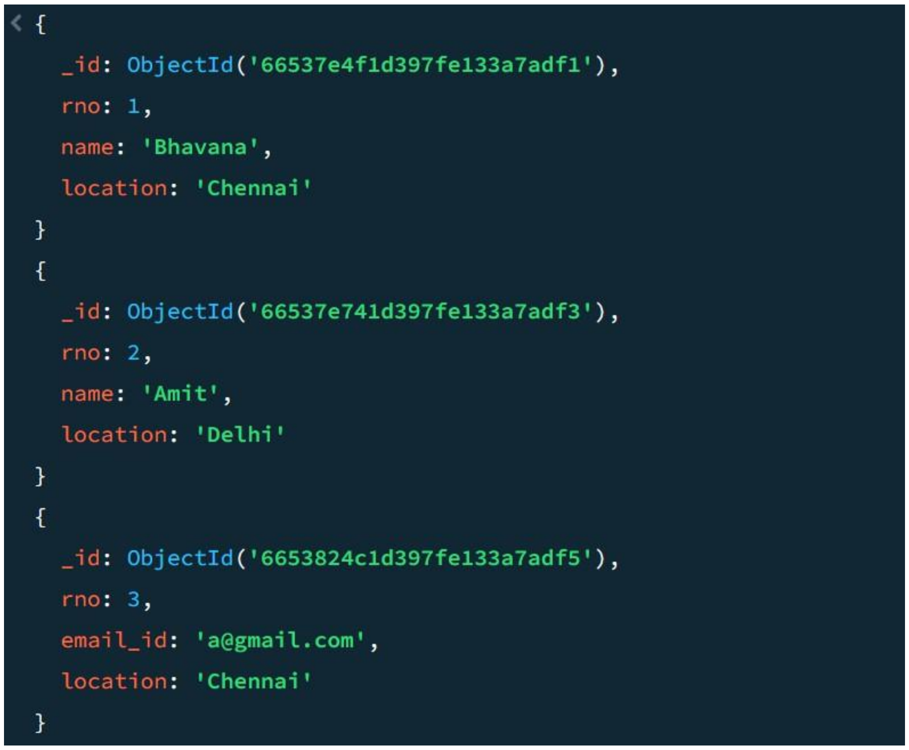
----

3. **OR Operation in MongoDB**: Similarly, MongoDB provides the $or operator to perform logical OR operation between multiple conditions in a query.

Syntax: db.collection.find({ $or: [{ field1: value1 }, { field2: value2 }] })

>db.details.find({$or: [{"location": "Chennai"}, {"location": "Delhi"}] })

Output:



```
< {
    _id: ObjectId('66537e4f1d397fe133a7adf1'),
    rno: 1,
    name: 'Bhavana',
    location: 'Chennai'
  }
  {
    _id: ObjectId('66537e741d397fe133a7adf3'),
    rno: 2,
    name: 'Amit',
    location: 'Delhi'
  }
  {
    _id: ObjectId('6653824c1d397fe133a7adf5'),
    rno: 3,
    email_id: 'a@gmail.com',
    location: 'Chennai'
  }
```

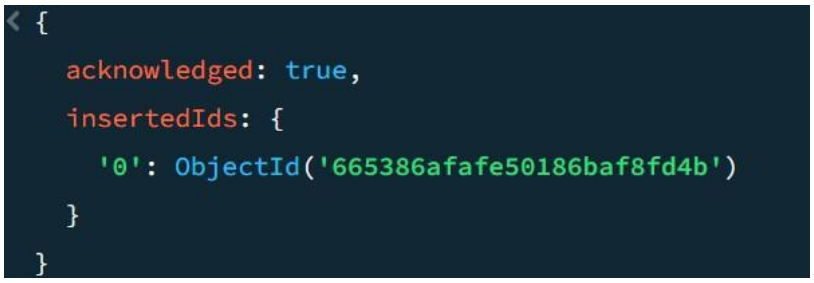
--
----

b. **Execute the Commands of MongoDB and operations in MongoDB: Insert, Query, Update, Delete and Projection. (Note: use any collection).**

1. **Insert Operation:** Use the insertOne() method to insert a single document into a collection.

Syntax: db.collection.insertOne({ field1: value1, field2: value2, field3: value3 })

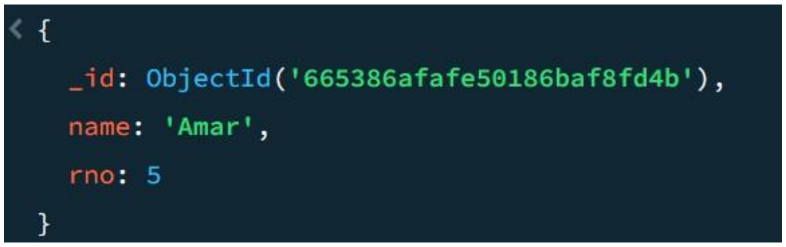
Every row/document can be different than other
> **db.details.insert({name:'Amar',rno:5})**
Output:



```
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('665386afafe50186baf8fd4b')
    }
  }
```

Verification Code:

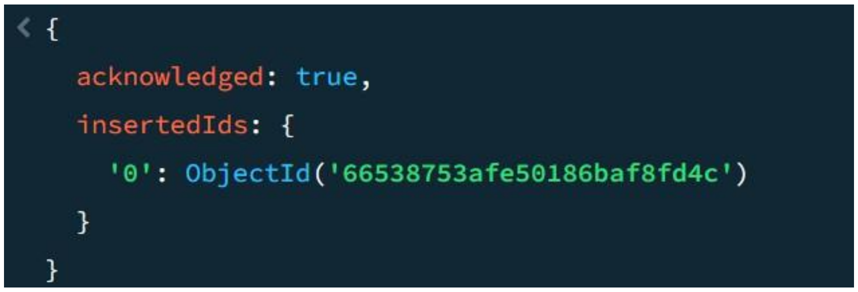
>db.details.find({name:'Amar',rno:5})



```
< {
    _id: ObjectId('665386afafe50186baf8fd4b'),
    name: 'Amar',
    rno: 5
  }
```

> **db.details.insert({rno:6, email_id:'d@gmail.com'})**

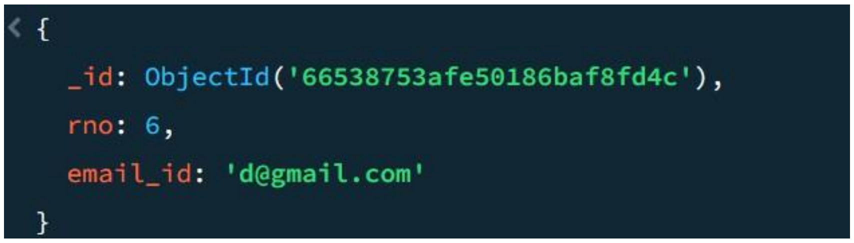
Output:



```
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('66538753afe50186baf8fd4c')
    }
  }
```

Verification Code:

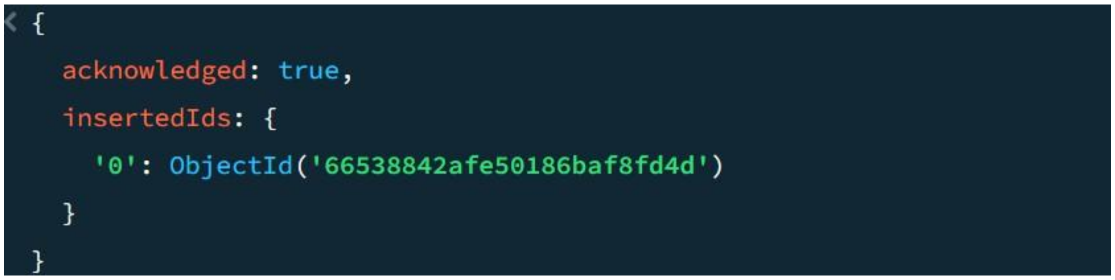
>db.details.find({rno:6, email_id:'d@gmail.com'})



```
< {
    _id: ObjectId('66538753afe50186baf8fd4c'),
    rno: 6,
    email_id: 'd@gmail.com'
  }
```

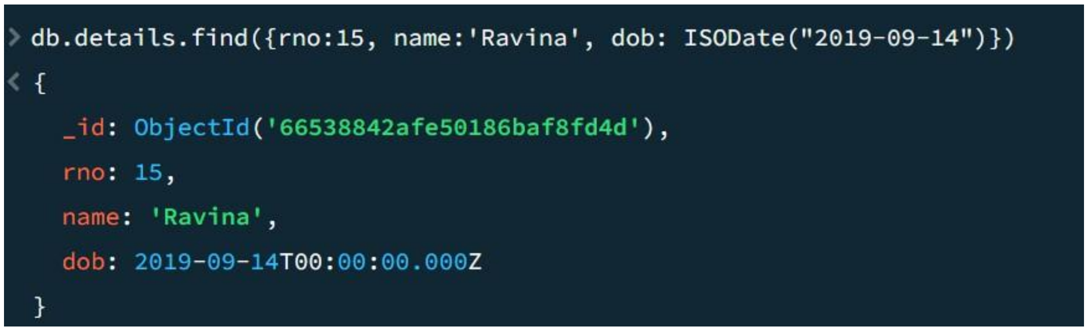
// To insert date use ISODate function

> **db.details.insert({rno:15, name:'Ravina', dob: ISODate("2019-09-14")})**



```
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('66538842afe50186baf8fd4d')
    }
  }
```

Verification Code:

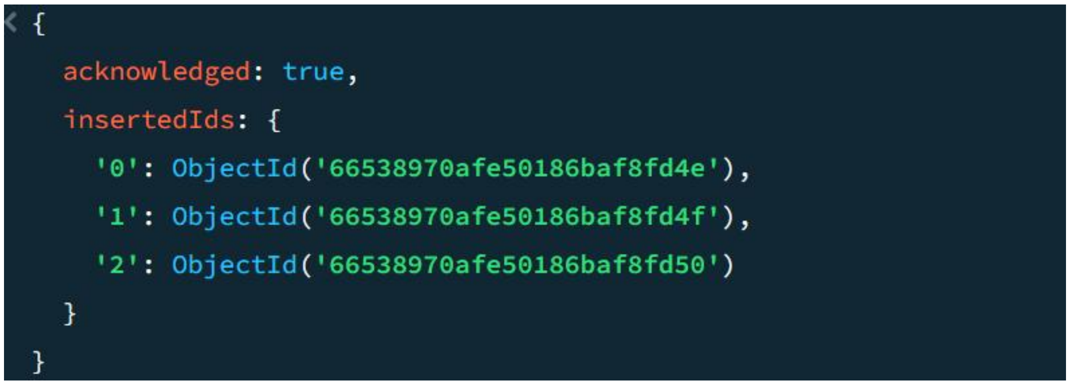


```
> db.details.find({rno:15, name:'Ravina', dob: ISODate("2019-09-14")})
< {
    _id: ObjectId('66538842afe50186baf8fd4d'),
    rno: 15,
    name: 'Ravina',
    dob: 2019-09-14T00:00:00.000Z
  }
```

//Insert multiple documents at once

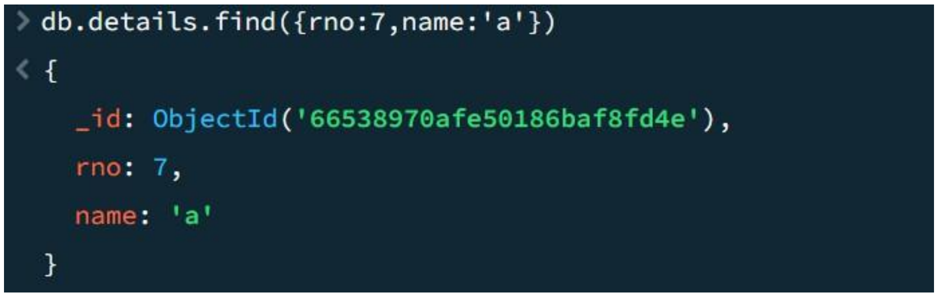
> db.details.insert([{rno:7,name:'a'},{rno:8,name:'b'},{rno:8,name:'c'}])

Output:



```
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('66538970afe50186baf8fd4e'),
      '1': ObjectId('66538970afe50186baf8fd4f'),
      '2': ObjectId('66538970afe50186baf8fd50')
    }
  }
```

Verification Code: db.details.find({rno:7,name:'a'})

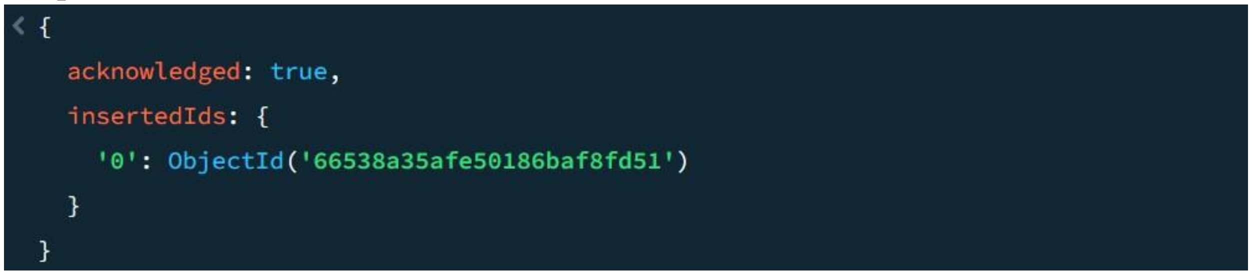


```
> db.details.find({rno:7,name:'a'})
< {
    _id: ObjectId('66538970afe50186baf8fd4e'),
    rno: 7,
    name: 'a'
  }
```

// to insert multiple values for one key using []

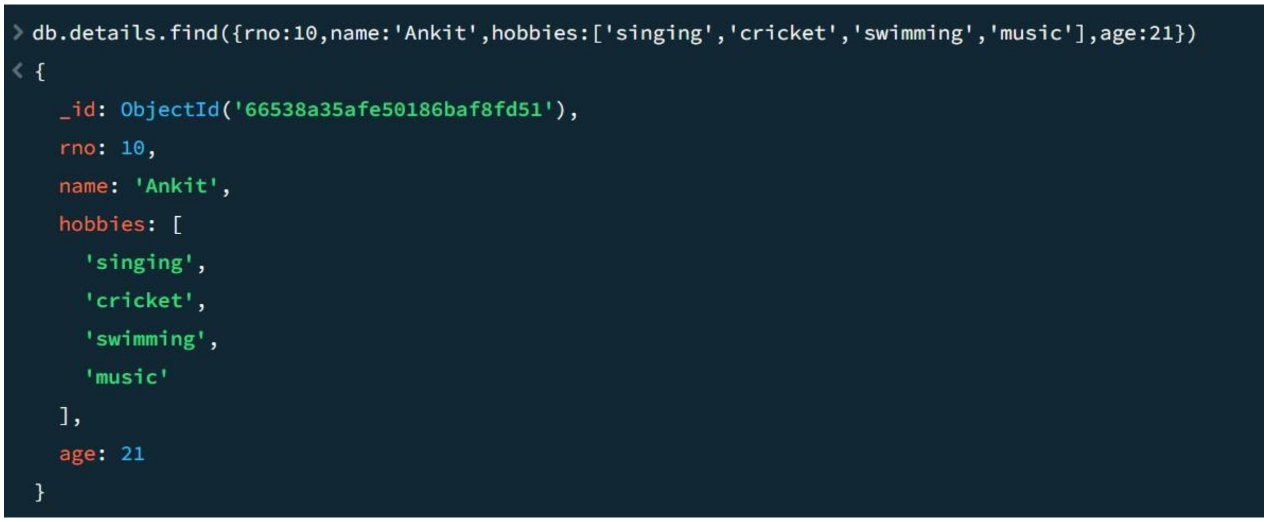
>db.details.insert({rno:10,name:'Ankit',hobbies:['singing','cricket','swimming', 'music'],age:21})

Output:



```
< {
    acknowledged: true,
    insertedIds: {
      '0': ObjectId('66538a35afe50186baf8fd51')
    }
  }
```

Verification Code:
db.details.find({rno:10,name:'Ankit',hobbies:['singing','cricket','swimming', 'music'],age:21})


```
> db.details.find({rno:10,name:'Ankit',hobbies:['singing','cricket','swimming','music'],age:21})
< {
    _id: ObjectId('66538a35afe50186baf8fd51'),
    rno: 10,
    name: 'Ankit',
    hobbies: [
      'singing',
      'cricket',
      'swimming',
      'music'
    ],
    age: 21
  }
```

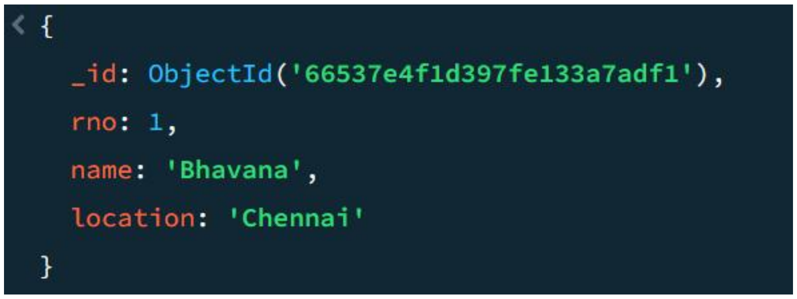
----

2. **Query Operation:** Use the find() method to query documents from a collection.

Syntax: db.collection.find({ field: value })

>db.details.find({rno:1})

Output:

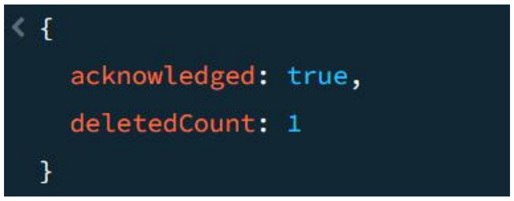

```
< {
    _id: ObjectId('66537e4f1d397fe133a7adf1'),
    rno: 1,
    name: 'Bhavana',
    location: 'Chennai'
  }
```

3. **Delete Operation:** Use the deleteOne() method to delete a single document from a collection.

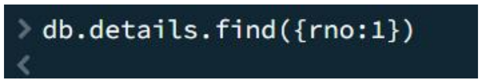
Syntax: db.collection.deleteOne({ field: value })

>db.details.deleteOne({rno:1})

Output:


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

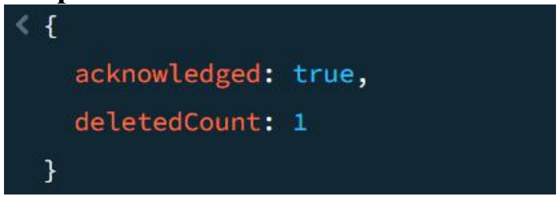
Verification Code:



```
> db.details.find({rno:1})
<
```

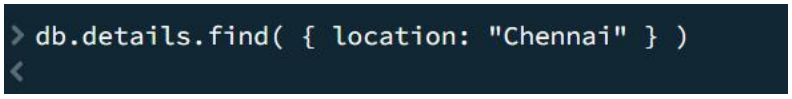
>db. details.deleteMany({ location: "Chennai" })

Output:



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

Verification Code:

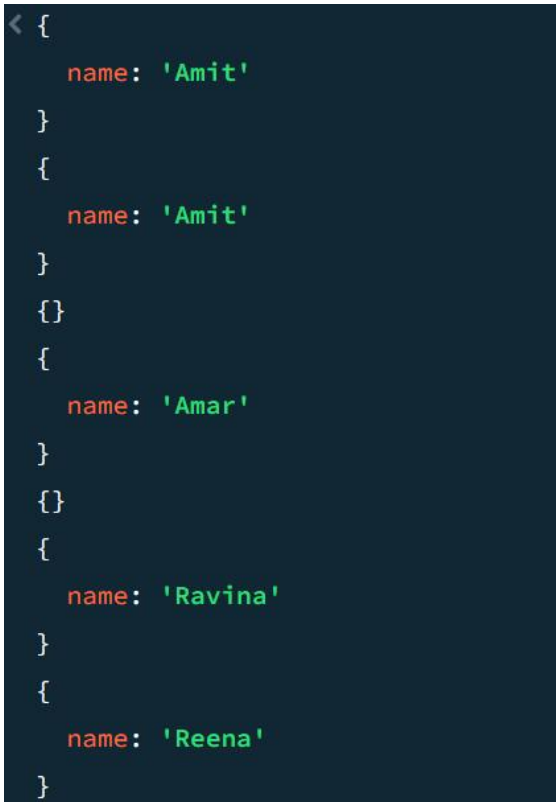


```
> db.details.find( { location: "Chennai" } )
<
```

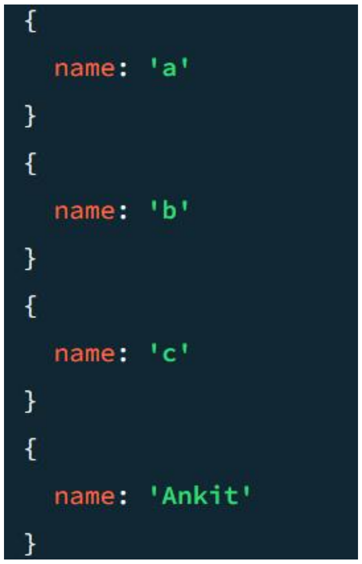
4. **Projection Operation:** Use the second parameter of the find() method to specify which fields to include or exclude in the query result.

Syntax: db.collection.find({}, { field1: 1, field2: 1, _id: 0 })

// Find command to show only names without condition
> db. details.find({},{name:1,_id:0})



```
< {
    name: 'Amit'
  }
  {
    name: 'Amit'
  }
  {}
  {
    name: 'Amar'
  }
  {}
  {
    name: 'Ravina'
  }
  {
    name: 'Reena'
  }
```



```
{
  name: 'a'
}
{
  name: 'b'
}
{
  name: 'c'
}
{
  name: 'Ankit'
}
```

Program 2

a. **Develop a MongoDB query to select certain fields and ignore some fields of the documents from any collection.**

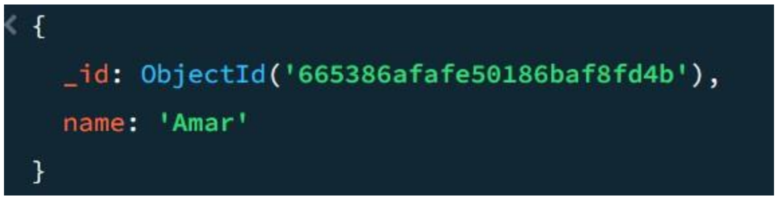
Syntax: db.collection.find({}, { field1: 1, field2: 1, _id: 0 })

- db.collection.find({}) is used to retrieve all documents from the collection.
- { field1: 1, field2: 1, _id: 0 } specifies the projection document where:
- field1: 1 and field2: 1 indicates that these fields will be included in the result.
- _id: 0 indicates that the _id field will be excluded from the result.

//Find command with condition with giving name field only to show

```
> db. details.find({rno:5},{name:1})
```

Output:

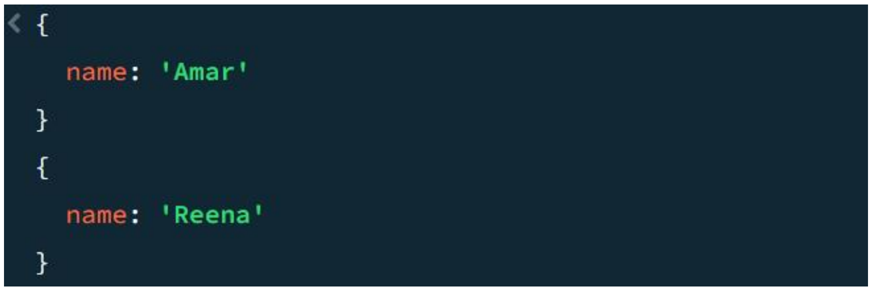


```
< {
    _id: ObjectId('665386afafe50186baf8fd4b'),
    name: 'Amar'
  }
```

//Find command with condition with giving name field only to show and _id to hide

```
>db. details.find({rno:5},{name:1,_id:0})
```

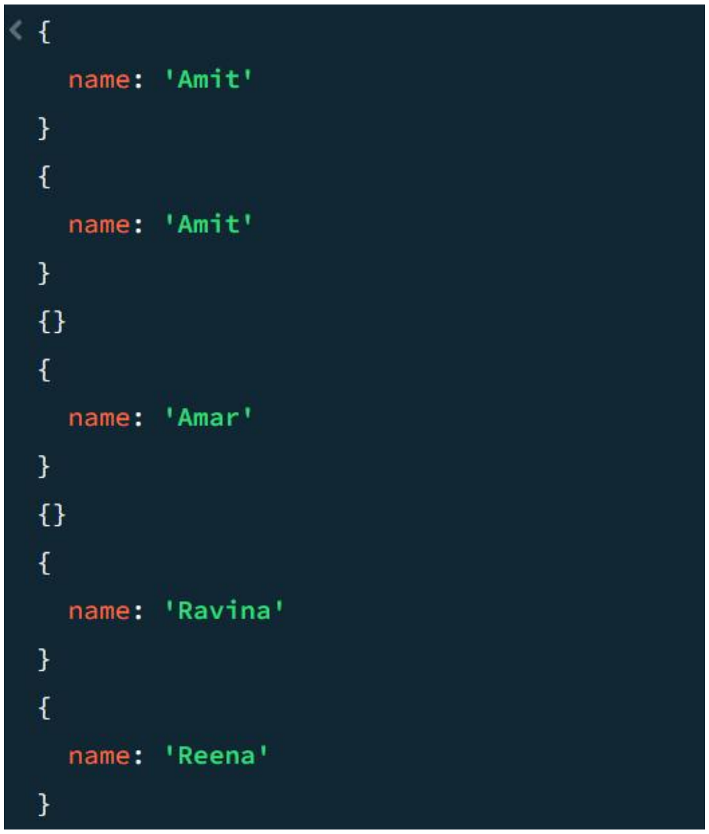
Output:



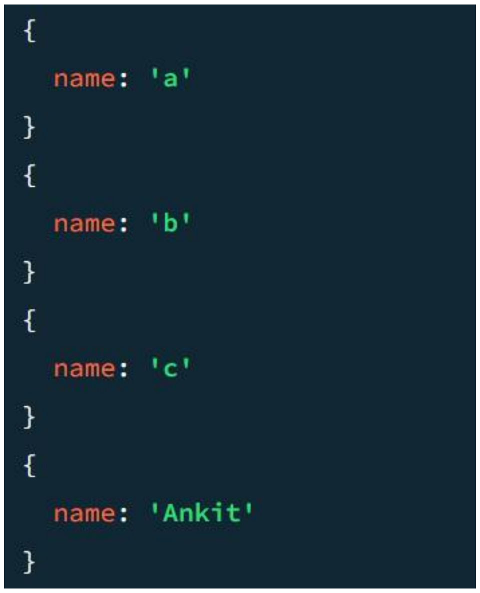
```
< {
    name: 'Amar'
  }
  {
    name: 'Reena'
  }
```

// Find command to show only names without condition

> db. details.find({},{name:1,_id:0})



```
{
  name: 'Amit'
}
{
  name: 'Amit'
}
{}
{
  name: 'Amar'
}
{}
{
  name: 'Ravina'
}
{
  name: 'Reena'
}
```



```
{
  name: 'a'
}
{
  name: 'b'
}
{
  name: 'c'
}
{
  name: 'Ankit'
}
```

b. Develop a MongoDB query to display the first 5 documents from the results obtained in a. [use of limit and find]

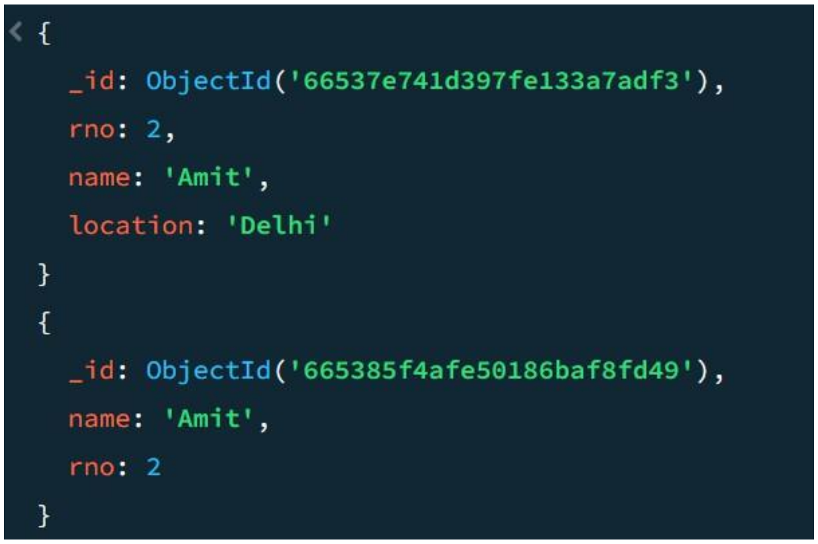
Limit Operation: Used to restrict the number of documents returned by a query. This is particularly useful when you're dealing with large datasets and you only need a subset of documents.

Syntax: db.collection.find({}, { field1: 1, field2: 1, _id: 0 }).limit(5)

- Limit (5) limits the number of documents returned to 5.
- **// Limit use to show only some records from starting- following command shows only first 2 records from collection**

  > db. details.find().limit(2)

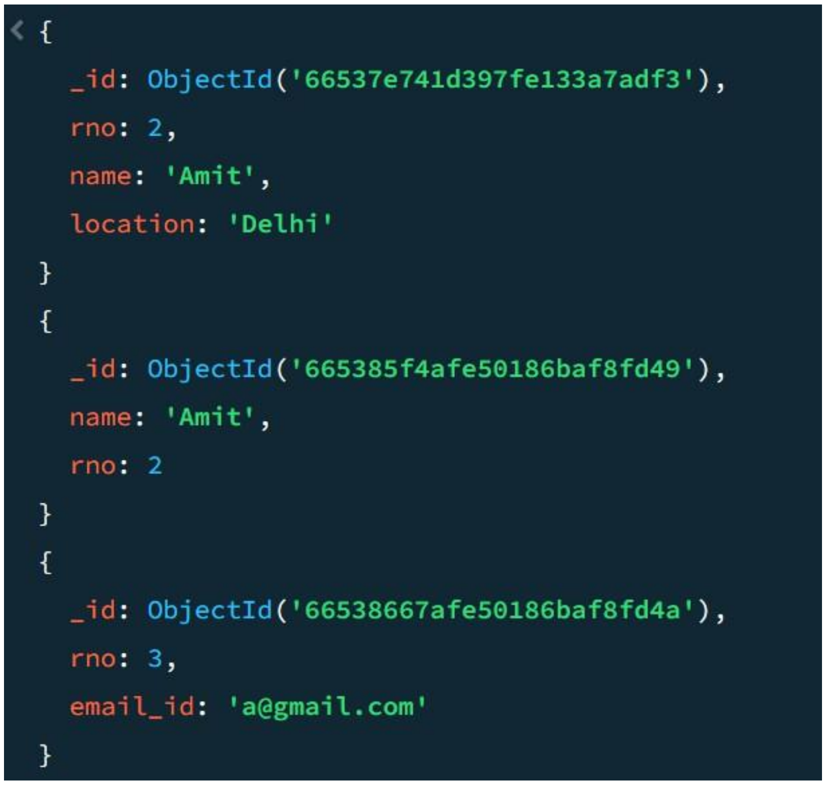
  Output:



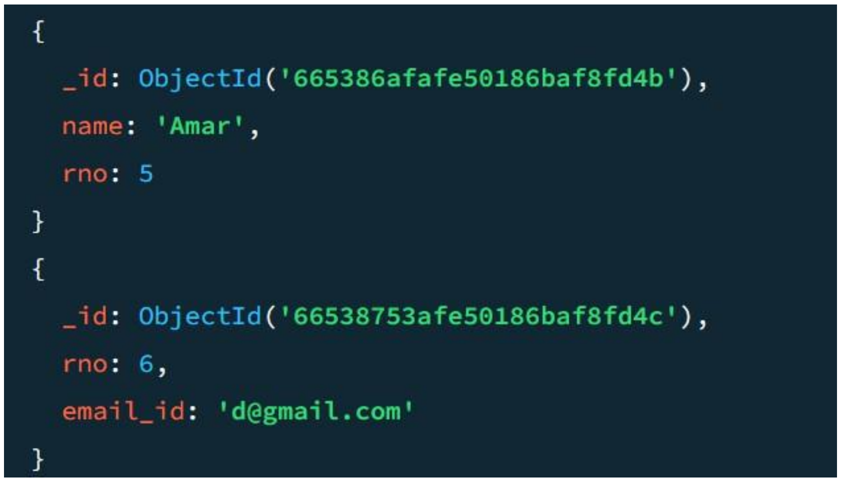
```
< {
    _id: ObjectId('66537e741d397fe133a7adf3'),
    rno: 2,
    name: 'Amit',
    location: 'Delhi'
  }
  {
    _id: ObjectId('665385f4afe50186baf8fd49'),
    name: 'Amit',
    rno: 2
  }
```

> db. details.find().limit(5)

Output:



```
< {
    _id: ObjectId('66537e741d397fe133a7adf3'),
    rno: 2,
    name: 'Amit',
    location: 'Delhi'
  }
  {
    _id: ObjectId('665385f4afe50186baf8fd49'),
    name: 'Amit',
    rno: 2
  }
  {
    _id: ObjectId('66538667afe50186baf8fd4a'),
    rno: 3,
    email_id: 'a@gmail.com'
  }
```



```
{
  _id: ObjectId('665386afafe50186baf8fd4b'),
  name: 'Amar',
  rno: 5
}
{
  _id: ObjectId('66538753afe50186baf8fd4c'),
  rno: 6,
  email_id: 'd@gmail.com'
}
```

Program 3

a. Execute query selectors (comparison selectors, logical selectors) and list out the results on any collection

b. Execute query selectors (Geospatial selectors, Bitwise selectors) and list out the results on any collection

Comparison Selectors: Comparison selectors are used to compare fields against specific values or other fields. Here are some common comparison selectors:

$eq - Matches values that are equal to a specified value.

$ne - Matches all values that are not equal to a specified value.

$gt - Matches values that are greater than a specified value.

$gte - Matches values that are greater than or equal to a specified value.

$lt - Matches values that are less than a specified value.

$lte - Matches values that are less than or equal to a specified value.

$in - Matches any of the values specified in an array.

$nin - Matches none of the values specified in an array.

Logical Selectors: Logical selectors are used to combine multiple conditions in a query. Here are some common logical selectors:

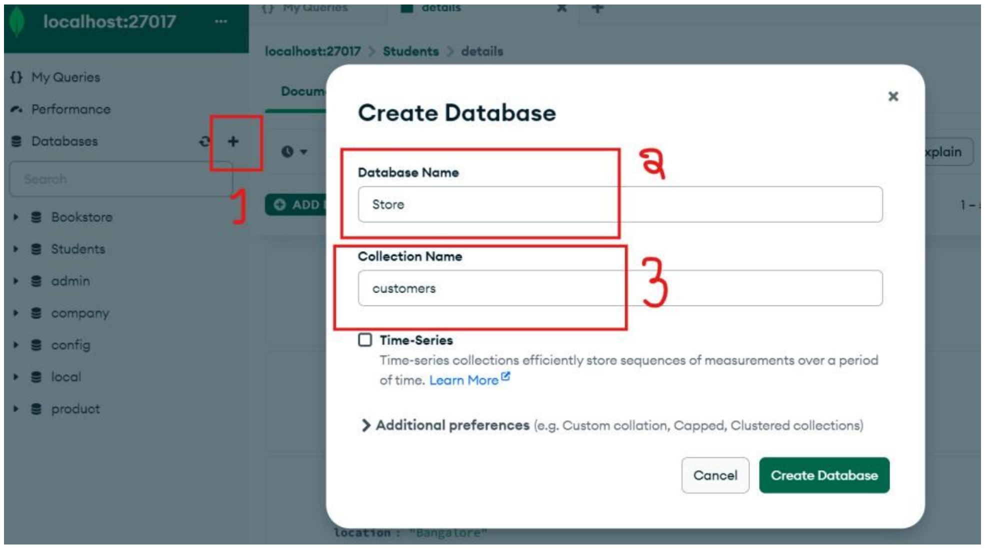
$and - Joins query clauses with a logical AND and requires that all conditions be true.

$or - Joins query clauses with a logical OR and requires that at least one condition be true.

$not - Inverts the effect of a query expression and returns documents that do not match the query expression.

$nor - Joins query clauses with a logical NOR and requires that none of the conditions be true.

Create a database **Store** and collection **customers** in Mongo DB IDE**.**



In MongoDB Shell:

```
>use Store
> db.customers.insertMany([ { _id: 1, name: "Alice", age: 30, city: "New York" },
                            { _id: 2, name: "Bob", age: 25, city: "San Francisco" },
                           { _id: 3, name: "Charlie", age: 35, city: "Los Angeles" },
                            { _id: 4, name: "David", age: 28, city: "Chicago" },
                             {_id: 5, name: "Eve", age: 32, city: "Miami" } ])
```

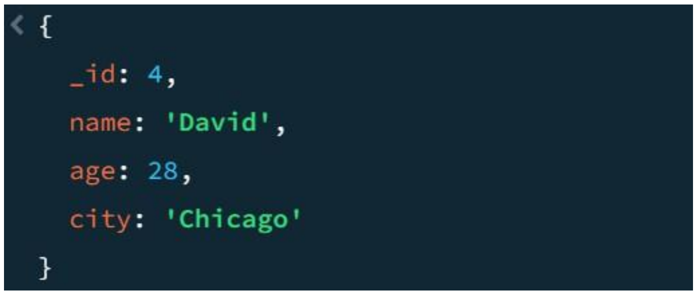
a. **Execute query selectors (comparison selectors, logical selectors) and list out the results on any collection.**

<u>Using Comparison Selectors</u>

1. **Find customers aged 28:**

```
>db.customers.find({ "age": { "$eq": 28 } })
```

Output:

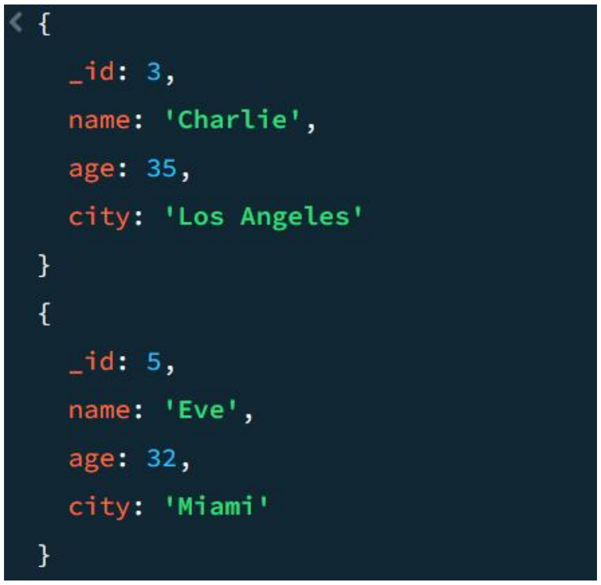


```
< {
    _id: 4,
    name: 'David',
    age: 28,
    city: 'Chicago'
  }
```

2. **Find customers older than 30:**

   >db.customers.find({ "age": { "$gt": 30 } })

   Output:



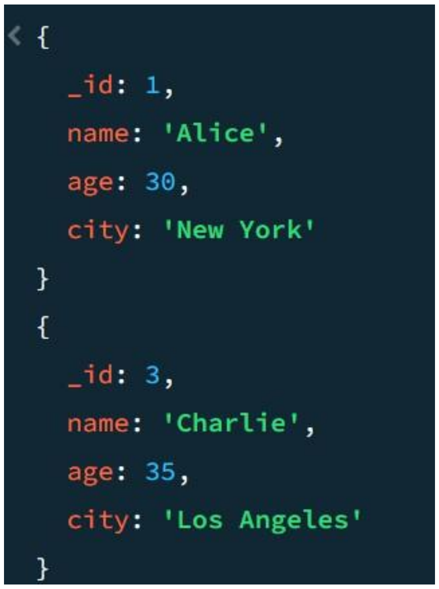
```
< {
    _id: 3,
    name: 'Charlie',
    age: 35,
    city: 'Los Angeles'
  }
  {
    _id: 5,
    name: 'Eve',
    age: 32,
    city: 'Miami'
  }
```

<u>Using Logical Selectors</u>

3. **Find customers in city is New York OR city is Los Angeles:**

```
>db.customers.find({
   $or: [
      { city: "New York" },
      { city: " Los Angeles"  }
   ] })
```

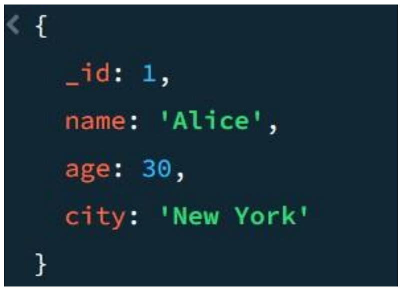
Output:


```
< {
    _id: 1,
    name: 'Alice',
    age: 30,
    city: 'New York'
  }
  {
    _id: 3,
    name: 'Charlie',
    age: 35,
    city: 'Los Angeles'
  }
```

4. Find customers age 30 and city New York

```
>db.customers.find({
   $and: [
      { age: 30 },
      { city:"New York"  }
   ] })
```

Output:

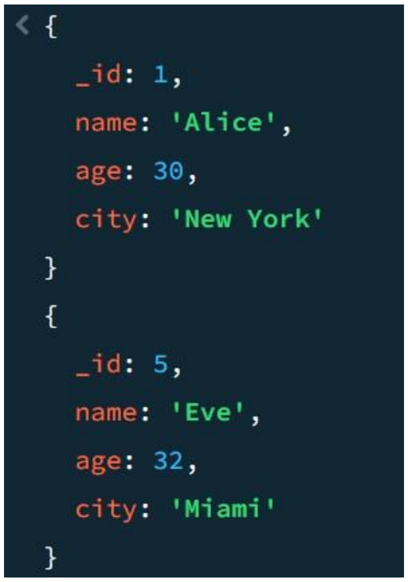

```
< {
    _id: 1,
    name: 'Alice',
    age: 30,
    city: 'New York'
  }
```

Using Both Comparison and Logical Selectors

5. **Find customers greater than or equal to 18, less than 35, in city New York or Miami**

```
>db.customers.find({$and: [
      { age: { $gte: 18 } },        // age greater than or equal to 18
       { age: { $lt: 35 } },         // age less than 35
       { city: { $in: ["New York", "Miami"] } }   // city is either "New York" or
"Miami"
  ]})
```

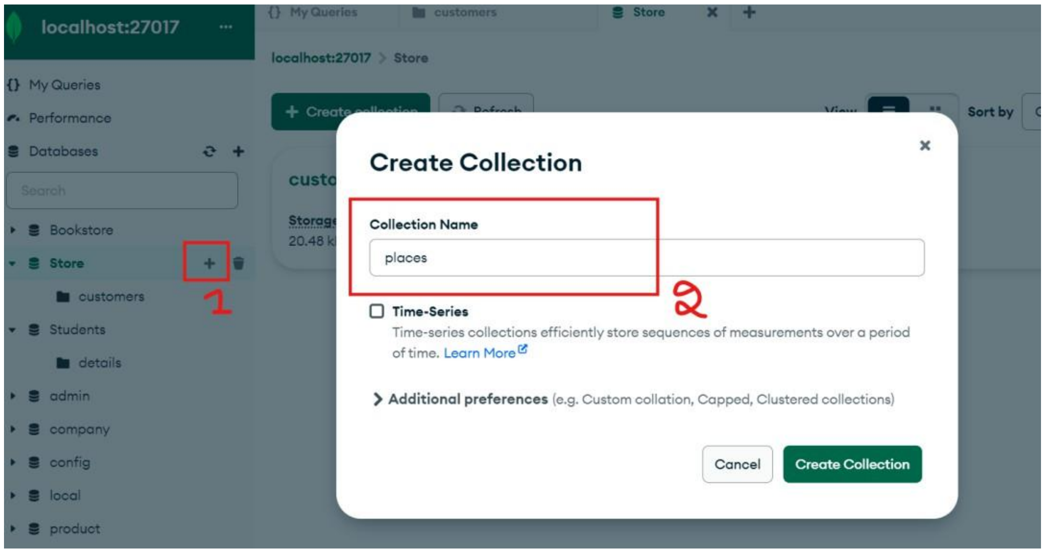
Output:



```
< {
    _id: 1,
    name: 'Alice',
    age: 30,
    city: 'New York'
  }
  {
    _id: 5,
    name: 'Eve',
    age: 32,
    city: 'Miami'
  }
```

b. **Execute query selectors (Geospatial selectors, Bitwise selectors) and list out the results on any collection**

Under database **Store**, create a collection **places** in Mongo DB IDE**.**



Create Collection

Collection Name: places

Geospatial Selectors: MongoDB supports geospatial queries for geospatial data. It provides two types of geospatial indexes: 2d indexes and 2d sphere indexes.

Add the following documents in the **places collection** in MongoDB Shell.

```
>db.places.insertMany([
{ _id: 1, name: "Place A", location: { type: "Point", coordinates: [ -73.97, 40.77 ] } }, // New
York
 { _id: 2, name: "Place B", location: { type: "Point", coordinates: [ -122.43, 37.77 ] } }, //
San Francisco
   { _id: 3, name: "Place C", location: { type: "Point", coordinates: [ -118.25, 34.05 ] } }, //
Los Angeles
   { _id: 4, name: "Place D", location: { type: "Point", coordinates: [ -87.63, 41.88 ] } }, //
Chicago
   { _id: 5, name: "Place E", location: { type: "Point", coordinates: [ -80.19, 25.77 ] } }  //
Miami
])
```

Create a 2dsphere Index on location:

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

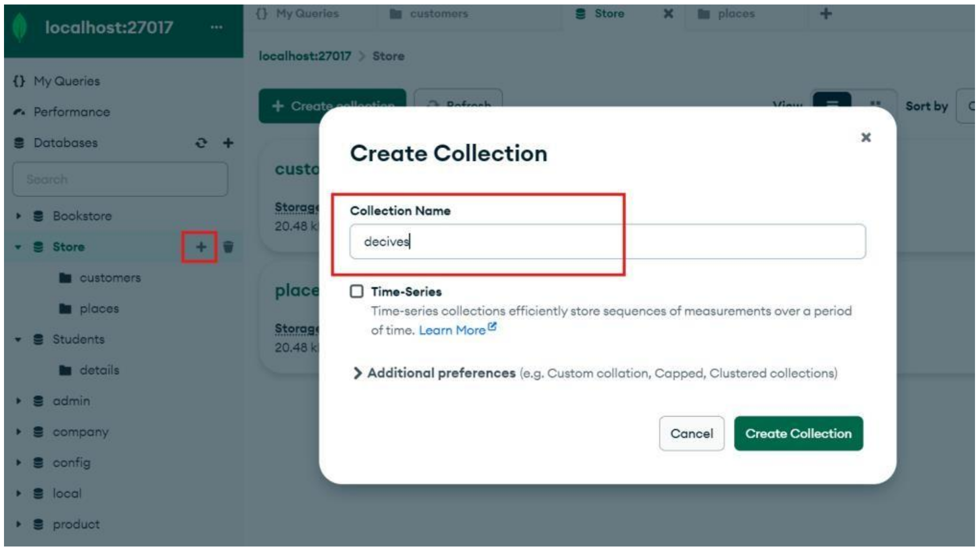
```
db.places.find({}, { location: 1, _id: 0
}).pretty();
```

Geospatial Query (Find places within 10km of a given point):

```
>db.places.find({ location: {

          $near: {
                $geometry:
                { type: "Point",
                coordinates: [ -73.97, 40.77 ]
        }, $maxDistance: 10000 // 10km in meters
    }}})
```

2. Bitwise Selectors

Under database **Store**, create a collection **devices** in Mongo DB IDE**.**

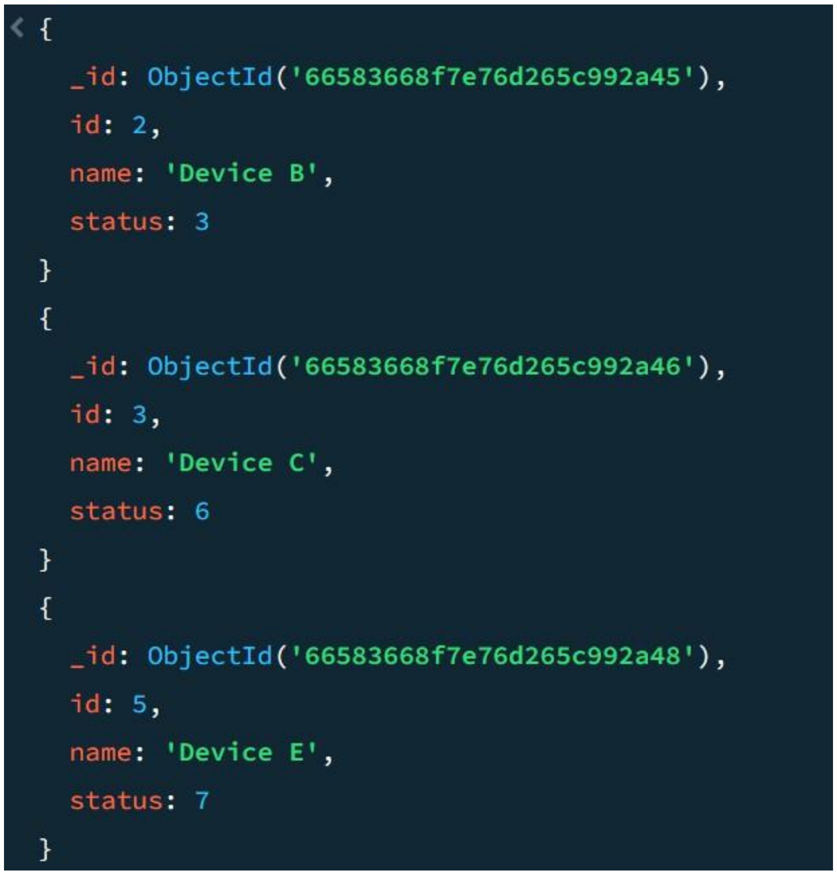


We'll use a collection devices with fields id, name, and status (where status is a bitwise flag).

```
>db.devices.insertMany([
  { _id: 1, name: "Device A", status: 5 }, // 0101 in binary
  { _id: 2, name: "Device B", status: 3 }, // 0011 in binary
  { _id: 3, name: "Device C", status: 6 }, // 0110 in binary
  { _id: 4, name: "Device D", status: 12 }, // 1100 in binary
  { _id: 5, name: "Device E", status: 7 }  // 0111 in binary ])
```

Bitwise AND Query (Find devices where the 2nd bit is set):

>db.devices.find({ status: { $bitsAllSet: 2 } })

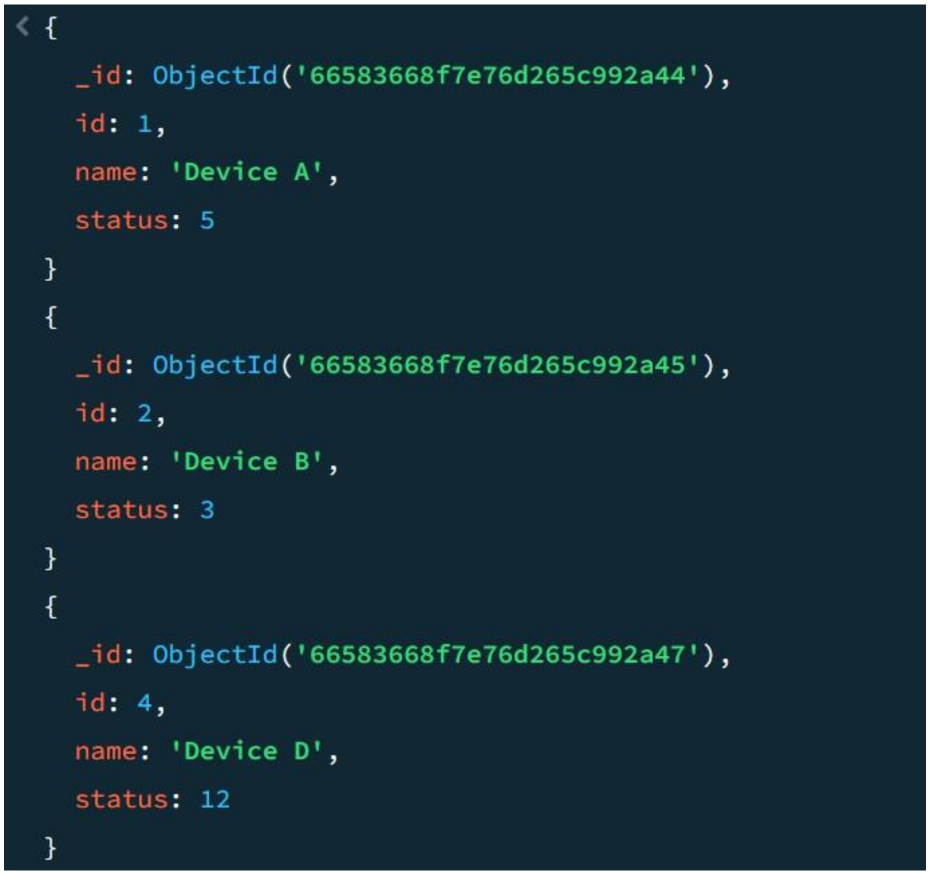


```
< {
    _id: ObjectId('66583668f7e76d265c992a45'),
    id: 2,
    name: 'Device B',
    status: 3
  }
  {
    _id: ObjectId('66583668f7e76d265c992a46'),
    id: 3,
    name: 'Device C',
    status: 6
  }
  {
    _id: ObjectId('66583668f7e76d265c992a48'),
    id: 5,
    name: 'Device E',
    status: 7
  }
```

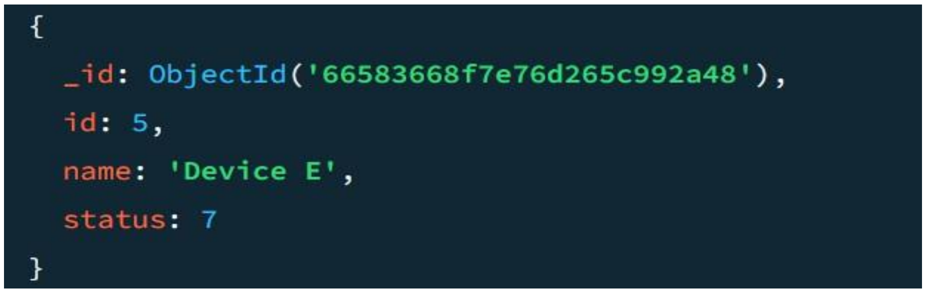
Bitwise OR Query (Find devices where any bit in 0101 is set):

To find all devices where any of the bits at positions 0 or 3 are set (i.e., either the least significant bit or the fourth bit is set), you can use the $bitsAnySet operator as follows:

> **db.devices.find({ "status": { "$bitsAnySet": [0, 3] } })**



```
< {
    _id: ObjectId('66583668f7e76d265c992a44'),
    id: 1,
    name: 'Device A',
    status: 5
  }
  {
    _id: ObjectId('66583668f7e76d265c992a45'),
    id: 2,
    name: 'Device B',
    status: 3
  }
  {
    _id: ObjectId('66583668f7e76d265c992a47'),
    id: 4,
    name: 'Device D',
    status: 12
  }
```



```
  {
    _id: ObjectId('66583668f7e76d265c992a48'),
    id: 5,
    name: 'Device E',
    status: 7
  }
```

In MongoDB, the main geospatial query operators include:

1. **$geoWithin**: Finds documents within a specified geometry (e.g., a polygon).
2. **$geoIntersects**: Finds documents that intersect with a specified geometry.
3. **$near**: Finds documents near a specified point, using a 2dsphere index.
4. **$nearSphere**: Similar to $near, but calculates distances using spherical geometry.
5. **$center**: Finds documents within a circular area (used with legacy coordinate pairs).
6. **$centerSphere**: Finds documents within a circular area on a sphere (used with legacy coordinate pairs).

7. **$box**: Finds documents within a rectangular area (used with legacy coordinate pairs).
8. **$polygon**: Finds documents within a polygon defined by multiple points (used with legacy coordinate pairs).

In MongoDB, the main bitwise query operators include:

1. **$bitsAllClear**: Matches documents where all of the given bit positions are clear (i.e., 0).
2. **$bitsAllSet**: Matches documents where all of the given bit positions are set (i.e., 1).
3. **$bitsAnyClear**: Matches documents where any of the given bit positions are clear (i.e., 0).
4. **$bitsAnySet**: Matches documents where any of the given bit positions are set (i.e., 1).

Explanation

- **Geospatial Selector**:
  - **$near**: Finds documents near a specified point. Requires a 2dsphere index on the location field.
  - **$geometry**: Specifies the reference point as a GeoJSON object.
  - **$maxDistance**: Limits the distance from the reference point (in meters).
- **Bitwise Selector**:
  - **$bitsAllSet**: Matches documents where all of the given bit positions are 1.
  - **$bitsAnySet**: Matches documents where any of the given bit positions are 1.

By executing these queries, you can filter documents based on geospatial proximity and bitwise conditions.

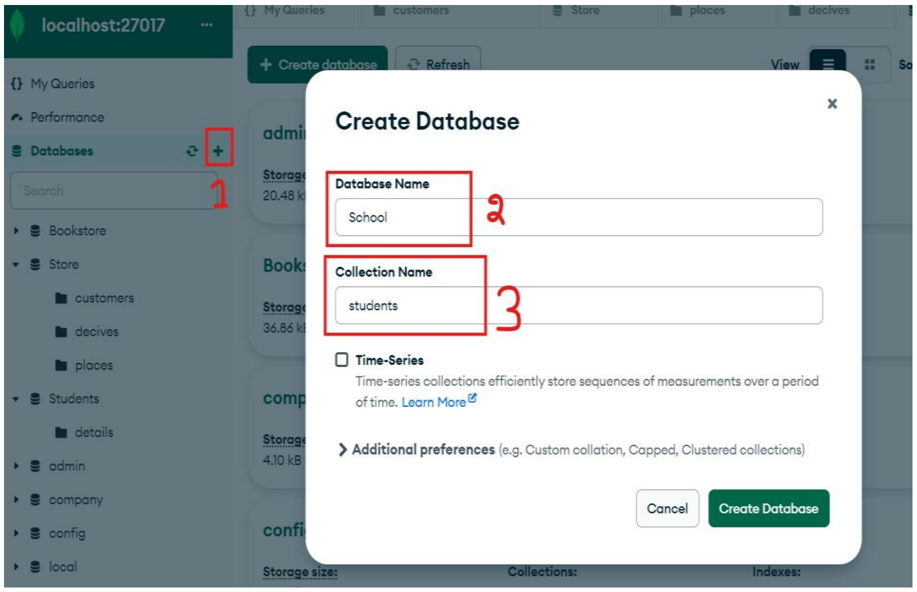
Program 4

Create and demonstrate how projection operators ($, $elematch and $slice) would be used in the MongoDB.

$elemMatch: The $elemMatch operator is used to match documents that contain an array field with at least one element that matches all the specified query criteria.

$slice: The $slice projection operator is used within the projection document to limit the number of elements returned from an array field.

Create a database **School** and collection **students** in Mongo DB IDE**.**



Create Database

Database Name: School

Collection Name: students

Time-Series

Time-series collections efficiently store sequences of measurements over a period of time. Learn More

Additional preferences (e.g. Custom collation, Capped, Clustered collections)

Add the following documents in the **details collection** in MongoDB IDE.

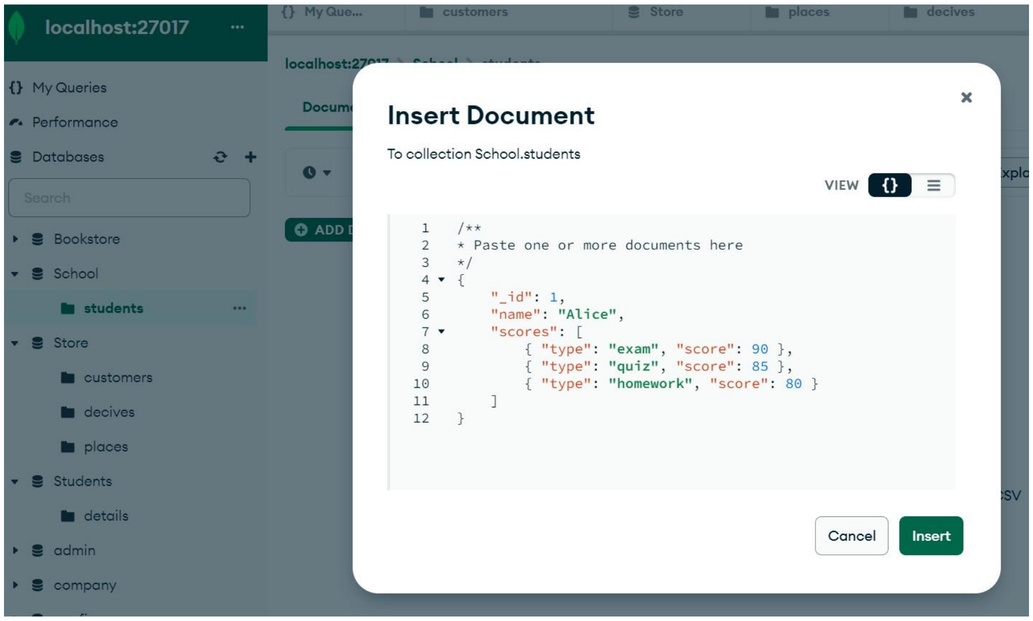
{

"_id": 1, "name":

"Alice",

"scores": [{ "type": "exam", "score": 90 }, { "type": "quiz", "score": 85 }, { "type": "homework", "score": 80 }]

}



```
/**
* Paste one or more documents here
*/
{
    "_id": 1,
    "name": "Alice",
    "scores": [
        { "type": "exam", "score": 90 },
        { "type": "quiz", "score": 85 },
        { "type": "homework", "score": 80 }
    ]
}
```

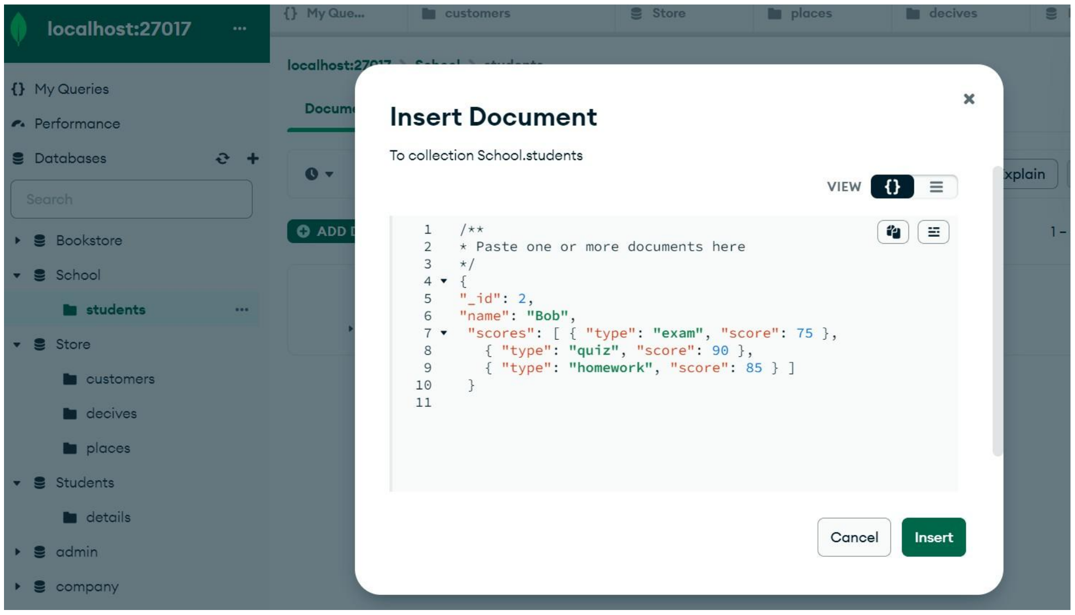
{

"_id": 2,

"name": "Bob",

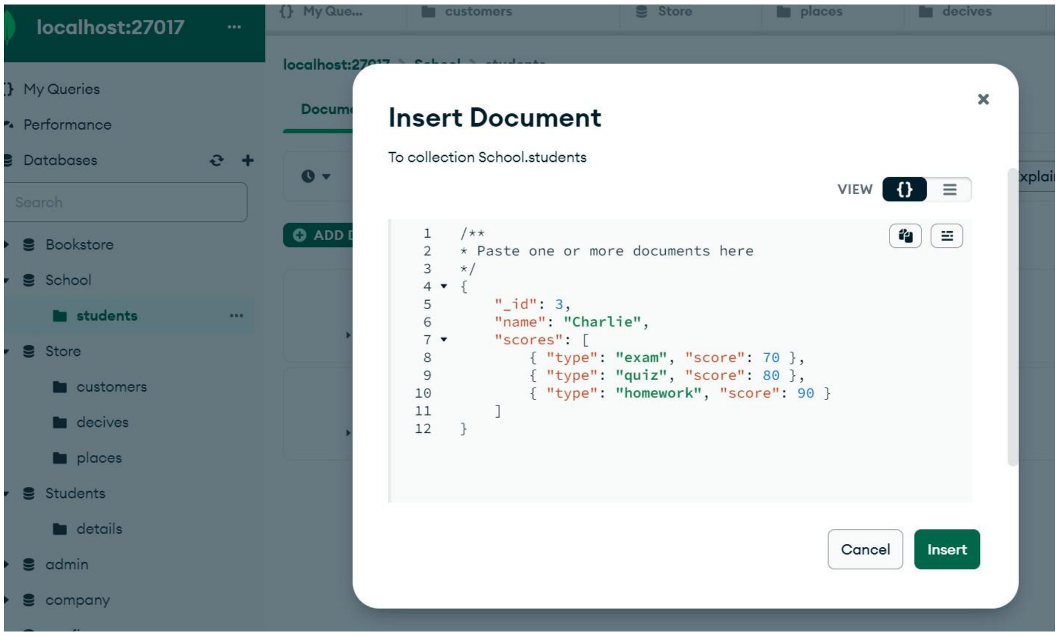
"scores": [{ "type": "exam", "score": 75 }, { "type": "quiz", "score": 90 }, { "type": "homework", "score": 85 }]

}



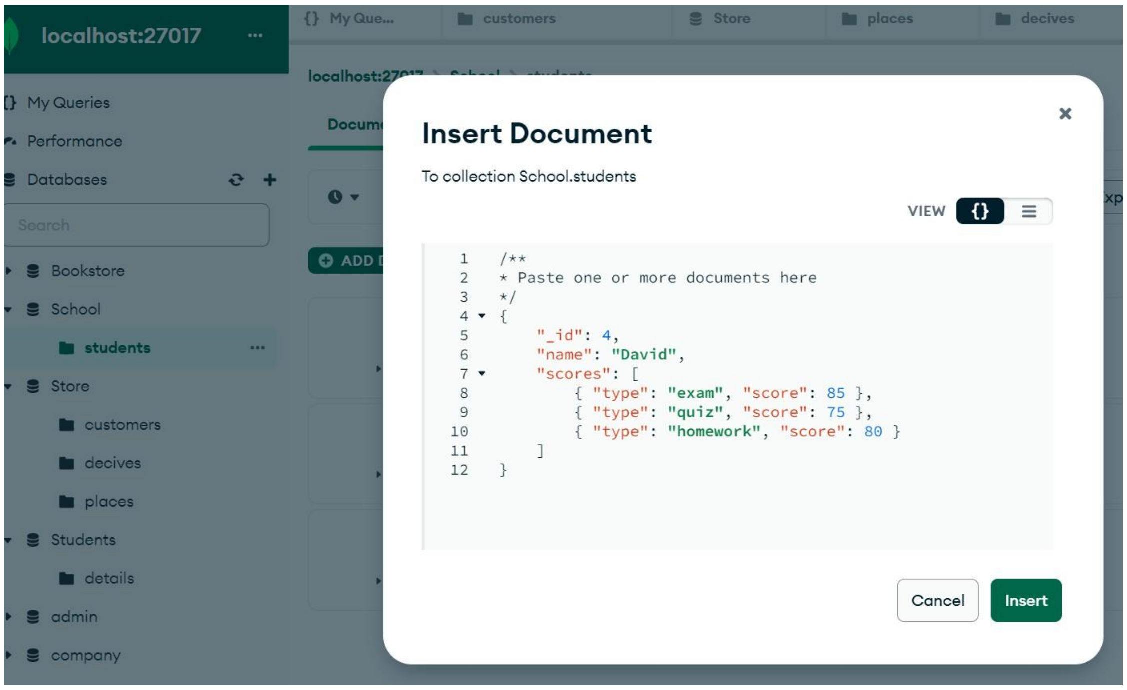
```
/**
* Paste one or more documents here
*/
{
"_id": 2,
"name": "Bob",
 "scores": [ { "type": "exam", "score": 75 },
   { "type": "quiz", "score": 90 },
   { "type": "homework", "score": 85 } ]
 }
```

```
{
  "_id": 3,
  "name": "Charlie",
  "scores": [
    { "type": "exam", "score": 70 },
    { "type": "quiz", "score": 80 },
    { "type": "homework", "score": 90 }
  ]
}
```



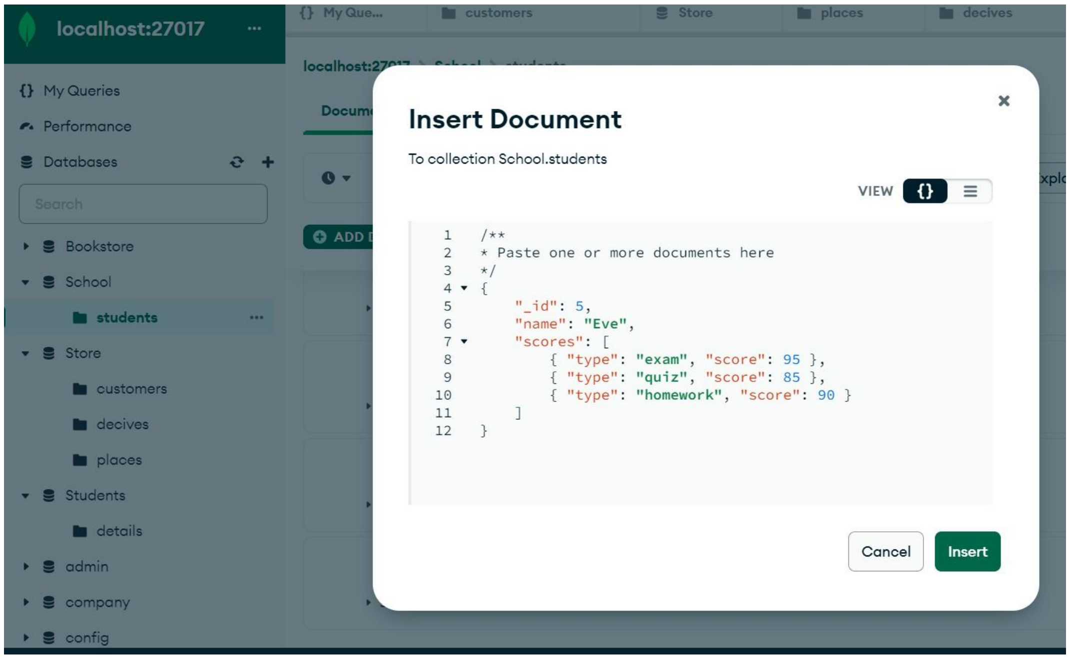
```
/**
* Paste one or more documents here
*/
{
    "_id": 3,
    "name": "Charlie",
    "scores": [
        { "type": "exam", "score": 70 },
        { "type": "quiz", "score": 80 },
        { "type": "homework", "score": 90 }
    ]
}
```

```
{
   "_id": 4,
   "name": "David",
   "scores": [
      { "type": "exam", "score": 85 },
      { "type": "quiz", "score": 75 },
      { "type": "homework", "score": 80 }
   ]
}
```



```
/**
* Paste one or more documents here
*/
{
    "_id": 4,
    "name": "David",
    "scores": [
        { "type": "exam", "score": 85 },
        { "type": "quiz", "score": 75 },
        { "type": "homework", "score": 80 }
    ]
}
```

```
{
  "_id": 5,
  "name": "Eve",
  "scores": [
    { "type": "exam", "score": 95 },
    { "type": "quiz", "score": 85 },
    { "type": "homework", "score": 90 }
  ]
}
```



```
/**
* Paste one or more documents here
*/
{
    "_id": 5,
    "name": "Eve",
    "scores": [
        { "type": "exam", "score": 95 },
        { "type": "quiz", "score": 85 },
        { "type": "homework", "score": 90 }
    ]
}
```

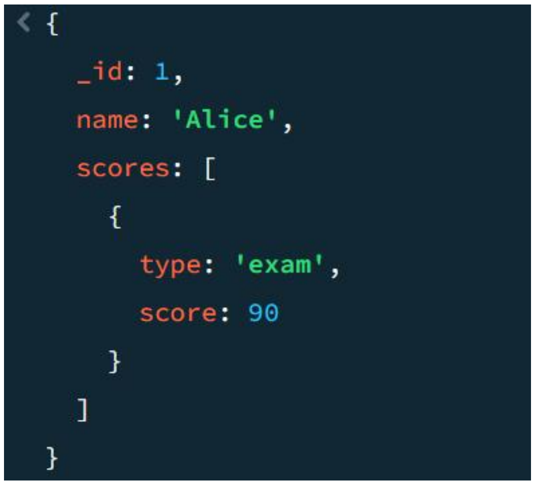
1. <u>$ Operator</u>

The $ operator is used to project a single element from an array that matches a specified condition. For instance, to find the exam score of Alice, you would use:

// To project only the first element in the grades array that is greater than or equal to 85, we can use the following query:

```
db.students.find(

  { "name": "Alice", "scores.type": "exam" },

  { "name": 1, "scores.$": 1 }

)
```

Output:



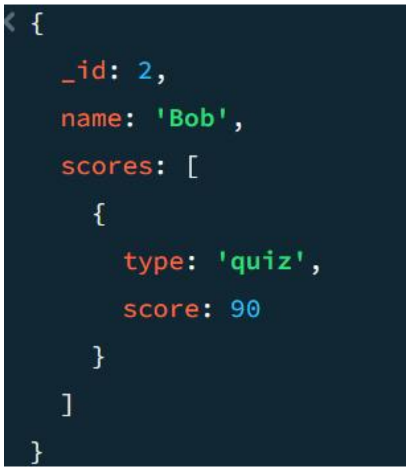
```
< {
    _id: 1,
    name: 'Alice',
    scores: [
      {
        type: 'exam',
        score: 90
      }
    ]
  }
```

2. $elemMatch Operator

The **$elemMatch** operator is used to project the first matching element from an array. To get the quiz score of Bob, you would use:

```
>db.students.find(
  { "name": "Bob" },
  { "name": 1, "scores": { $elemMatch: { "type": "quiz" } } }
)
```

Output:

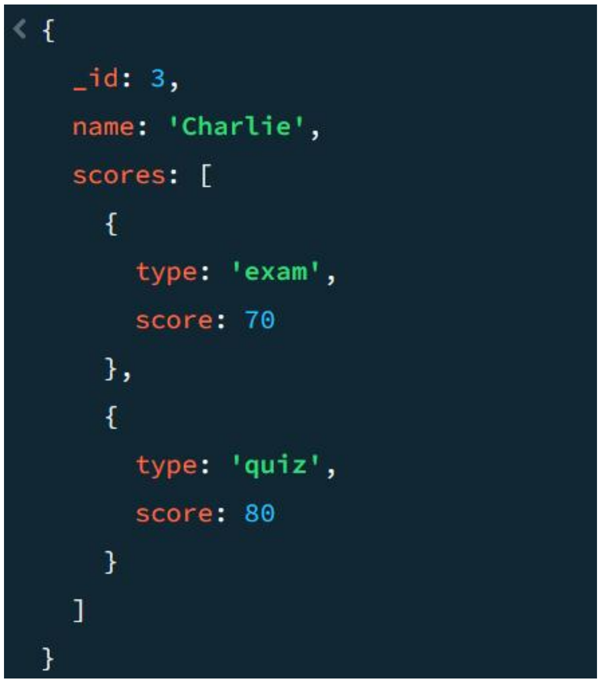


```
< {
    _id: 2,
    name: 'Bob',
    scores: [
      {
        type: 'quiz',
        score: 90
      }
    ]
  }
```

3. $slice Operator

The $slice operator is used to limit the number of elements returned from an array. To get the first two scores of Charlie, you would use:

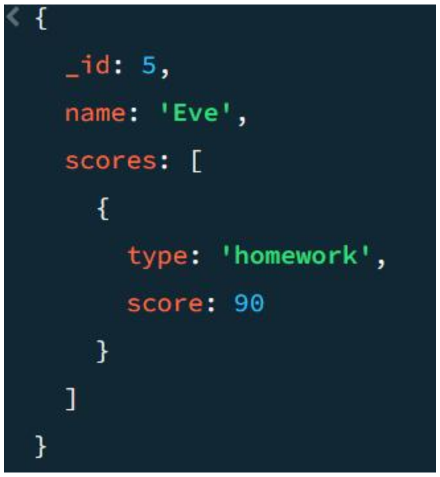
db.students.find({ "name": "Charlie" }, { scores: { $slice: 2 } });



```
< {
    _id: 3,
    name: 'Charlie',
    scores: [
      {
        type: 'exam',
        score: 70
      },
      {
        type: 'quiz',
        score: 80
      }
    ]
  }
```

Alternatively, you can use negative values with **$slice** to get elements from the end of the array. To get the last score of Eve, you would use:

Output:

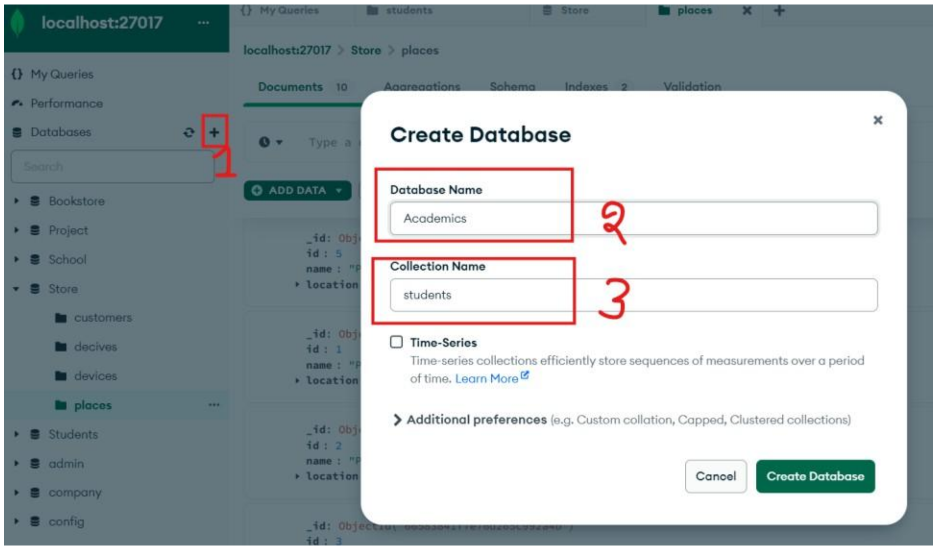


```
< {
    _id: 5,
    name: 'Eve',
    scores: [
      {
        type: 'homework',
        score: 90
      }
    ]
  }
```

Program 5

Execute Aggregation operations ($avg, $min, $max, $push, $addToSet etc.). Encourage students to execute several queries to demonstrate various aggregation operators.

Create a database of **Academics** and a collection of **students** in Mongo IDE.



Database Name: Academics

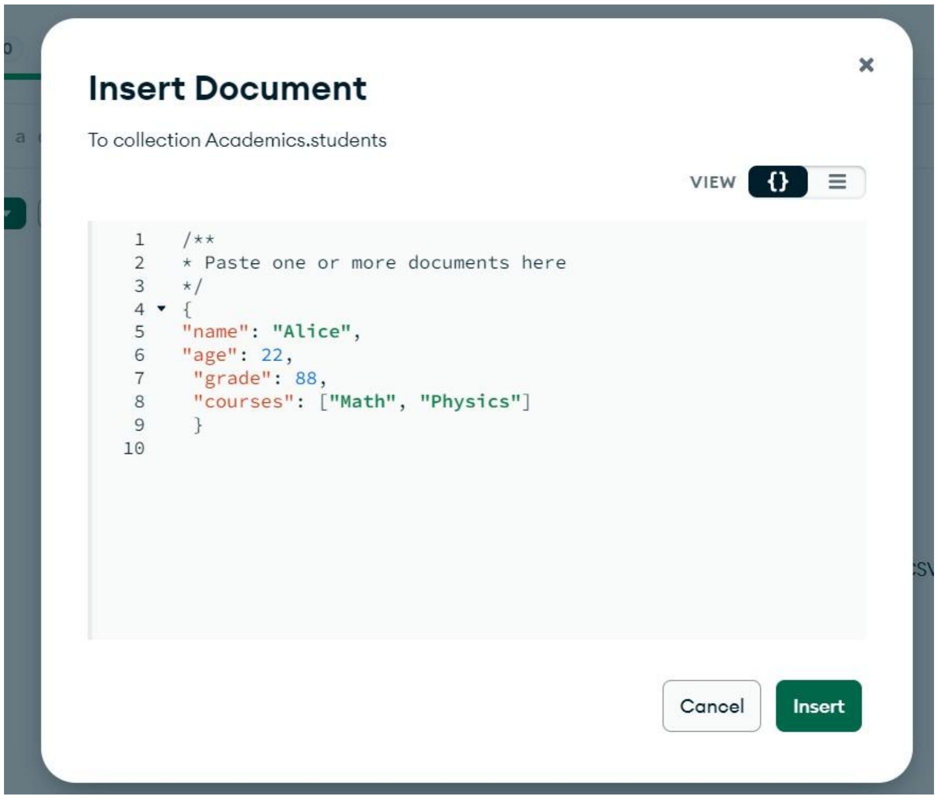
Collection Name: students

Time-series collections efficiently store sequences of measurements over a period of time. Learn More

(e.g. Custom collation, Capped, Clustered collections)

Add the following documents in the **students collection** in MongoDB IDE.

```
{
"name": "Alice",
"age": 22,
 "grade": 88,
 "courses": ["Math", "Physics"]
 }
```



```
/**
* Paste one or more documents here
*/
{
"name": "Alice",
"age": 22,
 "grade": 88,
 "courses": ["Math", "Physics"]
 }
```

{

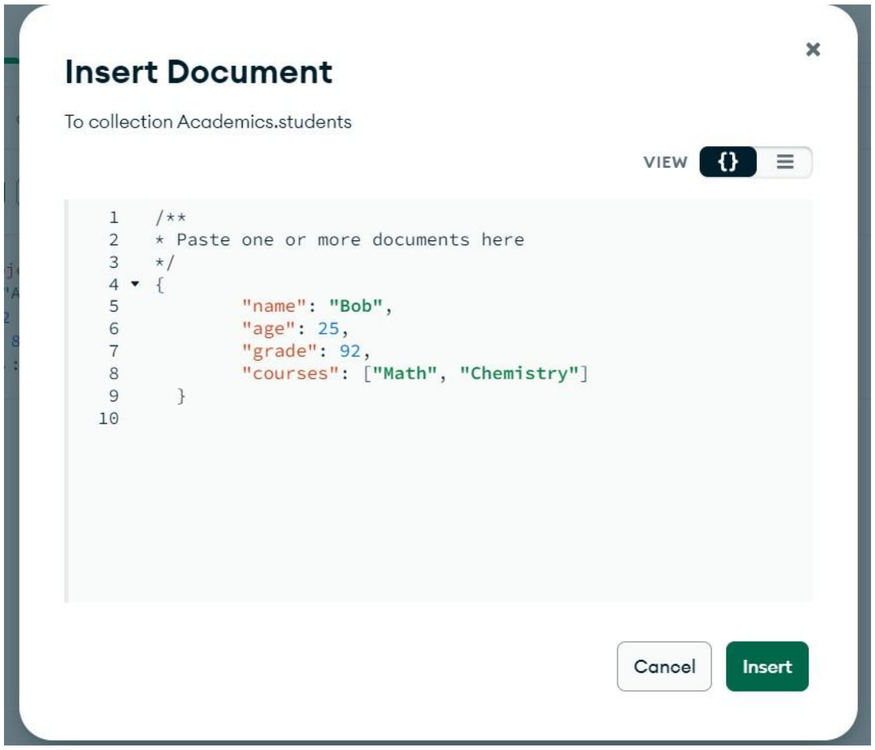
"name": "Bob",

"age": 25,

"grade": 92,

"courses": ["Math", "Chemistry"]

}



```
/**
* Paste one or more documents here
*/
{
        "name": "Bob",
        "age": 25,
        "grade": 92,
        "courses": ["Math", "Chemistry"]
  }
```

{

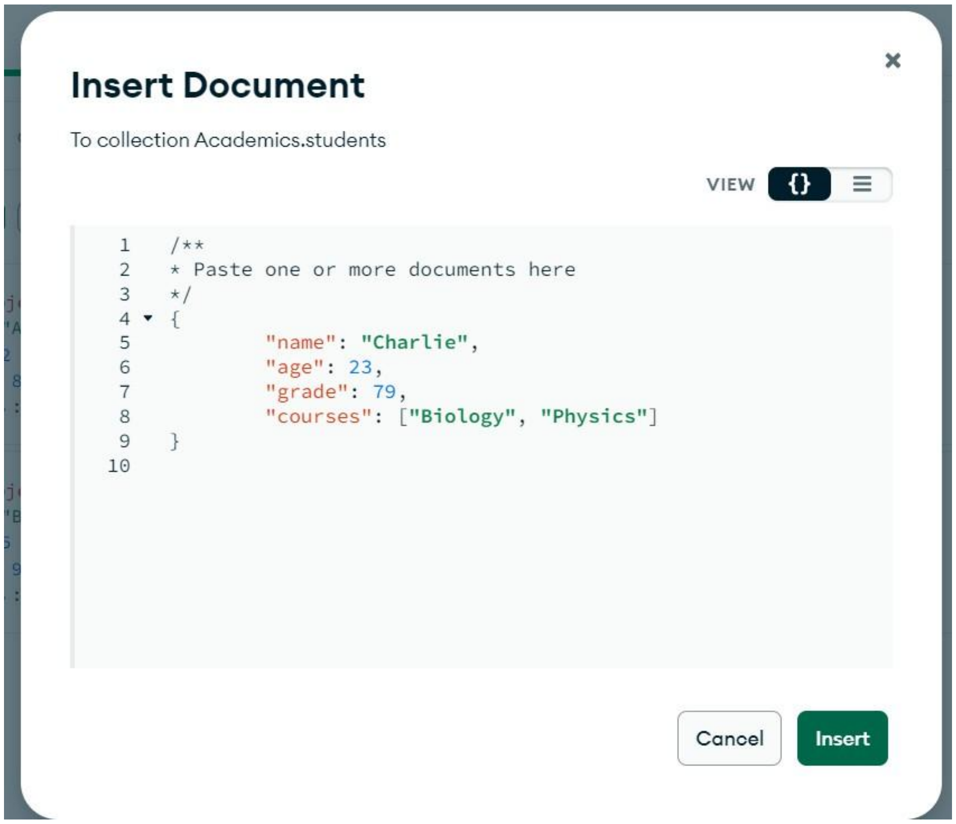
"name": "Charlie",

"age": 23,

"grade": 79,

"courses": ["Biology", "Physics"]

}



```
/**
* Paste one or more documents here
*/
{
        "name": "Charlie",
        "age": 23,
        "grade": 79,
        "courses": ["Biology", "Physics"]
}
```

{

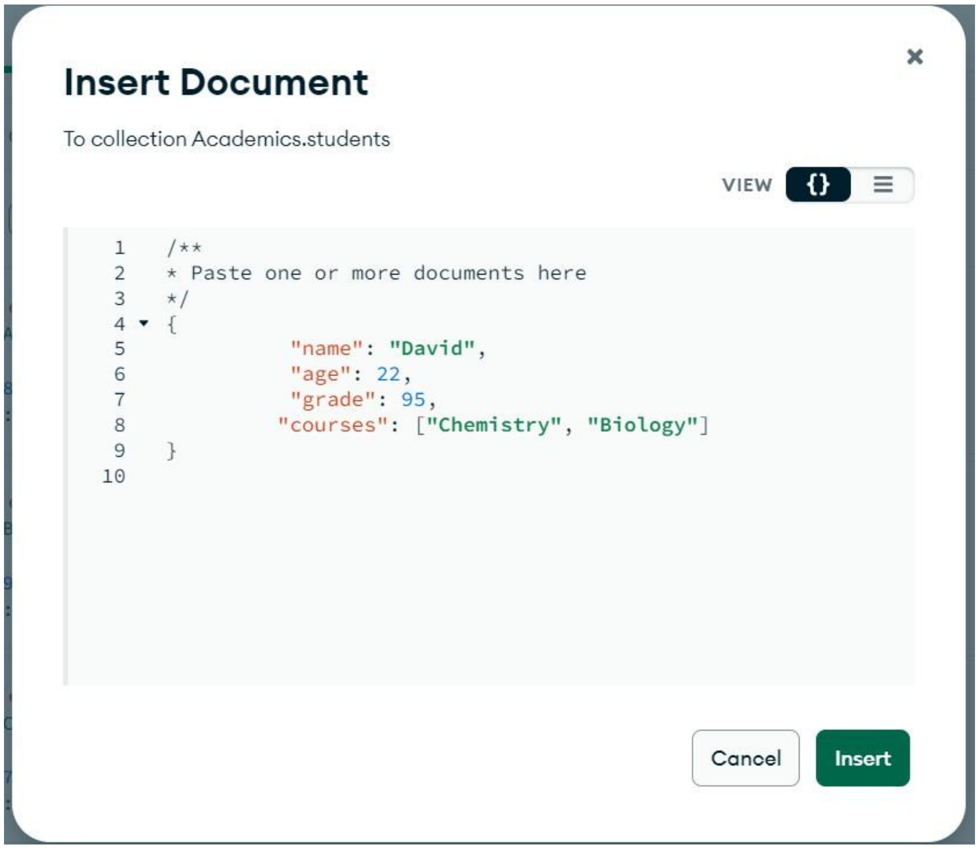
"name": "David",

"age": 22,

"grade": 95,

"courses": ["Chemistry", "Biology"]

}



```
/**
* Paste one or more documents here
*/
{
           "name": "David",
           "age": 22,
           "grade": 95,
          "courses": ["Chemistry", "Biology"]
}
```

{

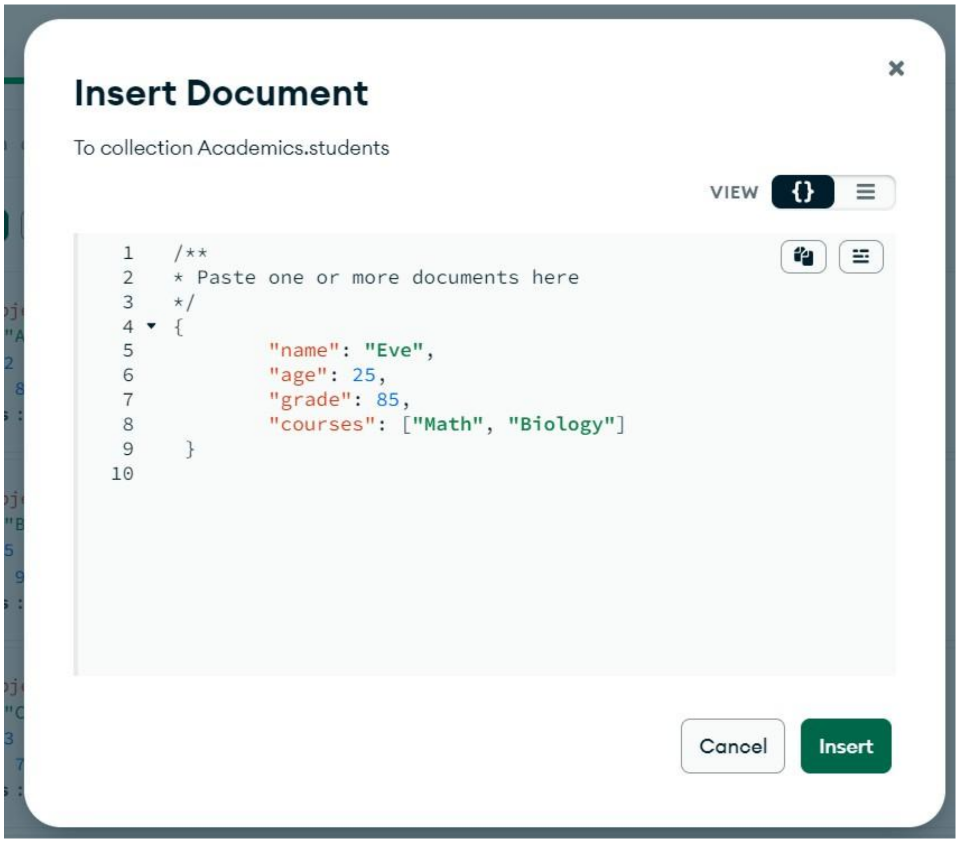
"name": "Eve",

"age": 25,

"grade": 85,

"courses": ["Math", "Biology"]

}



```
/**
 * Paste one or more documents here
 */
{
        "name": "Eve",
        "age": 25,
        "grade": 85,
        "courses": ["Math", "Biology"]
 }
```

1. **$avg** - Calculate the average grade of all students

   > db.students.aggregate([

   {

   $group: {

   _id: null,

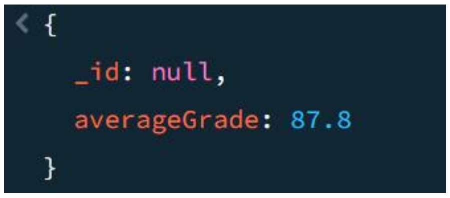
   averageGrade: { $avg: "$grade" }

   }

   }

   ])

   Output:

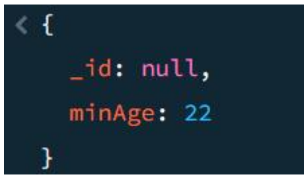


```
< {
    _id: null,
    averageGrade: 87.8
  }
```

2. **$min** - Find the minimum age of students

```
>db.students.aggregate([
   {
      $group: {
         _id: null,
         minAge: { $min: "$age" }
      }
   }
])
```

Output:

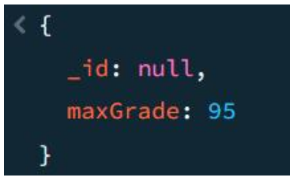


```
< {
    _id: null,
    minAge: 22
  }
```

3. **$max** - Find the maximum grade among students

```
>db.students.aggregate([
   {
      $group: {
         _id: null,
         maxGrade: { $max: "$grade" }
      }
   }
])
```

Output:

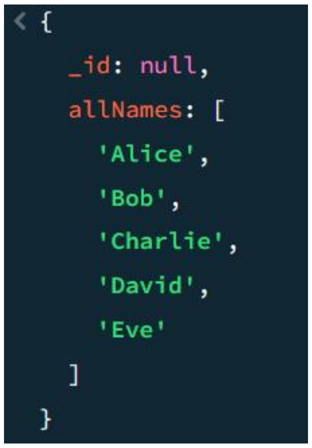


```
< {
    _id: null,
    maxGrade: 95
  }
```

4. **$push** - List all student names in an array

```
db.students.aggregate([
  {
    $group: {
      _id: null,
      allNames: { $push: "$name" }
    }
  }
])
```

Output:

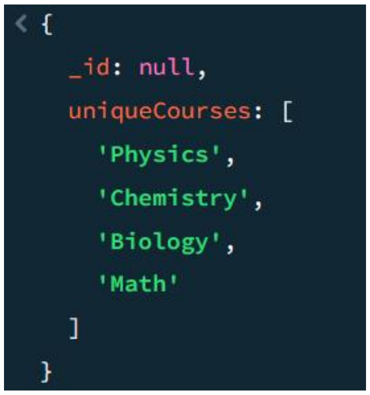


```
< {
    _id: null,
    allNames: [
      'Alice',
      'Bob',
      'Charlie',
      'David',
      'Eve'
    ]
  }
```

5. **$addToSet** - List all unique courses taken by students

```
>db.students.aggregate([

  {

    $unwind: "$courses"

  },

  {

    $group: {
            _id: null,

            uniqueCourses: { $addToSet: "$courses" }

        }

    }

])
```

Output:

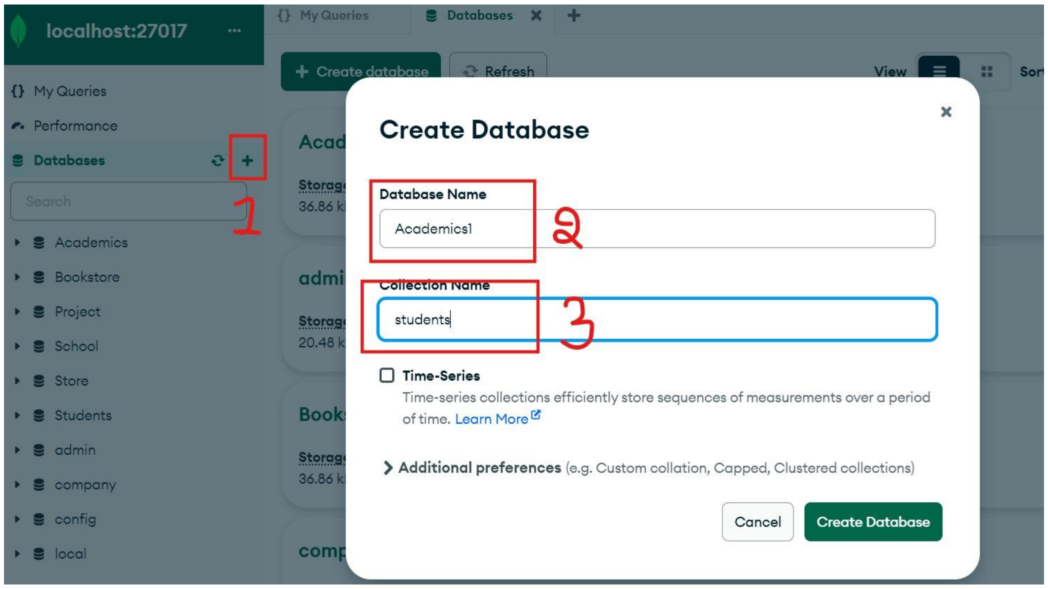


```
< {
    _id: null,
    uniqueCourses: [
      'Physics',
      'Chemistry',
      'Biology',
      'Math'
    ]
  }
```

Program 6

Execute Aggregation Pipeline and its operations (pipeline must contain $match, $group, $sort, $project, $skip etc. students encourage to execute several queries to demonstrate various aggregation operators)

Create a database **Academics1** and collection **students** in Mongo IDE.



Create Database

Database Name: Academics1

Collection Name: students

Time-Series

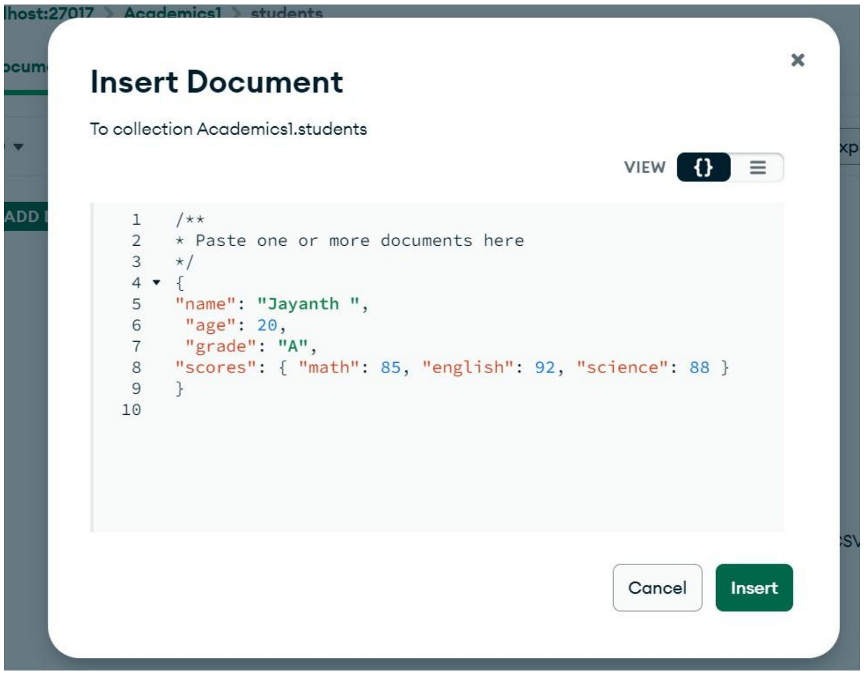
Time-series collections efficiently store sequences of measurements over a period of time. Learn More

Additional preferences (e.g. Custom collation, Capped, Clustered collections)

Cancel | Create Database

Add the following documents in the **students collection** in MongoDB IDE.

```
{
        "name": "Jayanth ",
         "age": 20,
         "grade": "A",
        "scores": { "math": 85, "english": 92, "science": 88 }
}
```



```
/**
* Paste one or more documents here
*/
{
"name": "Jayanth ",
 "age": 20,
 "grade": "A",
"scores": { "math": 85, "english": 92, "science": 88 }
}
```

{

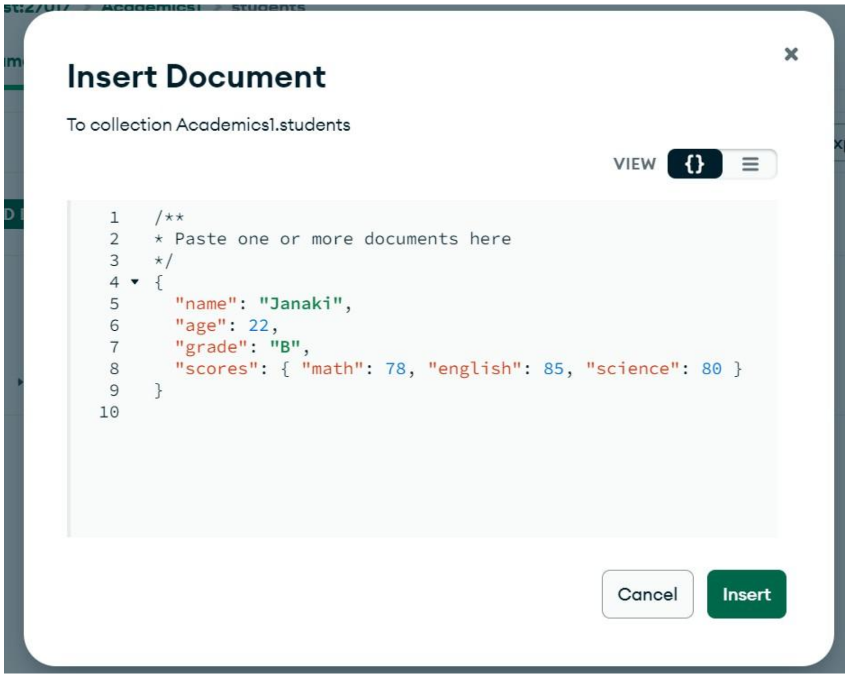
"name": "Janaki",

"age": 22, "grade":

"B",

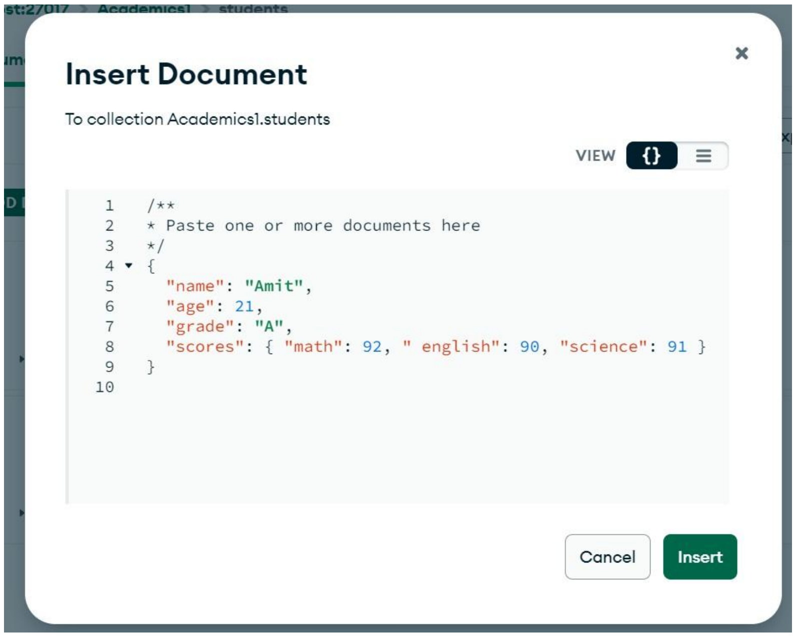
"scores": { "math": 78, "english": 85, "science": 80 }

}



```
/**
* Paste one or more documents here
*/
{
  "name": "Janaki",
  "age": 22,
  "grade": "B",
  "scores": { "math": 78, "english": 85, "science": 80 }
}
```

```
{
        "name": "Amit",
        "age": 21,
        "grade": "A",
        "scores": { "math": 92, " english": 90, "science": 91 }
}
```

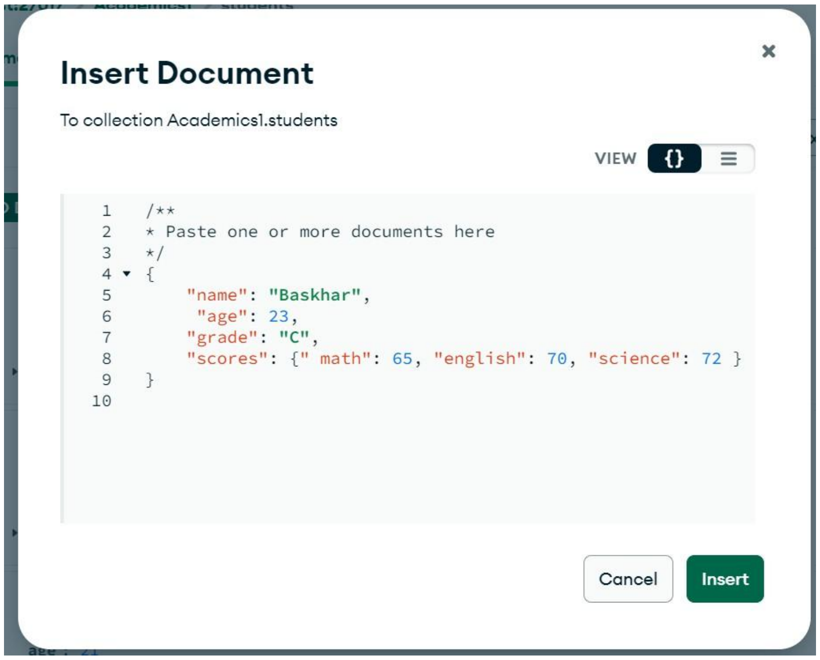


Insert Document

To collection Academics1.students

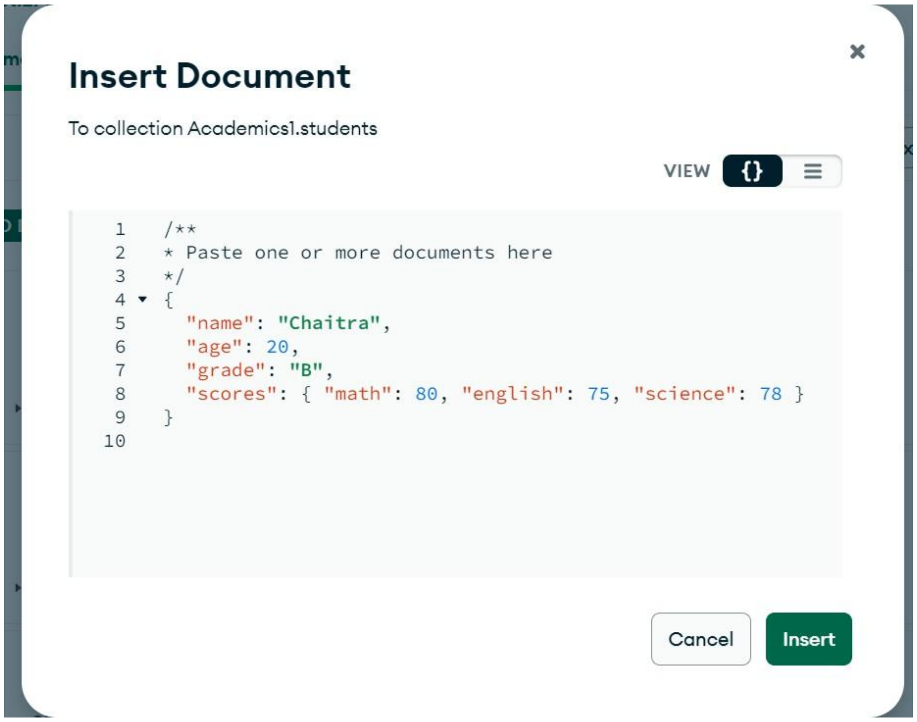
```
/**
* Paste one or more documents here
*/
{
  "name": "Amit",
  "age": 21,
  "grade": "A",
  "scores": { "math": 92, " english": 90, "science": 91 }
}
```

```
{
        "name": "Baskhar",
         "age": 23,
        "grade": "C",
        "scores": {" math": 65, "english": 70, "science": 72 }
}
```



```
/**
* Paste one or more documents here
*/
{
    "name": "Baskhar",
     "age": 23,
    "grade": "C",
    "scores": {" math": 65, "english": 70, "science": 72 }
}
```

```
{
        "name": "Chaitra",
        "age": 20,
        "grade": "B",
        "scores": { "math": 80, "english": 75, "science": 78 }
}
```



```
/**
* Paste one or more documents here
*/
{
  "name": "Chaitra",
  "age": 20,
  "grade": "B",
  "scores": { "math": 80, "english": 75, "science": 78 }
}
```

In MongoDB Shell

>use Academics1

Now, let's execute an aggregation pipeline with several stages:

1. **$match**: Filter students who are 21 years or older.
2. **$group**: Group by grade and calculate the average age.
3. **$sort**: Sort by average age in descending order.
4. **$project**: Project the grade and average age.
5. **$skip**: Skip the first result.

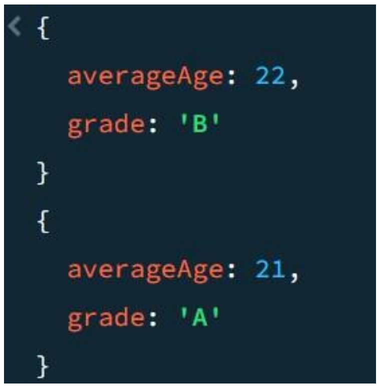
```
db.students.aggregate([

 {

      $match: { age: { $gte: 21 } }
```

```
  },


 {

      $group: {

     _id: "$grade",

     averageAge: { $avg: "$age" }

     }

 },

 {

   $sort: { averageAge: -1 }

 },

 {

   $project: {

     _id: 0,

     grade: "$_id",

     averageAge: 1

   }

 },

 {
```

$skip: 1

}])

Output:



```
< {
    averageAge: 22,
    grade: 'B'
  }
  {
    averageAge: 21,
    grade: 'A'
  }
```

Let's break down each stage:

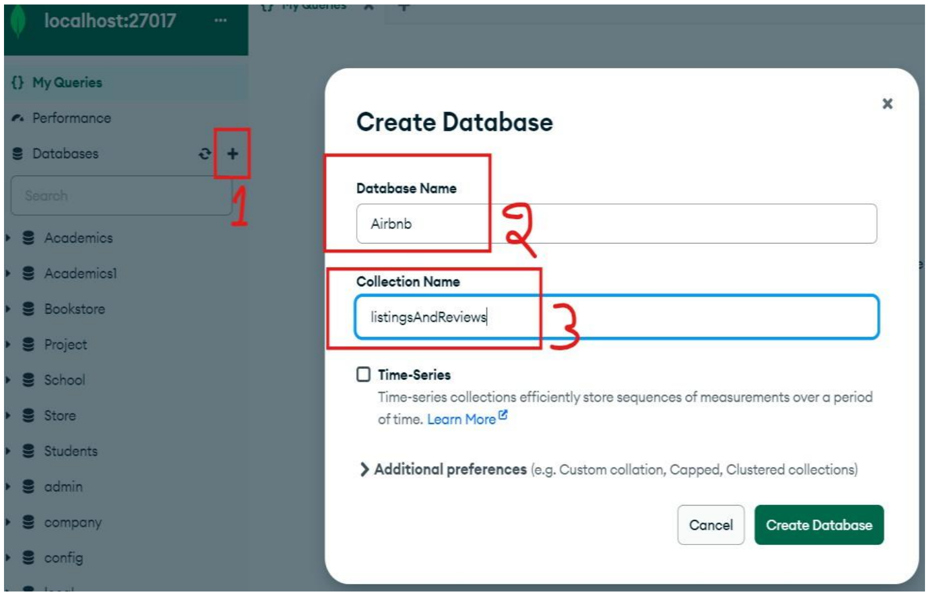
1. **$match**: Filters documents to include only those where age is greater than or equal to 21.
2. **$group**: Groups the documents by grade and computes the average age for each grade.
3. **$sort**: Sorts the resulting documents by averageAge in descending order.
4. **$project**: Projects the fields grade and averageAge, excluding the _id field.
5. **$skip**: Skips the first document in the sorted results.

When you execute this pipeline, you will get a result that first filters students by age, groups them by grade, calculates the average age, sorts by this average age in descending order, and finally skips the first result.

Program 7

a. **Find all listings with listing_url, name, address, host_picture_url in the listings And Reviews collection that have a host with a picture url.**

Create a database **Airbnb** and collection **listingsAndReviews** in Mongo IDE.


Create Database

Database Name: Airbnb

Collection Name: listingsAndReviews

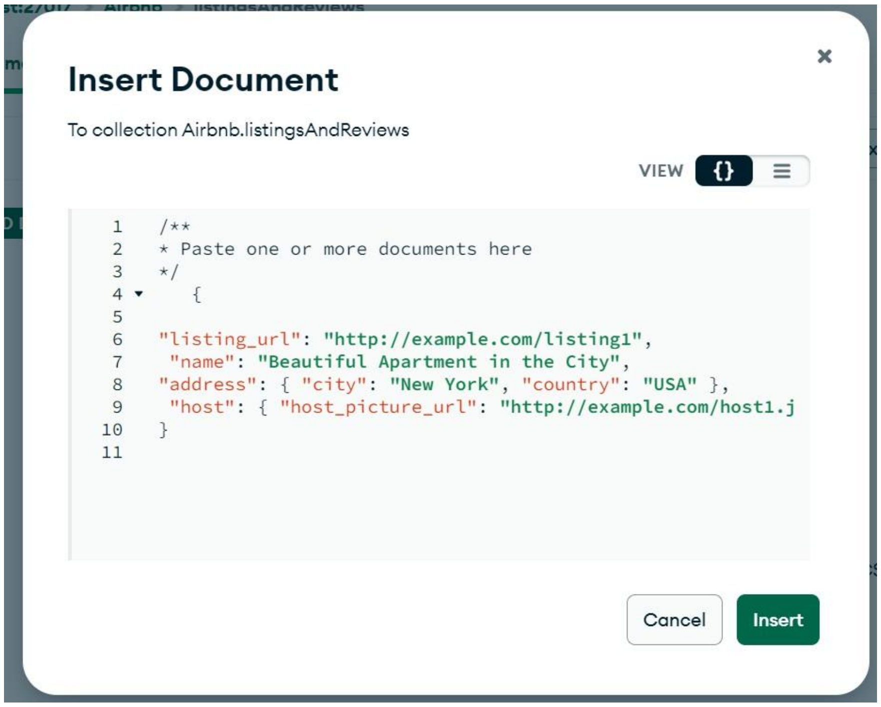
Time-Series

Time-series collections efficiently store sequences of measurements over a period of time. Learn More

Additional preferences (e.g. Custom collation, Capped, Clustered collections)

Add the following documents in the **listingsAndReviews collection** in MongoDB IDE.

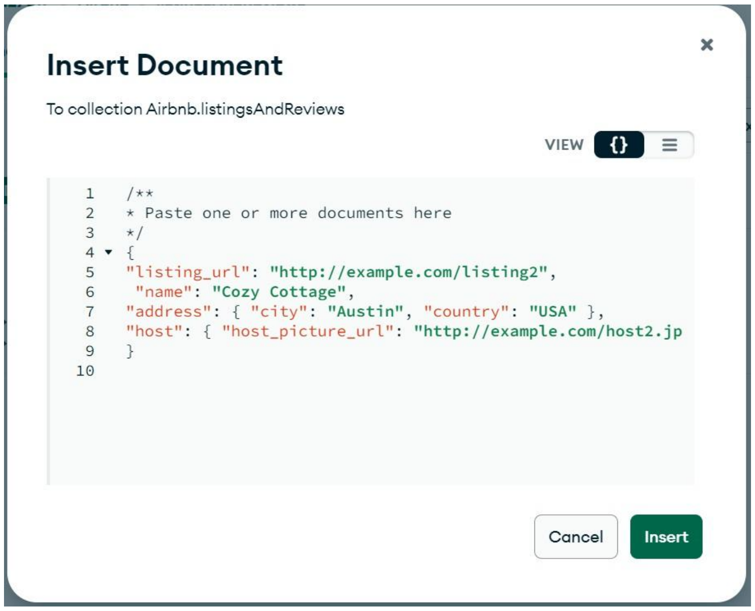
```
{

"listing_url": "http://example.com/listing1", "name":
 "Beautiful Apartment in the City",
"address": { "city": "New York", "country": "USA" },
 "host": { "host_picture_url": "http://example.com/host1.jpg" }
}
```



To collection Airbnb.listingsAndReviews

```
/**
* Paste one or more documents here
*/
   {

"listing_url": "http://example.com/listing1",
 "name": "Beautiful Apartment in the City",
"address": { "city": "New York", "country": "USA" },
 "host": { "host_picture_url": "http://example.com/host1.j
}

```

```
{
"listing_url": "http://example.com/listing2",
 "name": "Cozy Cottage",
"address": { "city": "Austin", "country": "USA" },
"host": { "host_picture_url": "http://example.com/host2.jpg" }
}
```



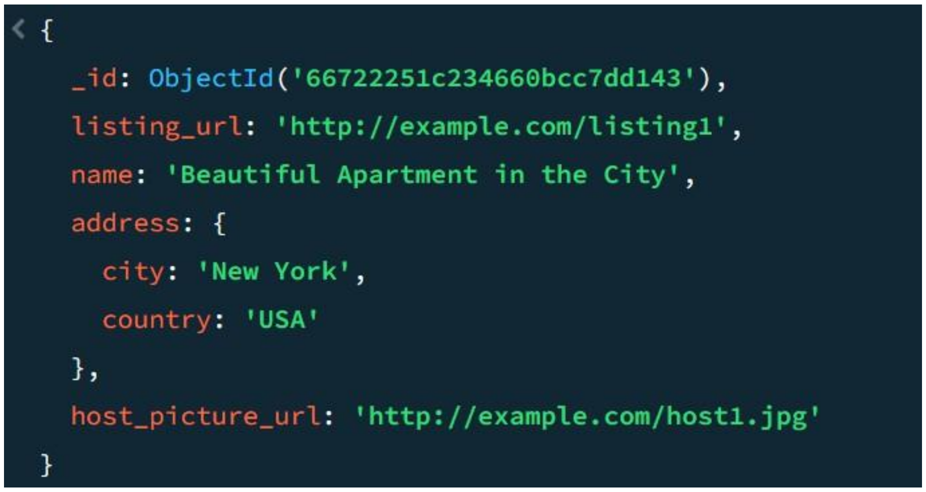
```
/**
* Paste one or more documents here
*/
{
"listing_url": "http://example.com/listing2",
 "name": "Cozy Cottage",
"address": { "city": "Austin", "country": "USA" },
"host": { "host_picture_url": "http://example.com/host2.jp
}
```

In MongoDB Shell

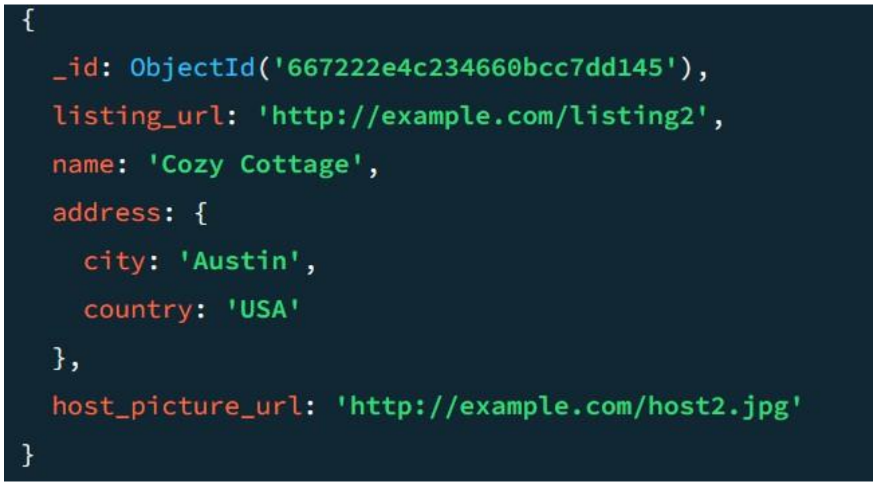
>use Airbnb

```
db.listingsAndReviews.aggregate([
  { $match: { "host.host_picture_url": { $exists: true, $ne: null } } },
    {
    $project:
      { listing_url:
      1,
      name: 1,
      address: 1,
      host_picture_url: "$host.host_picture_url" }
  }])
```

Output:



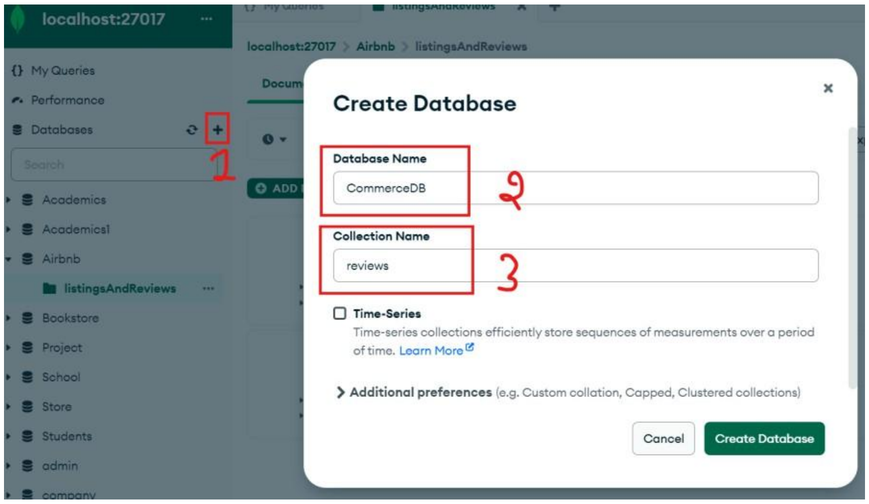
```
< {
    _id: ObjectId('66722251c234660bcc7dd143'),
    listing_url: 'http://example.com/listing1',
    name: 'Beautiful Apartment in the City',
    address: {
      city: 'New York',
      country: 'USA'
    },
    host_picture_url: 'http://example.com/host1.jpg'
  }
```



```
{
  _id: ObjectId('667222e4c234660bcc7dd145'),
  listing_url: 'http://example.com/listing2',
  name: 'Cozy Cottage',
  address: {
    city: 'Austin',
    country: 'USA'
  },
  host_picture_url: 'http://example.com/host2.jpg'
}
```

b. Using E-commerce collection write a query to display reviews summary.

Create a database **CommerceDB** and collection **reviews** in Mongo IDE.



Create Database

Database Name: CommerceDB

Collection Name: reviews

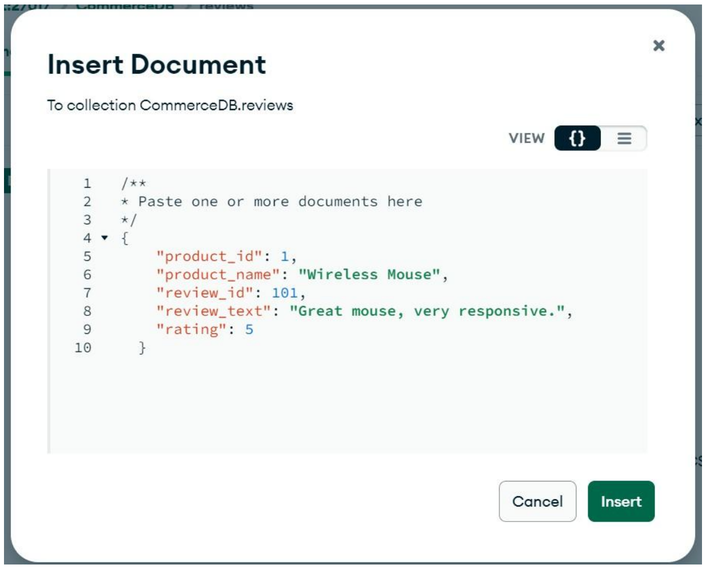
Time-Series
Time-series collections efficiently store sequences of measurements over a period of time. Learn More

Additional preferences (e.g. Custom collation, Capped, Clustered collections)

Cancel | Create Database

Add the following documents in the **reviews collection** in MongoDB IDE.

```
{
    "product_id": 1,
    "product_name": "Wireless Mouse",
    "review_id": 101,
    "review_text": "Great mouse, very responsive.",
    "rating": 5
 }
```

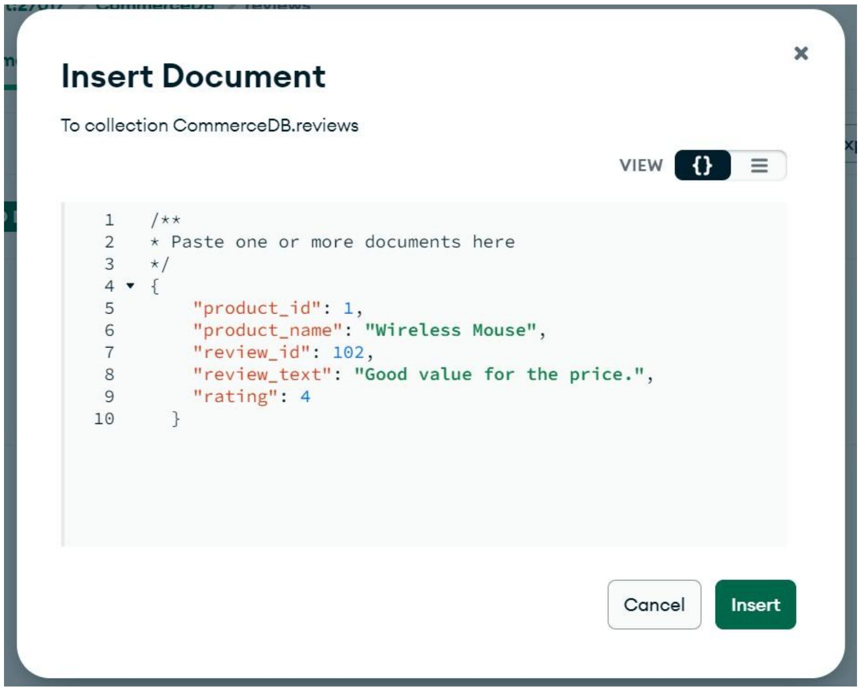

Insert Document

To collection CommerceDB.reviews

```
/**
* Paste one or more documents here
*/
{
    "product_id": 1,
    "product_name": "Wireless Mouse",
    "review_id": 101,
    "review_text": "Great mouse, very responsive.",
    "rating": 5
  }
```

Cancel | Insert

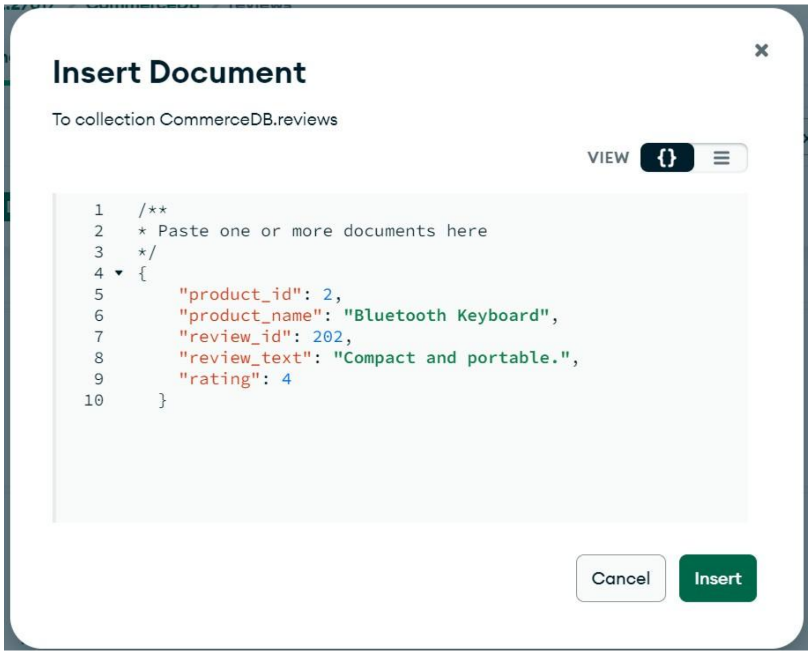
```
{
        "product_id": 1,
         "product_name": "Wireless Mouse",
        "review_id": 102,
        "review_text": "Good value for the price.",
        "rating": 4
}
```



```
/**
* Paste one or more documents here
*/
{
    "product_id": 1,
    "product_name": "Wireless Mouse",
    "review_id": 102,
    "review_text": "Good value for the price.",
    "rating": 4
  }
```

```
{
   "product_id": 2,
   "product_name": "Bluetooth Keyboard", "review_id":
   202,
   "review_text": "Compact and portable.",
   "rating": 4
```

}



```
/**
* Paste one or more documents here
*/
{
    "product_id": 2,
    "product_name": "Bluetooth Keyboard",
    "review_id": 202,
    "review_text": "Compact and portable.",
    "rating": 4
  }
```

{

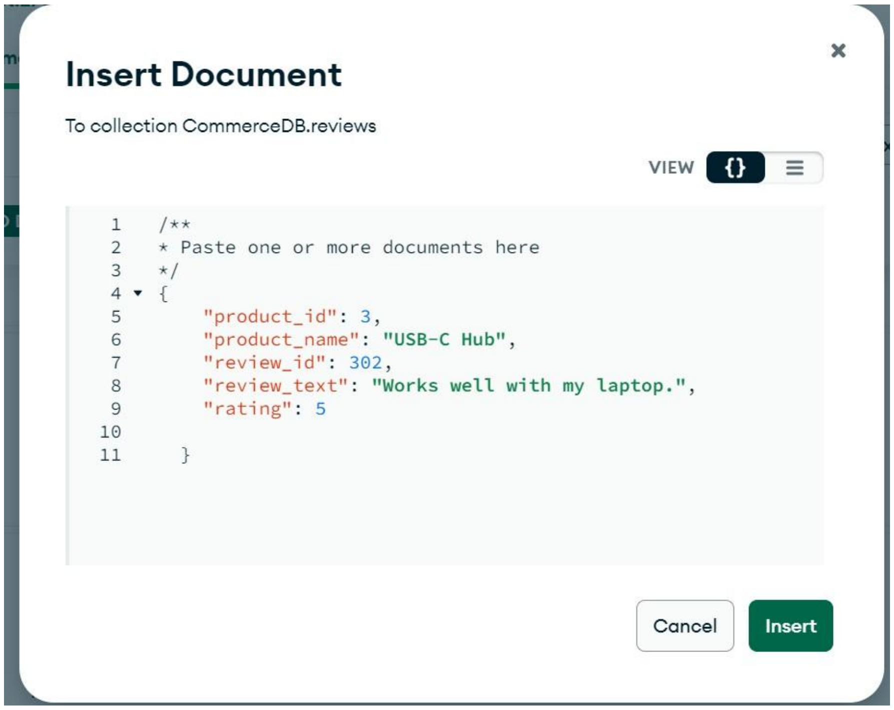
"product_id": 3, "product_name":

"USB-C Hub", "review_id": 302,

"review_text": "Works well with my laptop.",

"rating": 5

}



```
/**
* Paste one or more documents here
*/
{
    "product_id": 3,
    "product_name": "USB-C Hub",
    "review_id": 302,
    "review_text": "Works well with my laptop.",
    "rating": 5

  }
```

In MongoShell

>use CommerceDB

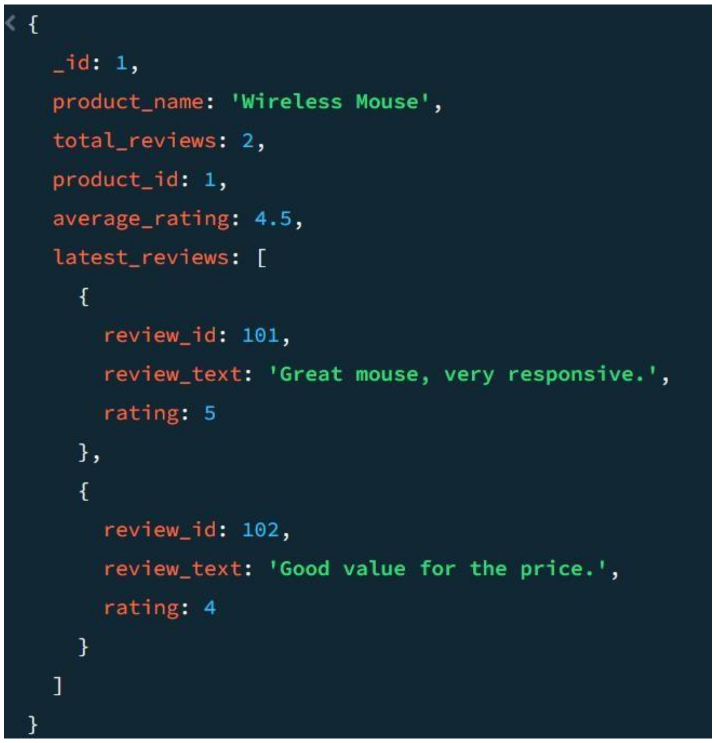
Query to Display Reviews Summary

```
db.reviews.aggregate([
 {
   $group: {
     _id: "$product_id",
     product_name: { $first: "$product_name" },
     total_reviews: { $sum: 1 },
     average_rating: { $avg: "$rating" },
        latest_reviews: {
          $push: {
            review_id: "$review_id",
            review_text: "$review_text",
            rating: "$rating",
          }
        }
      }
    },
    {
      $project:
        { product_id:
        "$_id",
        product_name: 1,
        total_reviews: 1,
        average_rating: "$average_rating",
        latest_reviews: {
          $slice: ["$latest_reviews", -3] // Adjust the number of latest reviews as needed
        }
```

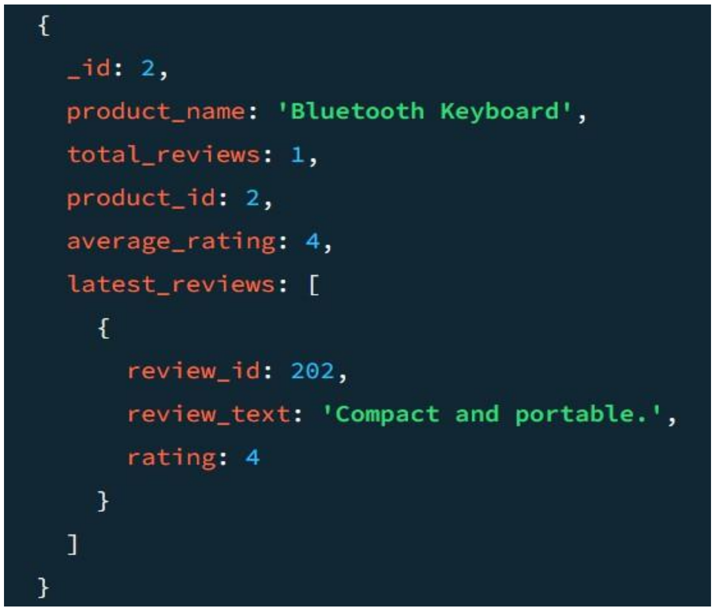
```
}
```

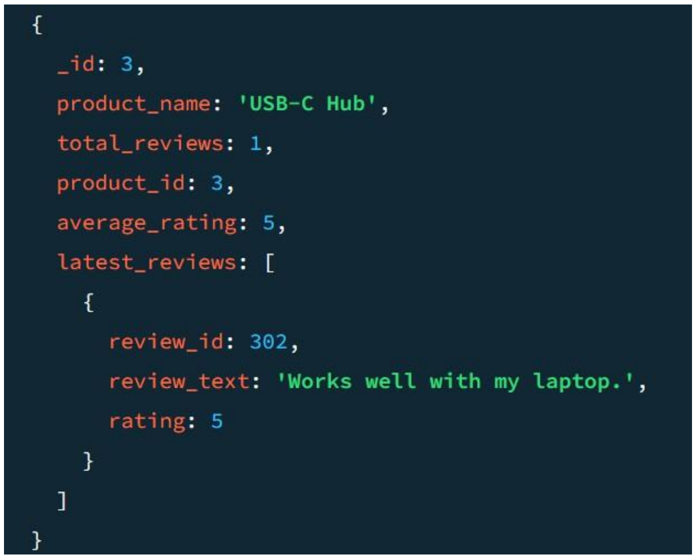
```
 },
 {
  $sort: { total_reviews: -1 } // Sort by total reviews in descending order
 }
])
```

Output:



```
< {
    _id: 1,
    product_name: 'Wireless Mouse',
    total_reviews: 2,
    product_id: 1,
    average_rating: 4.5,
    latest_reviews: [
      {
        review_id: 101,
        review_text: 'Great mouse, very responsive.',
        rating: 5
      },
      {
        review_id: 102,
        review_text: 'Good value for the price.',
        rating: 4
      }
    ]
  }
```

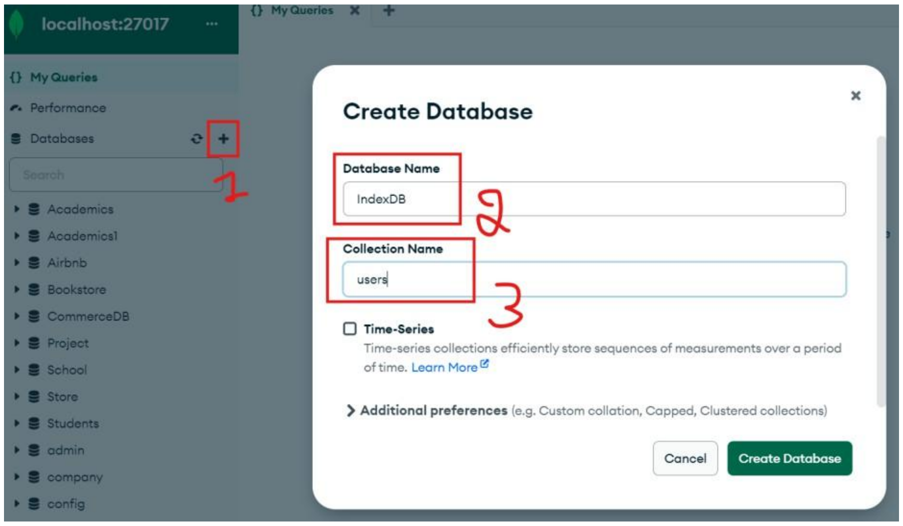

```
{
  _id: 2,
  product_name: 'Bluetooth Keyboard',
  total_reviews: 1,
  product_id: 2,
  average_rating: 4,
  latest_reviews: [
    {
      review_id: 202,
      review_text: 'Compact and portable.',
      rating: 4
    }
  ]
}
```


```
{
  _id: 3,
  product_name: 'USB-C Hub',
  total_reviews: 1,
  product_id: 3,
  average_rating: 5,
  latest_reviews: [
    {
      review_id: 302,
      review_text: 'Works well with my laptop.',
      rating: 5
    }
  ]
}
```

Program 8

8. a. Demonstrate creation of different types of indexes on collection (unique, sparse, compound and multikey indexes)

Create a database **IndexDB** and collection **users** in Mongo IDE.


Create Database

Database Name: IndexDB

Collection Name: users

localhost:27017
My Queries
Performance
Databases
1
2
3
Academics
Academics1
Airbnb
Bookstore
CommerceDB
Project
School
Store
Students
admin
company
config
Time-Series
Time-series collections efficiently store sequences of measurements over a period of time. Learn More
Additional preferences (e.g. Custom collation, Capped, Clustered collections)
Cancel
Create Database

Add the following documents in the **users collection** in MongoDB Shell.

```
db.users.insert({ us
ername: "John",
age: 30,
city:"Chennai",
"interests":["music","garden"],
Description:["good","avg","excellent"],
"hashedField": "hashedValue1",
"location": { "type": "Point", "coordinates": [ -73.97, 40.77 ] },
```

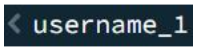
"createdAt": ISODate("2023-01-01T00:00:00Z") })

1. **Unique index:**

   A unique index ensures that the indexed field(s) do not have duplicate values

db.users.createIndex({ "username": 1 }, { unique: true })

Output:



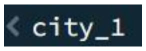
```
< username_1
```

2. **Sparse Index:**

   A sparse index only includes documents that have the indexed field.

   db.users.createIndex({ "city": 1 }, { sparse: true })

Output:



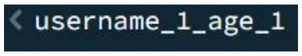
```
< city_1
```

3. **Compound Index:**

   A compound index includes multiple fields within a single index.

   db.users.createIndex({ “username”: 1, “age”: 1 })

   Output:

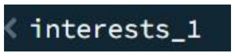


```
< username_1_age_1
```

4. **Multikey Index**

   A multikey index is created on an array field, indexing each value of the array.

   db.users.createIndex({ "interests": 1 })

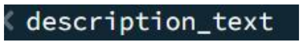


```
< interests_1
```

   Assuming interests is an array field in the user documents.

   db.users.createIndex({ description: "text" })

   Output:



```
< description_text
```

MongoDB provides text indexes to support text search queries on string content. text indexes can include any field whose value is a string or an array of string elements. Remember that you can have only one text index per collection so after creating one if you create another text index, you will get an error.

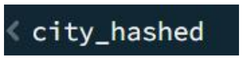
Hashed Index: Indexes where MongoDB hashes the index keys to create a more even distribution of keys.

db. users.createIndex({ city: "hashed" })

Hashed indexes are beneficial for sharding collections in MongoDB.

They distribute data across shards based on the hash value of the indexed field, improving query performance for filtering based on that field

Output:

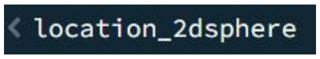


```
< city_hashed
```

Geo-spatial Index: Indexes used for geo-spatial queries.

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

Output:

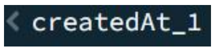


```
< location_2dsphere
```

TTL (Time-To-Live) Index: Indexes that automatically expire documents after a certain amount of time.

db. users.createIndex({ createdAt: 1 }, { expireAfterSeconds: 3600 })

Output:



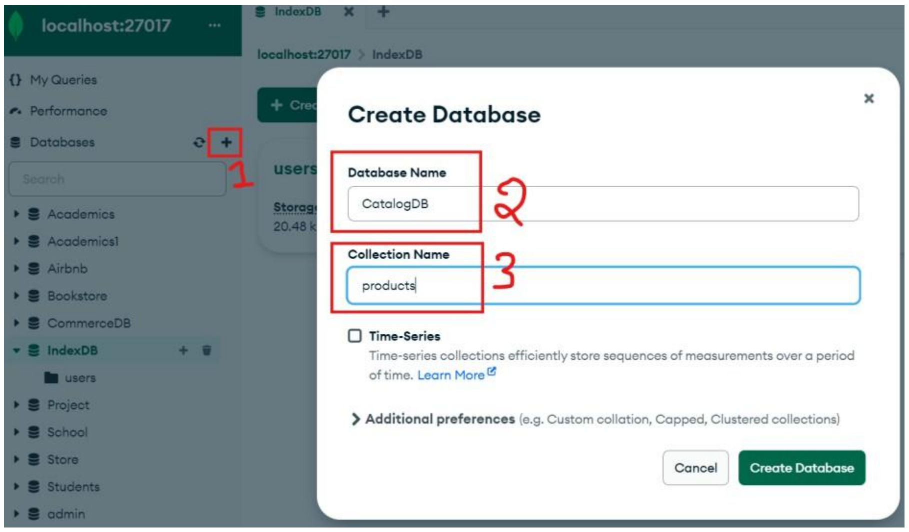
```
< createdAt_1
```

Program 9

9. a. Develop a query to demonstrate Text search using catalog data collection for a given word

b. Develop queries to illustrate excluding documents with certain words and phrases.

Create a database **CatalogDB** and collection **products** in Mongo IDE.

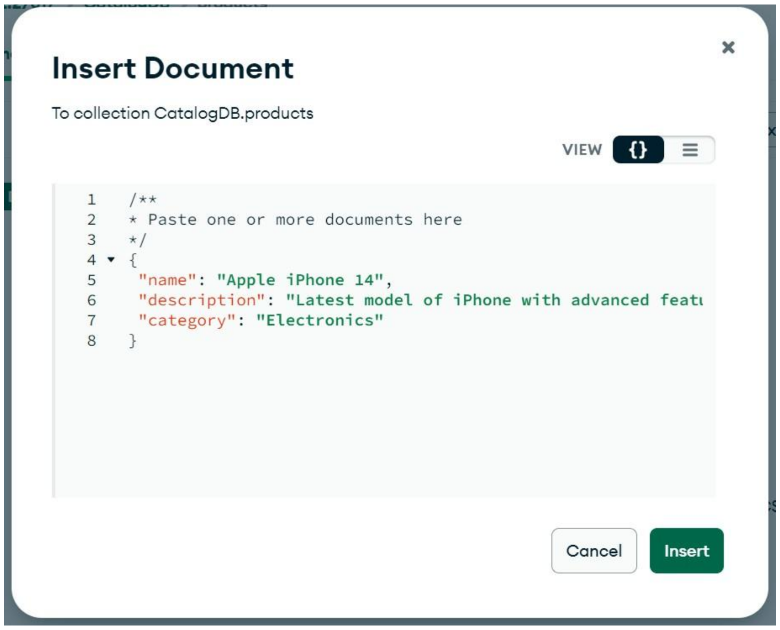

localhost:27017
IndexDB
localhost:27017 > IndexDB
My Queries
Performance
Databases
Search
Academics
Academics1
Airbnb
Bookstore
CommerceDB
IndexDB
users
Project
School
Store
Students
admin
1
2
3
Create Database
Time-Series
Time-series collections efficiently store sequences of measurements over a period of time. Learn More
Additional preferences (e.g. Custom collation, Capped, Clustered collections)
Cancel
Create Database

Database Name: CatalogDB

Collection Name: products

Add the following documents in the **collection products** in MongoDB IDE.

```
{
 "name": "Apple iPhone 14",
 "description": "Latest model of iPhone with advanced features",
 "category": "Electronics"
}
```


Insert Document

To collection CatalogDB.products

```
/**
* Paste one or more documents here
*/
{
 "name": "Apple iPhone 14",
 "description": "Latest model of iPhone with advanced featu
 "category": "Electronics"
}
```

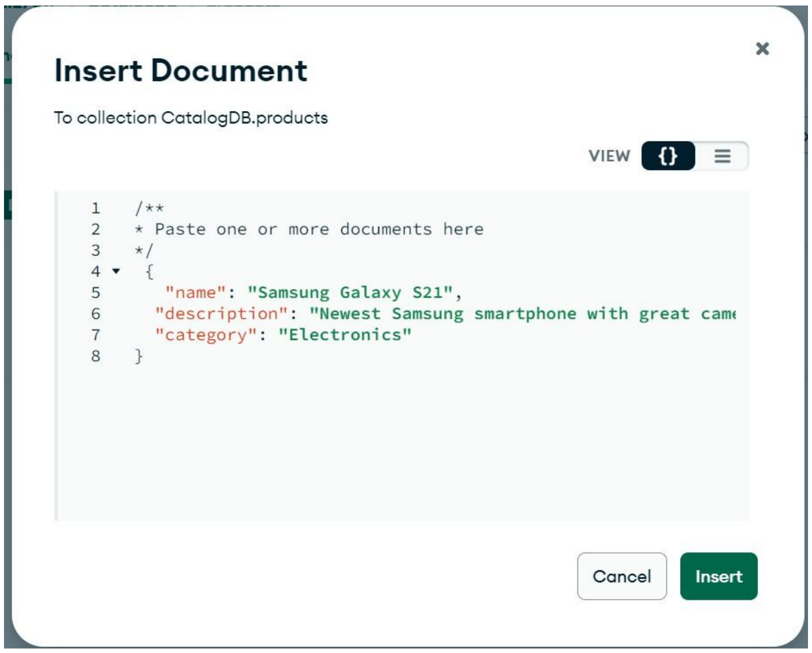
{

"name": "Samsung Galaxy S21",

"description": "Newest Samsung smartphone with great camera",

"category": "Electronics"

}


Insert Document

To collection CatalogDB.products

```
/**
* Paste one or more documents here
*/
 {
   "name": "Samsung Galaxy S21",
  "description": "Newest Samsung smartphone with great came
  "category": "Electronics"
}
```

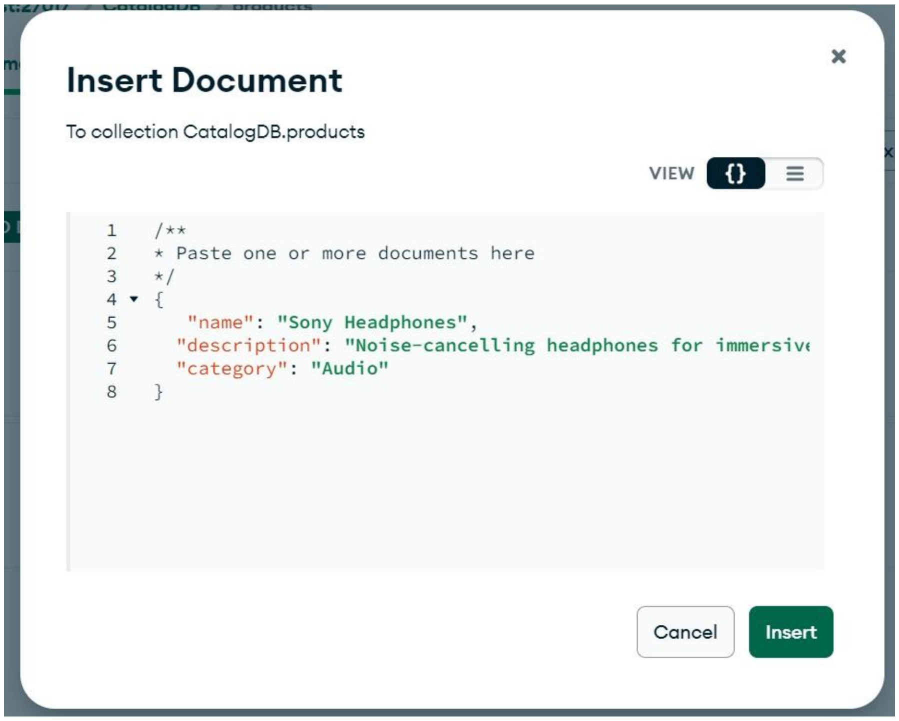
{

"name": "Sony Headphones",

"description": "Noise-cancelling headphones for immersive sound",

"category": "Audio"

}



```
/**
 * Paste one or more documents here
 */
{
   "name": "Sony Headphones",
  "description": "Noise-cancelling headphones for immersive
  "category": "Audio"
}
```

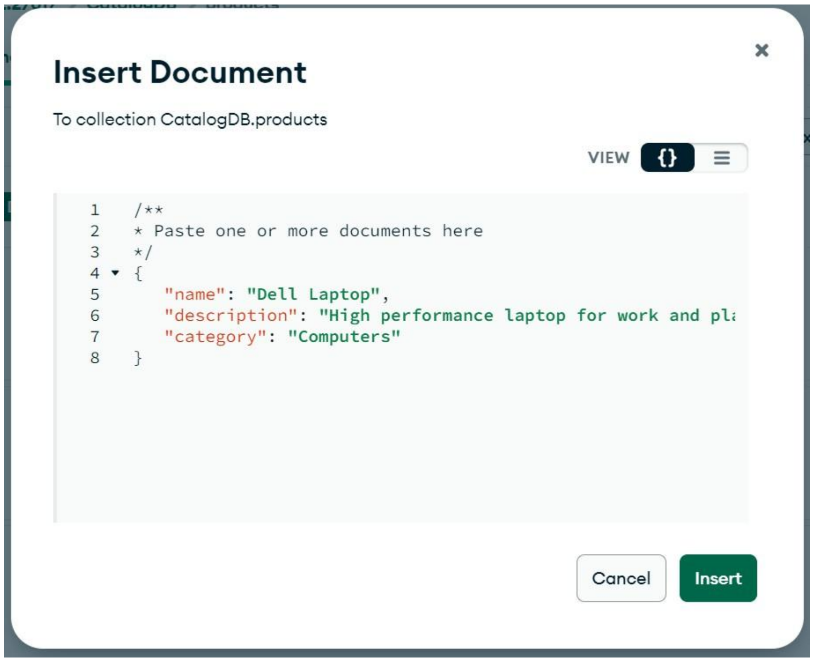
{

"name": "Dell Laptop",

"description": "High performance laptop for work and play", "category":

"Computers"

}



```
/**
* Paste one or more documents here
*/
{
    "name": "Dell Laptop",
    "description": "High performance laptop for work and pl
    "category": "Computers"
}
```

In MongoShell

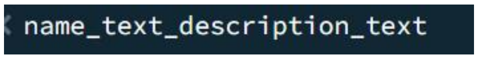
>use CatalogDB

a. 1. Create a Text Index

To enable text search, you need to create a text index on the fields you want to search. Here, we'll create a text index on the name and description fields:

db.products.createIndex({ name: "text", description: "text" })

Output:



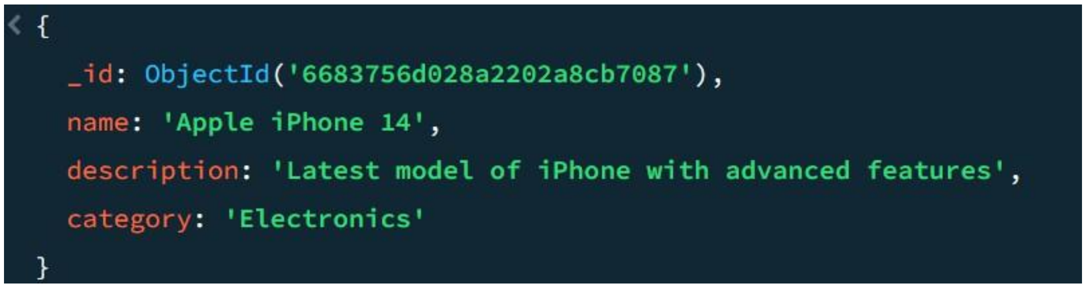
```
name_text_description_text
```

2. **Perform a Text Search**

Now, let's perform a text search. Suppose you want to search for products related to the word "latest":

db.products.find({ $text: { $search: "latest" } })

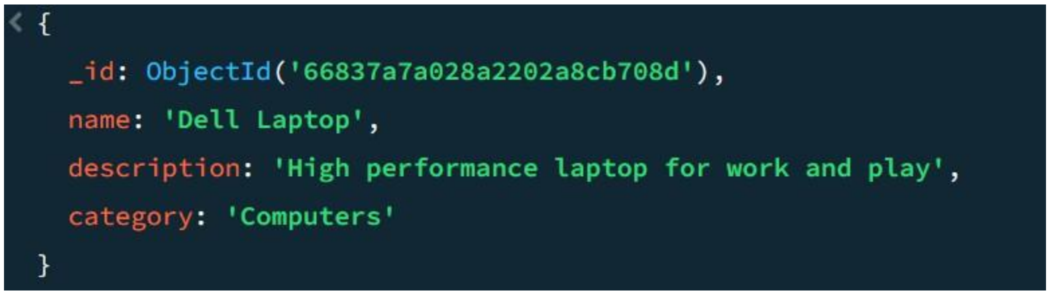
Output:



```
< {
    _id: ObjectId('6683756d028a2202a8cb7087'),
    name: 'Apple iPhone 14',
    description: 'Latest model of iPhone with advanced features',
    category: 'Electronics'
  }
```

db.products.find({ $text: { $search: "High performance" } })

Output:

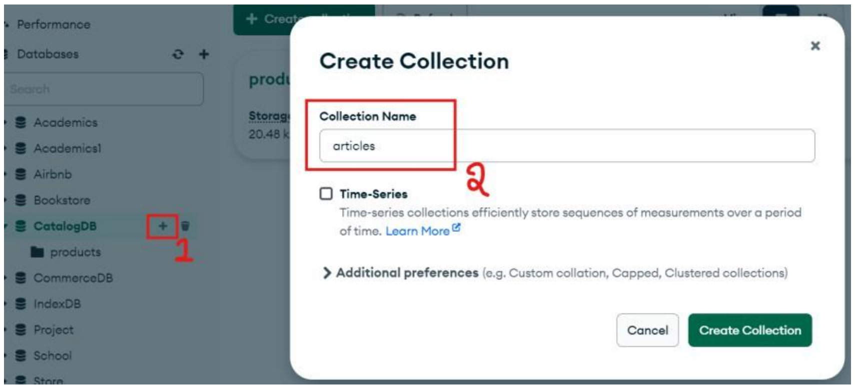


```
< {
    _id: ObjectId('66837a7a028a2202a8cb708d'),
    name: 'Dell Laptop',
    description: 'High performance laptop for work and play',
    category: 'Computers'
  }
```

b. Develop queries to illustrate excluding documents with certain words and phrases.

In MongoDB, you can use the $not operator combined with the $regex operator to exclude documents that contain certain words or phrases. Below are some examples of queries to illustrate this.

Add the following documents in the **collection articles** in MongoDB IDE.



Create Collection

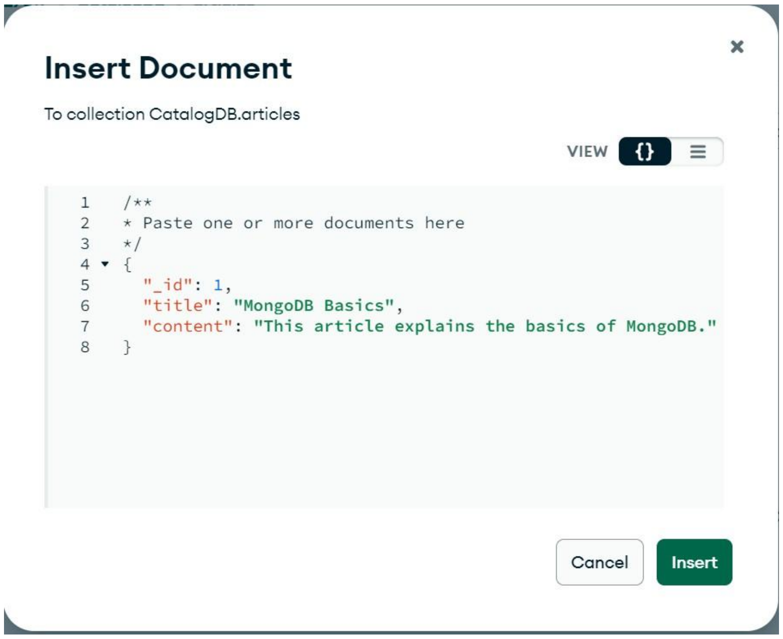
Collection Name: articles

Time-Series

Time-series collections efficiently store sequences of measurements over a period of time. Learn More

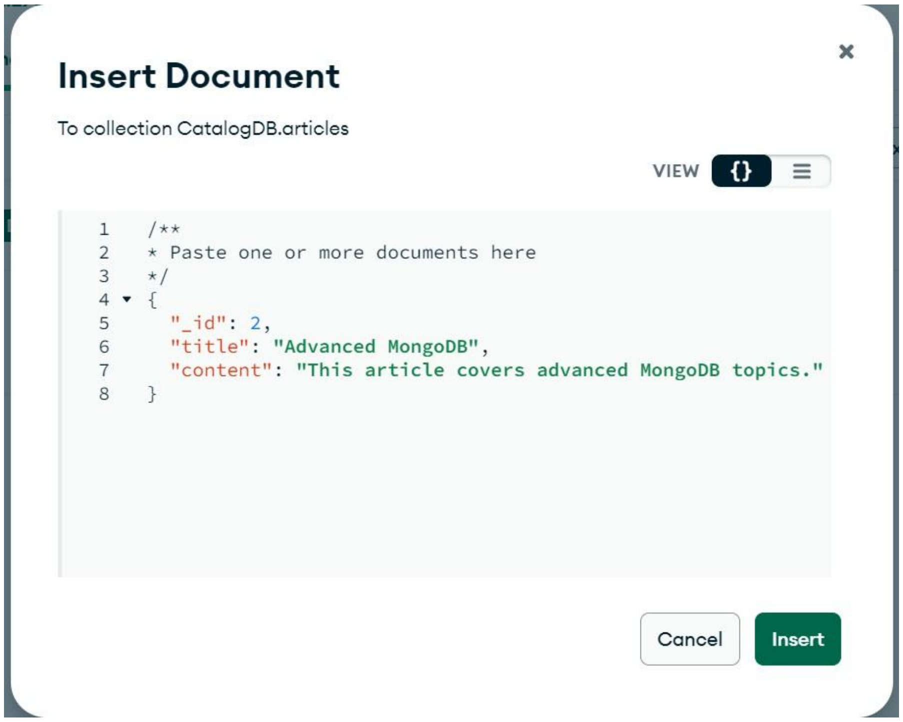
Additional preferences (e.g. Custom collation, Capped, Clustered collections)

```
{
  "_id": 1,
  "title": "MongoDB Basics",

  "content": "This article explains the basics of MongoDB."

}
```



```
/**
* Paste one or more documents here
*/
{
  "_id": 1,
  "title": "MongoDB Basics",
  "content": "This article explains the basics of MongoDB."
}
```

```
{

  "_id": 2,

  "title": "Advanced MongoDB",

  "content": "This article covers advanced MongoDB topics."

}
```



```
/**
* Paste one or more documents here
*/
{
  "_id": 2,
  "title": "Advanced MongoDB",
  "content": "This article covers advanced MongoDB topics."
}
```

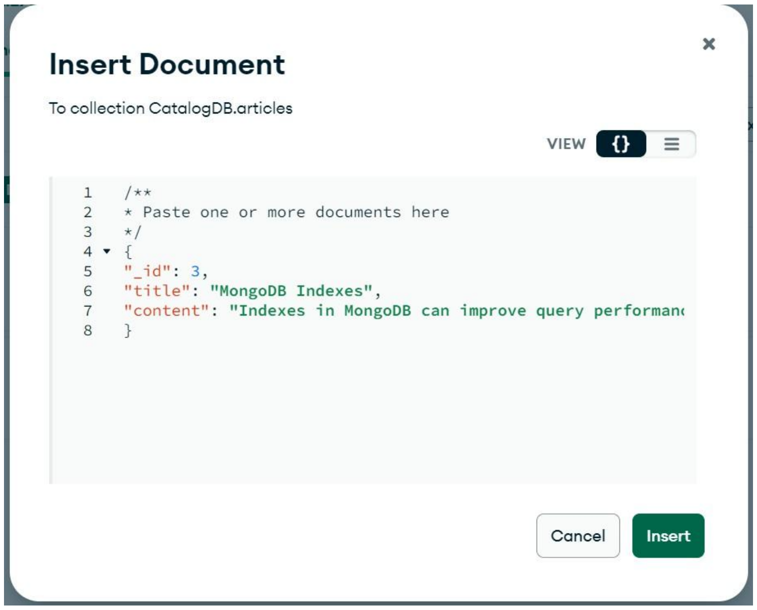
{

"_id": 3,

"title": "MongoDB Indexes",

"content": "Indexes in MongoDB can improve query performance."

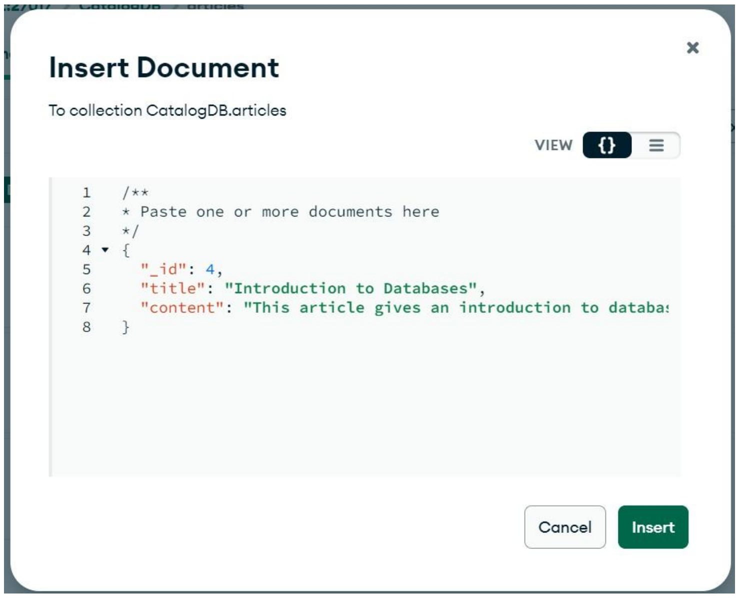
}



```
/**
* Paste one or more documents here
*/
{
"_id": 3,
"title": "MongoDB Indexes",
"content": "Indexes in MongoDB can improve query performanc
}
```

{

```
  "_id": 4,
  "title": "Introduction to Databases",
  "content": "This article gives an introduction to databases in general."
}
```



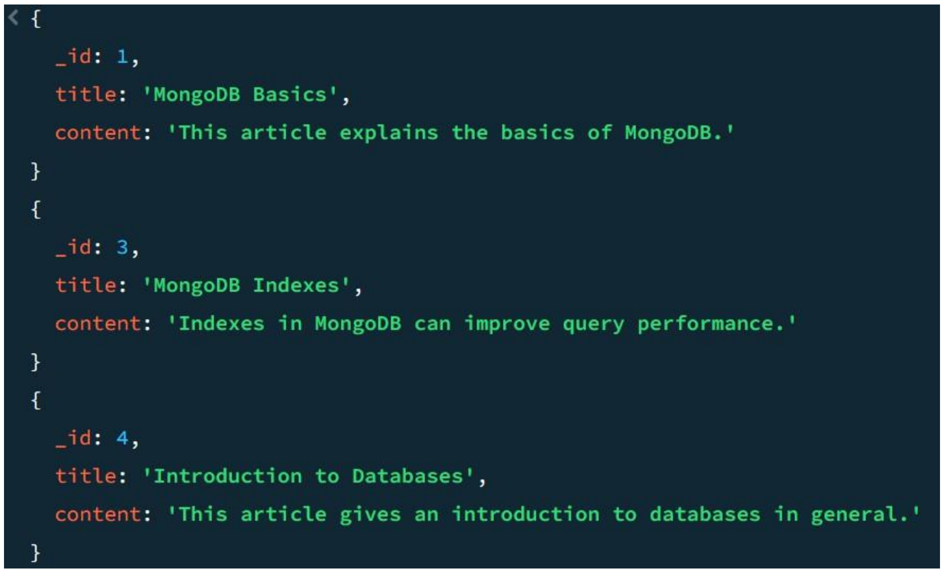
```
/**
 * Paste one or more documents here
 */
{
  "_id": 4,
  "title": "Introduction to Databases",
  "content": "This article gives an introduction to databas
}
```

1. Exclude Documents Containing a Specific Word

To exclude documents that contain the word "advanced" in the `'content'` field:

```
db.articles.find({ "content

  ": {

     $not: /advanced/

  }

})
```

Output:



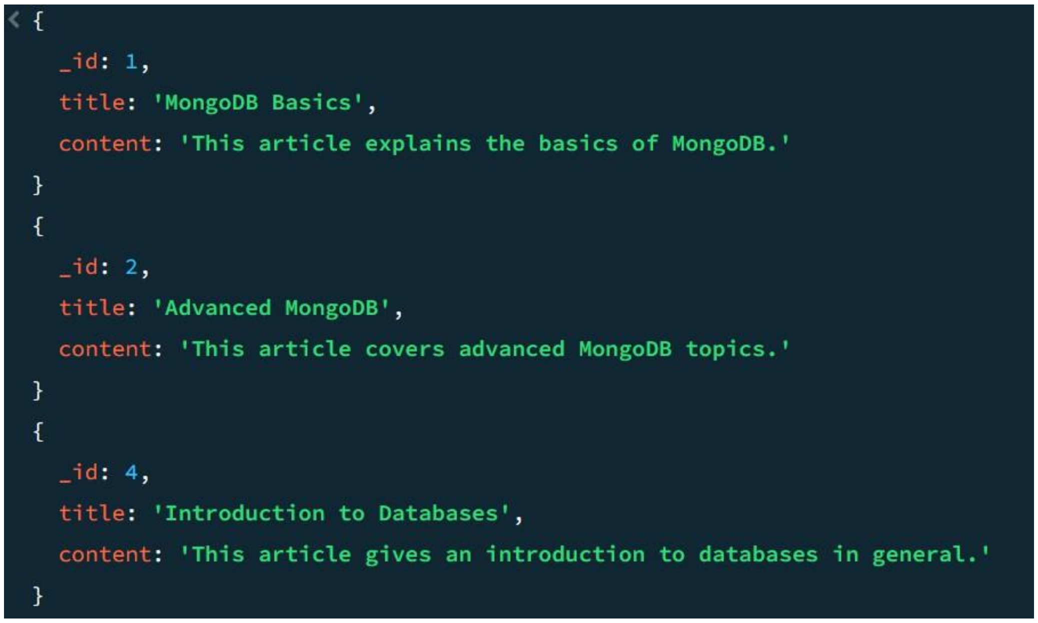
```
< {
    _id: 1,
    title: 'MongoDB Basics',
    content: 'This article explains the basics of MongoDB.'
  }
  {
    _id: 3,
    title: 'MongoDB Indexes',
    content: 'Indexes in MongoDB can improve query performance.'
  }
  {
    _id: 4,
    title: 'Introduction to Databases',
    content: 'This article gives an introduction to databases in general.'
  }
```

2. Exclude Documents Containing Any of Multiple Words

To exclude documents that contain either "improve" or "performance" in the `‘content’` field:

```
db.articles.find({
          "content": {
            $not: /(improve|performance)/
          }
})
```

Output:



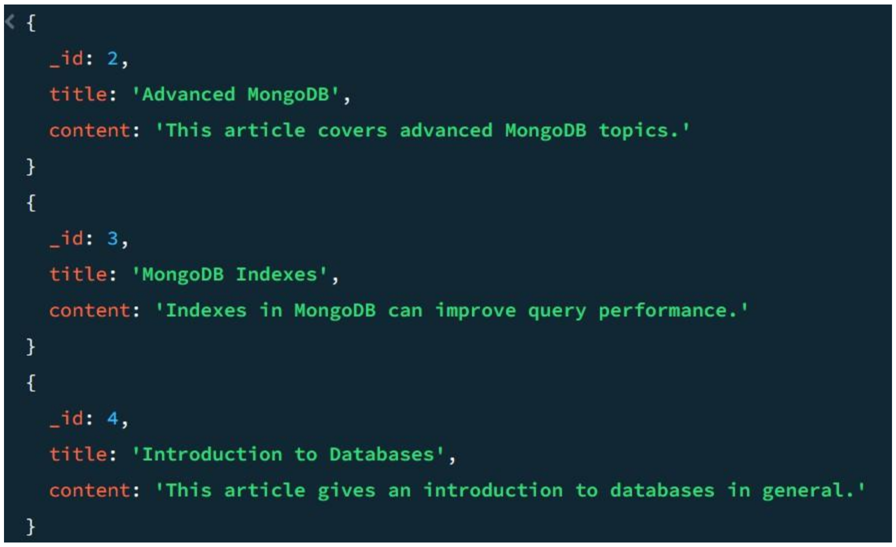
```
< {
    _id: 1,
    title: 'MongoDB Basics',
    content: 'This article explains the basics of MongoDB.'
  }
  {
    _id: 2,
    title: 'Advanced MongoDB',
    content: 'This article covers advanced MongoDB topics.'
  }
  {
    _id: 4,
    title: 'Introduction to Databases',
    content: 'This article gives an introduction to databases in general.'
  }
```

3. Exclude Documents Containing a Specific Phrase

To exclude documents that contain the phrase "MongoDB Basics" in the ‘`title`’ field:

```
db.articles.find({ "title
  ": {
    $not: /MongoDB Basics/
  }
})
```

Output:



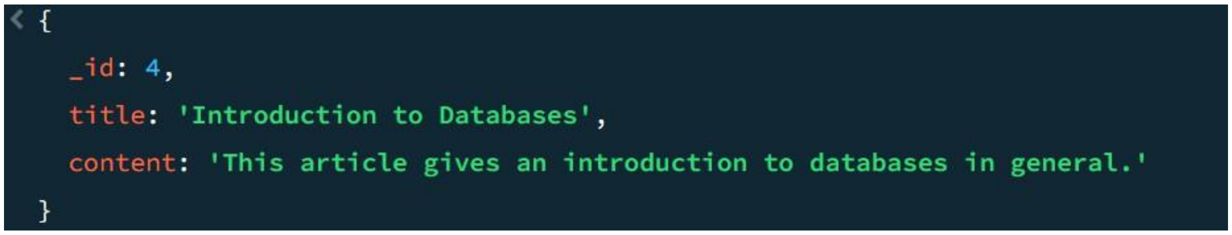
```
{
  _id: 2,
  title: 'Advanced MongoDB',
  content: 'This article covers advanced MongoDB topics.'
}
{
  _id: 3,
  title: 'MongoDB Indexes',
  content: 'Indexes in MongoDB can improve query performance.'
}
{
  _id: 4,
  title: 'Introduction to Databases',
  content: 'This article gives an introduction to databases in general.'
}
```

4. Exclude Documents Based on Multiple Fields

To exclude documents that contain "MongoDB" in the ‘`title`’ or "advanced" in the ‘`content`’:

```
db.articles.find({
   $and: [
      { "title": { $not: /MongoDB/ } },
      { "content": { $not: /advanced/ } }
   ]
})
```

Output:

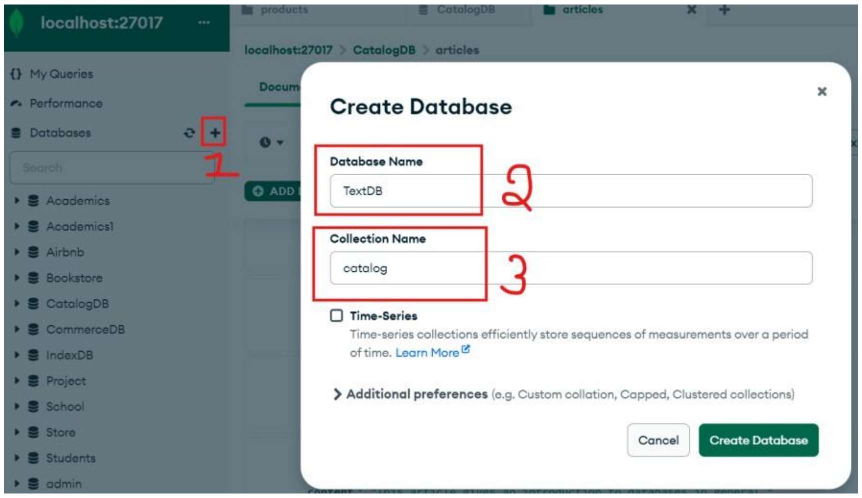


```
{
  _id: 4,
  title: 'Introduction to Databases',
  content: 'This article gives an introduction to databases in general.'
}
```

Program 10

Develop an aggregation pipeline to illustrate Text search on Catalog data collection.

Create a database **TextDB** and collection **catalog** in Mongo IDE.



Database Name: TextDB

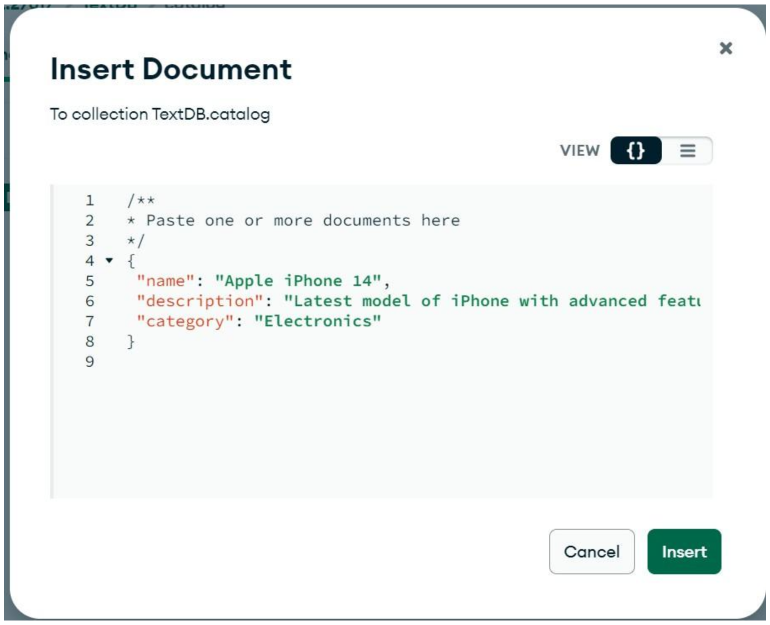
Collection Name: catalog

Time-series collections efficiently store sequences of measurements over a period of time. Learn More

(e.g. Custom collation, Capped, Clustered collections)

Add the following documents in the **catalog collection** in MongoDB Shell.

```
    {
 "name": "Apple iPhone 14",
 "description": "Latest model of iPhone with advanced features",
 "category": "Electronics"
}
```



```
/**
* Paste one or more documents here
*/
{
 "name": "Apple iPhone 14",
 "description": "Latest model of iPhone with advanced featu
 "category": "Electronics"
}
```

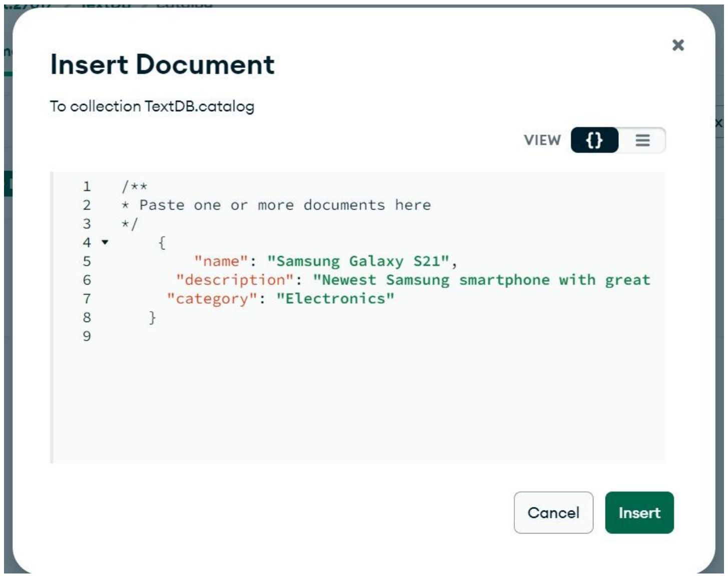
{

"name": "Samsung Galaxy S21",

"description": "Newest Samsung smartphone with great camera",

"category": "Electronics"

}



```
/**
* Paste one or more documents here
*/
    {
        "name": "Samsung Galaxy S21",
      "description": "Newest Samsung smartphone with great
     "category": "Electronics"
    }
```

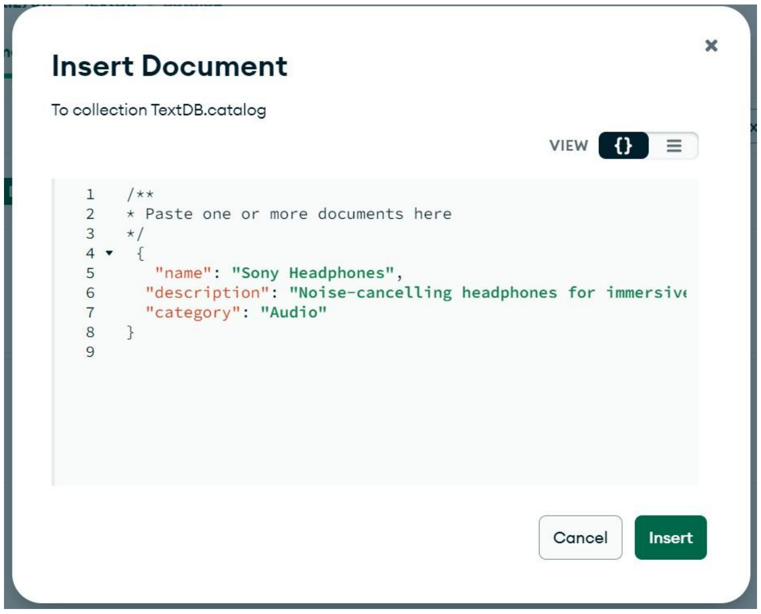
{

"name": "Sony Headphones",

"description": "Noise-cancelling headphones for immersive sound",

"category": "Audio"

}



```
/**
* Paste one or more documents here
*/
 {
   "name": "Sony Headphones",
  "description": "Noise-cancelling headphones for immersive
  "category": "Audio"
}
```

In MongoShell

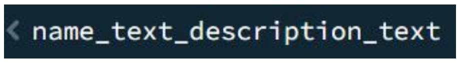
>use TextDB

Create a Text Index

First, create a text index on the 'name' and 'description' fields.

db.catalog.createIndex({ name: "text", description: "text" });

Output:



```
< name_text_description_text
```

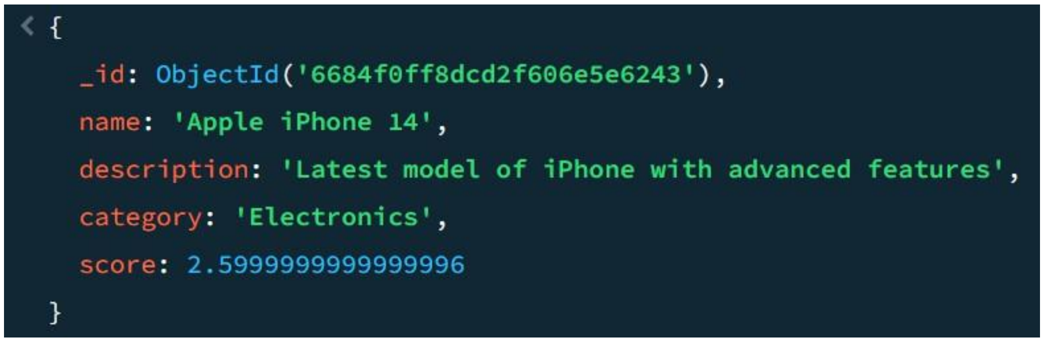
Define the Aggregation Pipeline

Now, create an aggregation pipeline to perform the text search and process the results. Below is an example pipeline:

db.catalog.aggregate([

```
{ $match: {
        $text: { $search: " Apple iPhone 14" } }
},
{$project:
    { name:
    1,
    description: 1,
    category: 1,
    score: { $meta: "textScore" }
  }
},
{
  $sort: {
    score: { $meta: "textScore" }
  } },
{
  $limit: 10
}]);
```

Output:



```
< {
    _id: ObjectId('6684f0ff8dcd2f606e5e6243'),
    name: 'Apple iPhone 14',
    description: 'Latest model of iPhone with advanced features',
    category: 'Electronics',
    score: 2.5999999999999996
  }
```