MONGODB

In this , we will explore the fundamental concepts of database operations, including how to effectively use the **"WHERE"** clause, logical operators like **"AND"** and **"OR"**, and the **CRUD** (Create, Read, Update, Delete) actions that form the foundation of database management.

Clause

WHERE

In the world of database management, the WHERE clause plays a crucial role in filtering and retrieving specific data from a collection or table. The WHERE clause allows you to apply a condition or set of conditions to narrow down the results, providing you with a subset of the data that meets your criteria.

Querying Documents

The WHERE clause in MongoDB allows you to filter and select specific documents from a collection based on one or more criteria.

Comparison Operators

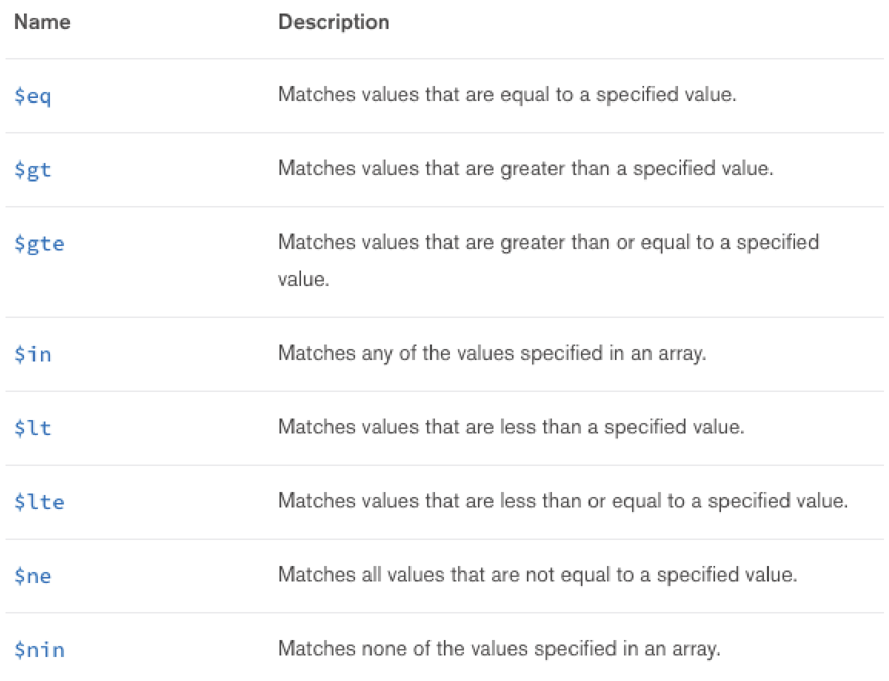
MongoDB supports various comparison operators like **$eq, $gt, $lt, $gte,** and **$lte** to perform advanced queries.

Logical Operators

The **$and, $or,** and **$not** operators can be used to combine multiple conditions for more complex queries.

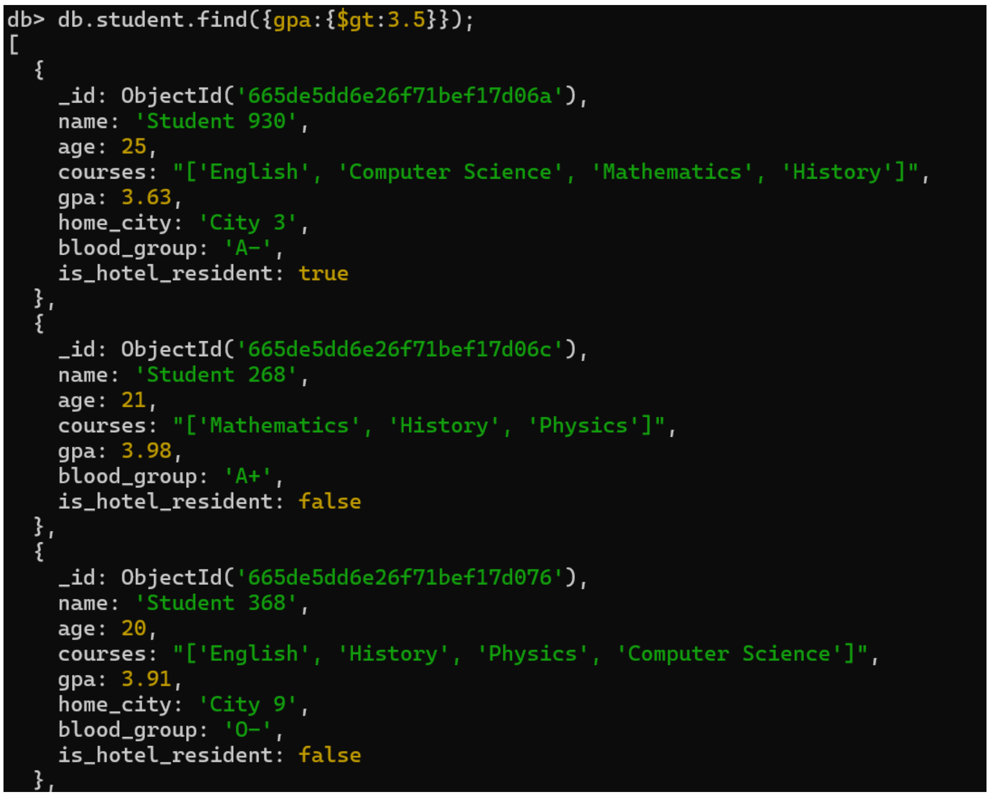
Example:

Imagine you have a collection of customer information, and you need to find all the customers who live in a particular city. The WHERE clause would allow you to specify the city as the condition, and the database would return only the records that match that criteria. This is an essential feature for working with large datasets, as it enables you to focus on the specific information you need, rather than sifting through the entire dataset.



Name	Description
`$eq`	Matches values that are equal to a specified value.
`$gt`	Matches values that are greater than a specified value.
`$gte`	Matches values that are greater than or equal to a specified value.
`$in`	Matches any of the values specified in an array.
`$lt`	Matches values that are less than a specified value.
`$lte`	Matches values that are less than or equal to a specified value.
`$ne`	Matches all values that are not equal to a specified value.
`$nin`	Matches none of the values specified in an array.

Syntax:



```
db> db.student.find({gpa:{$gt:3.5}});
[
  {
    _id: ObjectId('665de5dd6e26f71bef17d06a'),
    name: 'Student 930',
    age: 25,
    courses: "['English', 'Computer Science', 'Mathematics', 'History']",
    gpa: 3.63,
    home_city: 'City 3',
    blood_group: 'A-',
    is_hotel_resident: true
  },
  {
    _id: ObjectId('665de5dd6e26f71bef17d06c'),
    name: 'Student 268',
    age: 21,
    courses: "['Mathematics', 'History', 'Physics']",
    gpa: 3.98,
    blood_group: 'A+',
    is_hotel_resident: false
  },
  {
    _id: ObjectId('665de5dd6e26f71bef17d076'),
    name: 'Student 368',
    age: 20,
    courses: "['English', 'History', 'Physics', 'Computer Science']",
    gpa: 3.91,
    home_city: 'City 9',
    blood_group: 'O-',
    is_hotel_resident: false
  },
```

AND

When working with a collection of data, there may be times when you need to filter that collection based on multiple conditions. This is where the AND operator comes into play. The AND operator allows you to combine multiple conditions, ensuring that the results returned match all of the specified criteria.

Combining Conditions

The **$and** operator in MongoDB allows you to combine multiple conditions in a single query, ensuring that all the specified criteria are met.

Efficient Indexing

Using **$and** queries can leverage MongoDB's indexing capabilities to provide fast and efficient data retrieval.

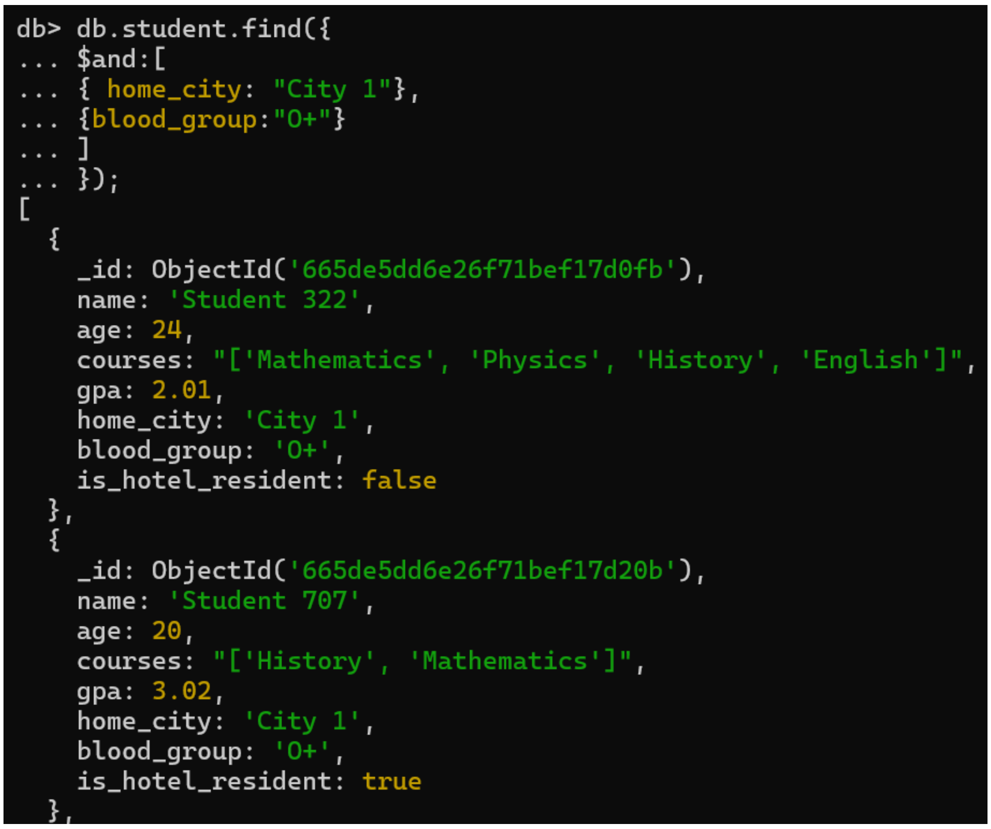
Complex Filtering

The **$and** operator is particularly useful when you need to apply multiple filters to your data, allowing for more precise and targeted queries.

Example

let's say you have a collection of customer records and you want to find all customers who are located in New York and have a balance greater than $1,000. By using the AND operator, you can easily retrieve this subset of customers that meet both conditions. This can be incredibly useful for narrowing down your data to the specific information you need, helping you make more informed decisions and take targeted actions.

The AND operator is a powerful tool in your data analysis , allowing you to slice and dice your data in meaningful ways.



```
db> db.student.find({
... $and:[
... { home_city: "City 1"},
... {blood_group:"O+"}
... ]
... });
[
  {
    _id: ObjectId('665de5dd6e26f71bef17d0fb'),
    name: 'Student 322',
    age: 24,
    courses: "['Mathematics', 'Physics', 'History', 'English']",
    gpa: 2.01,
    home_city: 'City 1',
    blood_group: 'O+',
    is_hotel_resident: false
  },
  {
    _id: ObjectId('665de5dd6e26f71bef17d20b'),
    name: 'Student 707',
    age: 20,
    courses: "['History', 'Mathematics']",
    gpa: 3.02,
    home_city: 'City 1',
    blood_group: 'O+',
    is_hotel_resident: true
  },
```

OR

When dealing with a collection of data, there may be times when you need to filter the results based on multiple conditions, but only one of those conditions needs to be met. This is where the OR operator comes into play.

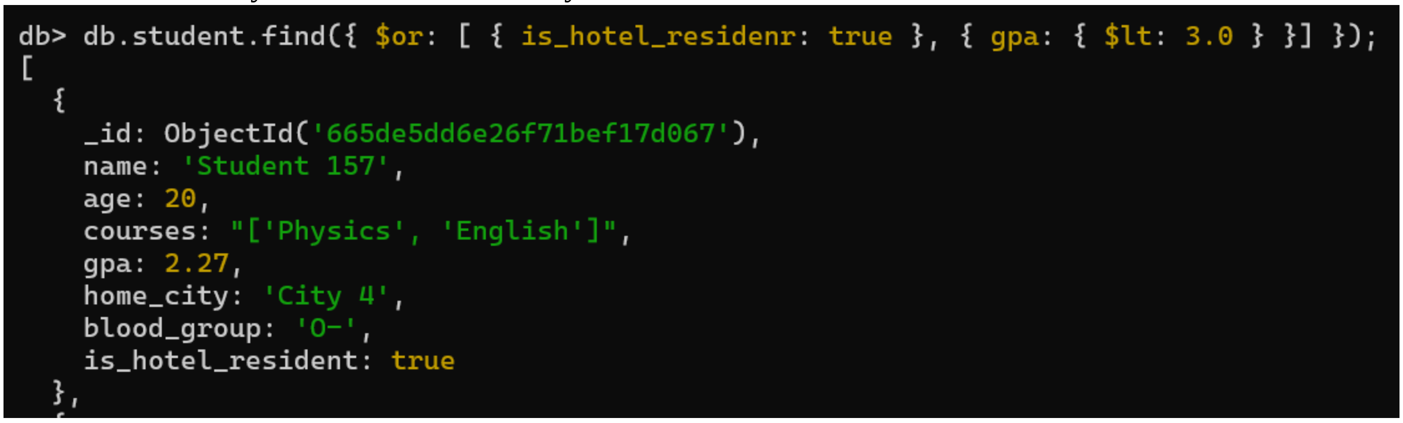
The OR operator allows you to specify multiple criteria, and as long as at least one of those criteria is satisfied, the record will be included in the filtered results. This is in contrast to the AND operator, which requires all specified criteria to be met for a record to be included.

<u>Example</u>

You might want to retrieve all customers who live in either New York or Los Angeles. In this case, you would use an OR condition to search for records where the city is equal to "New York" OR "Los Angeles".

1. Another use case could be finding all products that are either in stock or on sale. You would use an OR condition to search for records where the inventory_status is "in_stock" OR the sale_price is less than the regular price.

2. The OR operator is a powerful tool for creating more complex and flexible filtering criteria, allowing you to retrieve the data you need more efficiently.



```
db> db.student.find({ $or: [ { is_hotel_residenr: true }, { gpa: { $lt: 3.0 } }] });
[
  {
    _id: ObjectId('665de5dd6e26f71bef17d067'),
    name: 'Student 157',
    age: 20,
    courses: "['Physics', 'English']",
    gpa: 2.27,
    home_city: 'City 4',
    blood_group: 'O-',
    is_hotel_resident: true
  },
```

CURD:

This is applicable for a Collection (Table) or a Document (Row)

C- Create/ Insert

U- Update

R- Remove

D- Delete

Create/Insert The "Create" or "Insert" operation in CRUD is used to add new data to a collection or table. This could be creating a new user account, adding a product to an inventory, or inserting a new row of data into a database. The "Create" operation is essential for expanding and populating a data store with new information as needed.	**Remove** The "Remove" operation allows you to retrieve data from a collection or table. This could involve querying the database to find all users with a certain email address, or pulling a specific product record.
Update The "Update" operation is used to modify existing data in a collection or table. This could include changing a user's contact information, updating a product's price, or editing a record in the database. The "Update" function is essential for maintaining accurate and up-to-date data in your system.	**Delete** The "Delete" operation allows you to remove data from a collection or table. This could involve deleting a user account, removing a product from inventory, or erasing a specific record from the database. The "Delete" function is important for cleaning up and managing the data in your system as needed.

INSERT:

The **insert() method** in MongoDB inserts documents in the MongoDB collection. This method is also used to create a new **collection** by inserting documents.

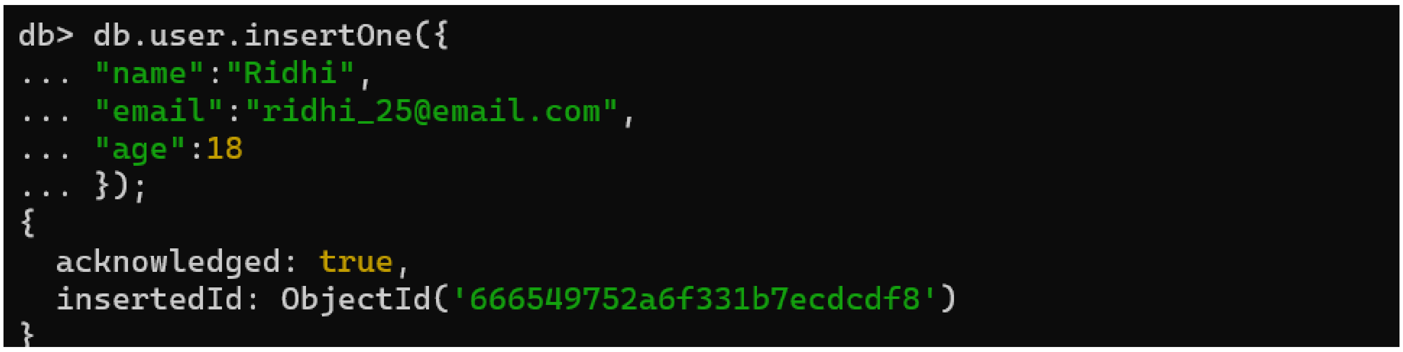
Syntax

```
db.Collection_name.insertOne(
<document or [document1, document2,…..]>,
{
   writeConcern: <document>,
   ordered: <boolean>
})
```

For example, let’s say we want to create a new document to represent a user. We could define the structure of the document like this

```
{
 "name": "Ridhi",
 "email": "ridhi_25@gmail.com",
 "age": 18
}
```

To insert this document into a MongoDB collection, we would use the insertOne() method. Here's an example:

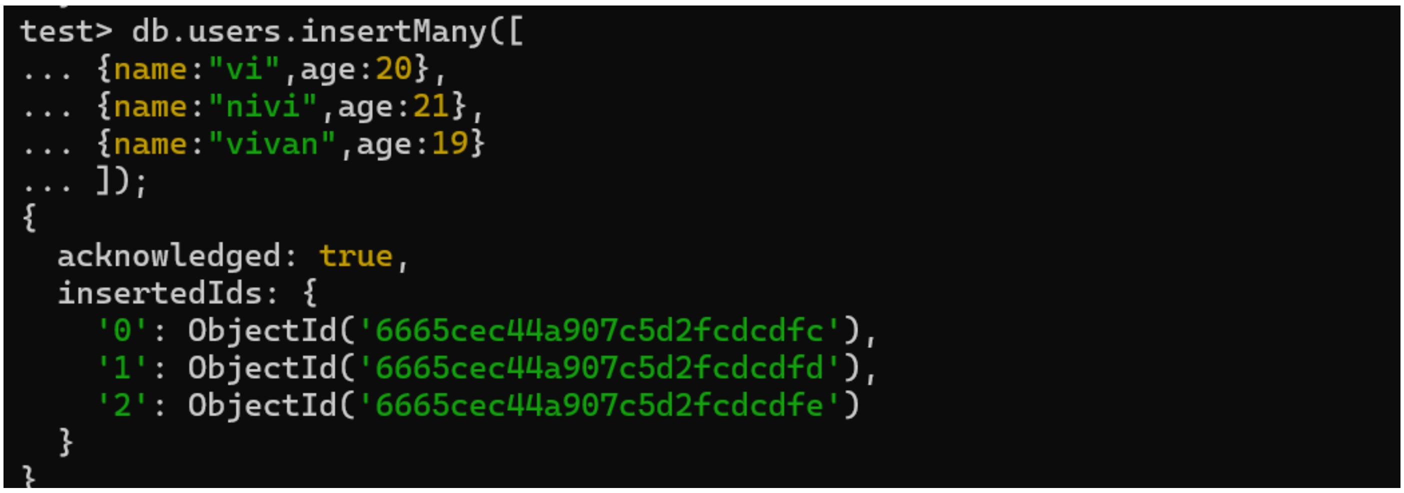

```
db> db.user.insertOne({
... "name":"Ridhi",
... "email":"ridhi_25@email.com",
... "age":18
... });
{
  acknowledged: true,
  insertedId: ObjectId('666549752a6f331b7ecdcdf8')
}
```

insertMany():

The **insertMany()** method inserts one or more documents in the collection. It takes array of documents to insert in the collection.

Syntax:

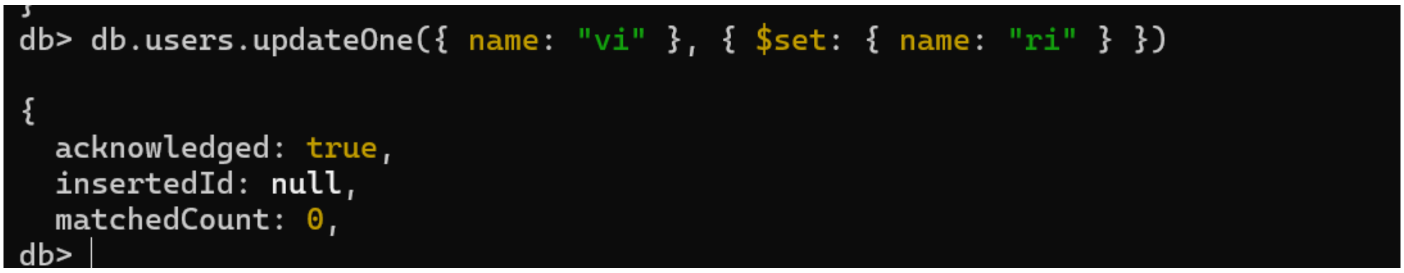
```
db.Collection_name.insertMany(
[<document 1>, <document 2>, ...],
{
   writeConcern: <document>,
   ordered: <boolean>
})
```


```
test> db.users.insertMany([
... {name:"vi",age:20},
... {name:"nivi",age:21},
... {name:"vivan",age:19}
... ]);
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('6665cec44a907c5d2fcdcdfc'),
    '1': ObjectId('6665cec44a907c5d2fcdcdfd'),
    '2': ObjectId('6665cec44a907c5d2fcdcdfe')
  }
}
```

UPDATE:

The **updateOne()** method in MongoDB updates the first matched document within the collection based on the given query.
MongoDB updateOne() Method Syntax

```
db.collection.updateOne(<filter>, <update>, {
  upsert: <boolean>,
  writeConcern: <document>,
  collation: <document>,
  arrayFilters: [<filterdocument1>, ...],
  hint: <document|string> // Available starting in MongoDB 4.2.1
})
```



```
db> db.users.updateOne({ name: "vi" }, { $set: { name: "ri" } })

{
  acknowledged: true,
  insertedId: null,
  matchedCount: 0,
db> 
```

updateMany():

The **updateMany()** method updates all the documents in MongoDB collections that match the given query. When you update your document, the value of the _id field remains unchanged. This method can also add new fields in the document. Specify an empty document(**{}**) in the selection criteria to update all collection documents.

Syntax:

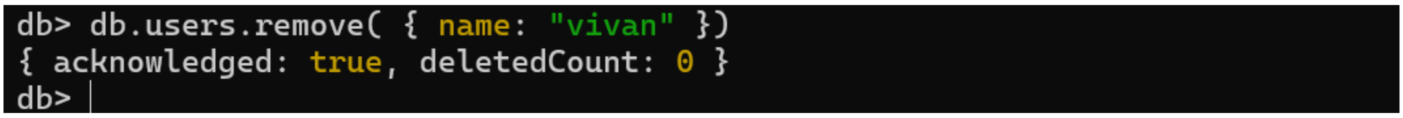
```
db.Collection_name.updateMany({Selection_Criteria},{$set:{Update_Data}},
{
   upsert: <boolean>,
   multi: <boolean>,
   writeConcern: <document>,
   collation: <document>,
   arrayFilters: [<filterdocument1>, ... ],
   hint: <document|string>
})
```

REMOVE():

The **remove()** method removes documents from the database. It can remove one or all documents from the collection that matches the given query expression. If you pass an empty document(**{}**) in this method, then it will remove all documents from the specified collection. It takes four parameters and returns an object that contains the status of the operation.

Syntax:

```
db.Collection_name.remove(

<matching_criteria>,

{

   justOne: <boolean>,

   writeConcern: <document>,

   collation: <document>

})
```

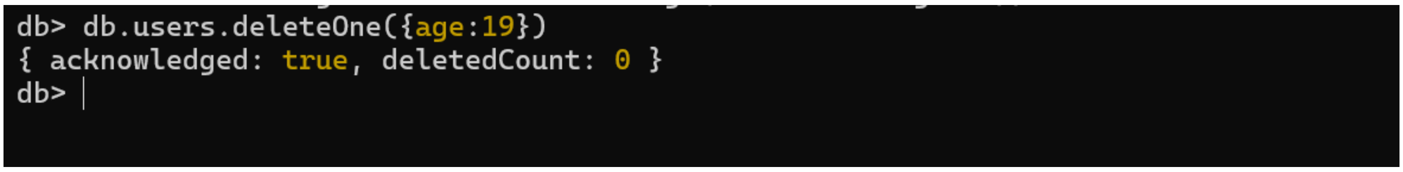


```
db> db.users.remove( { name: "vivan" })
{ acknowledged: true, deletedCount: 0 }
db> 
```

DELETE():

The **deleteOne()** method in MongoDB deletes the first document from the collection that matches the given selection criteria. It will delete/remove a single document from the collection.
If you use this method in the capped collection, then this method will give a **WriteError exception**, so to delete documents from the capped collection use the **drop() method**.
If you use this method in the sharded collection, then the query expression passed in this method must contain the sharded key/_id field, if not then this method will give an error. You can also use this method in multi-document transactions.
Syntax

```
db.Collection_name.deleteOne(
selection_criteria:<document>,
{
  writeConcern:<document>,
  collation:<document>,
 hint:<document/string>
})
```



```
db> db.users.deleteOne({age:19})
{ acknowledged: true, deletedCount: 0 }
db> 
```