4.Projection, Limit & Selectors

Projection:

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

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

But we only want to display the name and the age of the employee rather than displaying whole details. Now, here we use projection to display the name and age of the employee.

One can use projection with db.collection.find() method. In this method, the second parameter is the projection parameter, which is used to specify which fields are returned in the matching documents.

Syntax:

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

If the value of the field is set to 1 or true, then it means the field will include in the return document.

If the value of the field is set to 0 or false, then it means the field will not include in the return document.

You are allowed to use projection operators, but find() method does not support following projection operators, i.e., $, $elemMatch, $slice, and $meta.

There is no need to set _id field to 1 to return _id field, the find() method always return _id unless you set a _id field to 0.

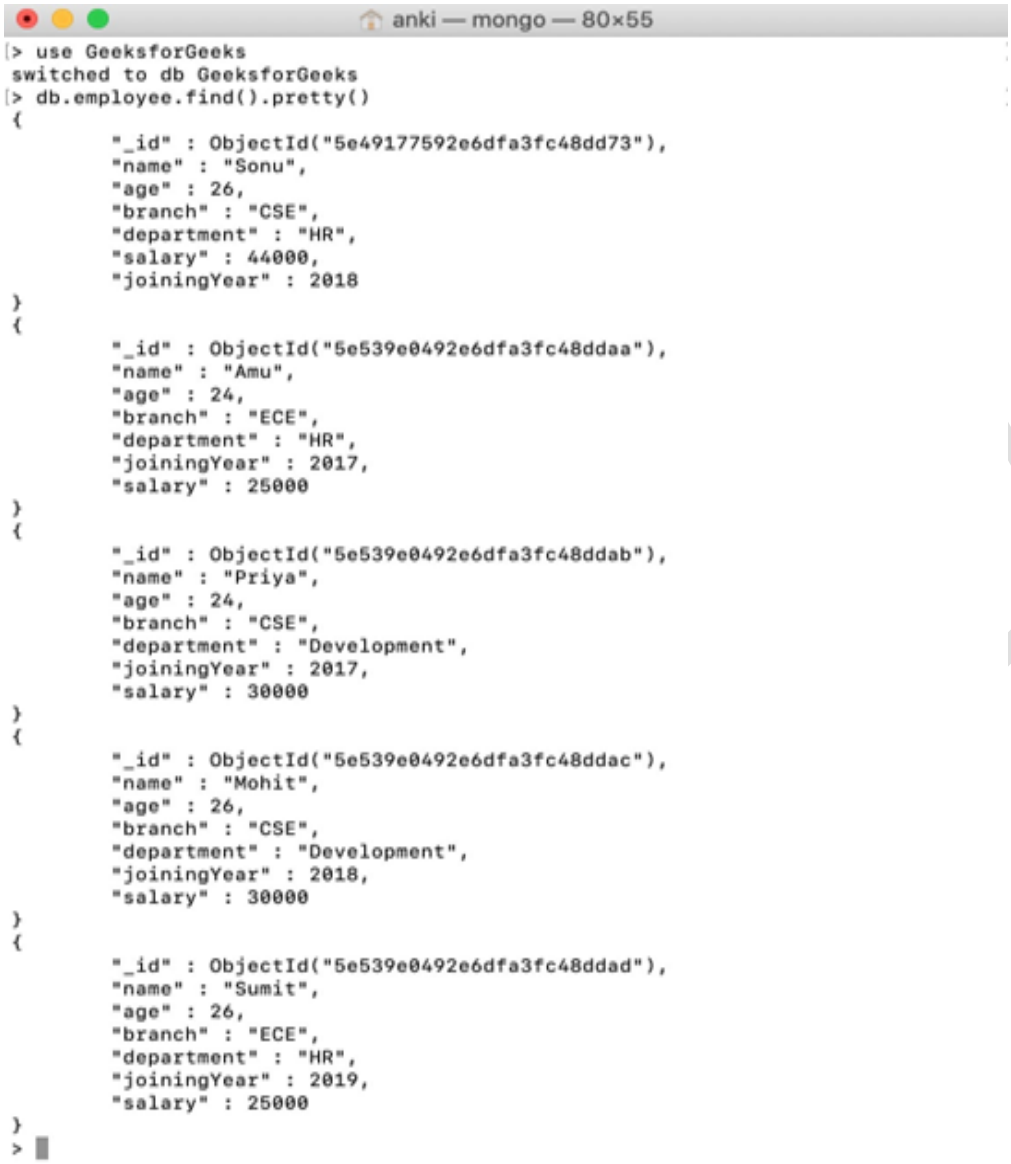
Examples:

In the following examples, we are working with:

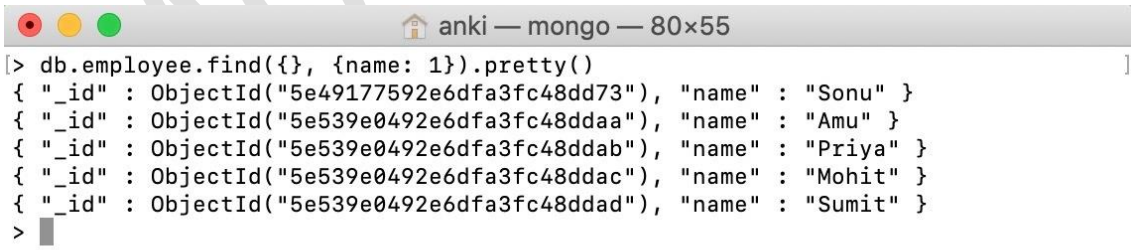
Database: db

Collection: employee

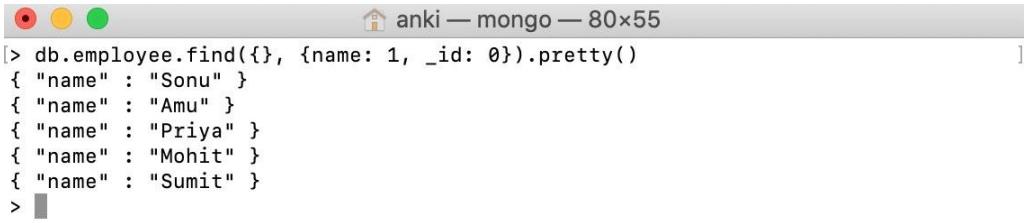
Document: five documents that contain the details of the employees in the form of yield-value pairs



Displaying the names of the employees –



Displaying the names of the employees without the _id field –



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

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

Limit:

In MongoDB, the **limit()** method limits the number of records or documents that you want. It basically defines the max limit of records/documents that you want. Or in other words, this method uses on cursor to specify the maximum number of documents/ records the cursor will return. We can use this method after the find() method and find() will give you all the records or documents in the collection. You can also use some conditions inside the find to give you the result that you want.

- In this method, we only pass numeric values.
- This method is undefined for values which is less than -2^{31} and greater than 2^{31}.
- Passing 0 in this method(limit(0)) is equivalent to no limit.

Syntax:

```
Cursor.limit()
```

Examples:

In the following examples, we are working with:

Database: db

Collections: gfg

Document: Eight documents contains the contect

Limit two documents:

```
db.gfg.find().limit(2)
```

Here, we only want the first two documents in the result. So, we pass 2 in the limit method.

```
> use geeksforgeeks
switched to db geeksforgeeks
> db.gfg.find().limit(2)
{ "_id" : ObjectId("6005d3158438681f01c53e7f"), "content" : "Data Structure" }
{ "_id" : ObjectId("6005d3258438681f01c53e80"), "content" : "Algorithims" }
>
```

Limit only two documents that match the given condition

```
db.gfg.find({“content”:}).limit(2)
```

Here, we only want the two documents that satisfy the given condition, i.e., {“content”:/c/i}) in the find() method. Here, content is key were we will check whether it contains ‘c’ character in the string or not. /c/ denotes that we are looking for strings that contain this ‘c’ character and in the end of /c/i, i denotes that it is case-insensitive.

```
> use geeksforgeeks
switched to db geeksforgeeks
> db.gfg.find().limit(2)
{ "_id" : ObjectId("6005d3158438681f01c53e7f"), "content" : "Data Structure" }
{ "_id" : ObjectId("6005d3258438681f01c53e80"), "content" : "Algorithims" }
>
```

Limit only three documents that match the given condition

```
db.gfg.find({“content”:/c/i}).limit(3)
```

Here, we only want the three documents that satisfy the given condition, i.e., {“content”:/c/i}) in the find() method. Here, content is key were we will check whether it contains *‘c’* character in the string or not. */c/* denotes that we are looking for strings that contain this ‘c’ character and in the end of */c/i,* i denotes that it is case-insensitive.

```
> use geeksforgeeks
switched to db geeksforgeeks
> db.gfg.find({"content":/c/i}).limit(3)
{ "_id" : ObjectId("6005d3158438681f01c53e7f"), "content" : "Data Structure" }
{ "_id" : ObjectId("6009c642df8008388bd7646a"), "content" : "Competitve Programming" }
{ "_id" : ObjectId("6009c8e4df8008388bd7646c"), "content" : "coding questions" }
>
```

Selectors:

Use the Select your language drop-down menu in the upper-right to set the language of the following examples or select MongoDB Compass.

This page provides examples of query operations using the Collection.find() method in the MongoDB Node.js Driver.

The examples on this page use the `inventory` collection. Connect to a test database in your MongoDB instance then create the `inventory` collection:

```
await db.collection('inventory').insertMany([
  { item: 'journal', qty: 25, size: { h: 14, w: 21, uom: 'cm' }, status: 'A'
  },
  {
    item: 'notebook', qty: 50, size: { h: 8.5, w: 11, uom: 'in' }, status: 'A'
  },
  {
    item: 'paper', qty: 100, size: { h: 8.5, w: 11, uom: 'in' }, status: 'D'
  },
  {
    item: 'planner', qty: 75, size: { h: 22.85, w: 30, uom: 'cm' }, status: 'D'
  },
  {
    item: 'postcard', qty: 45, size: { h: 10, w: 15.25, uom: 'cm' }, status: 'A'
  }
]);
```

Select All Documents in a Collection

To select all documents in the collection, pass an empty document as the query filter parameter to the find method. The query filter parameter determines the select criteria:

```
const cursor = db.collection('inventory').find({});
```

This operation uses a filter predicate of `{}`, which corresponds to the following SQL statement:

```
SELECT * FROM inventory
```

To see supported options for the `find()` method, see find().

Specify Equality Condition

To specify equality conditions, use `<field>:<value>` expressions in the query filter document:

```
{ <field1>: <value1>, ... }
```

The following example selects from the inventory collection all documents where the status equals "D":

```
const cursor = db.collection('inventory').find({ status: 'D' });
```

This operation uses a filter predicate of { status: "D" }, which corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "D"
```

Note:

The MongoDB Compass query bar autocompletes the current query based on the keys in your collection's documents, including keys in embedded sub-documents.

Specify Conditions Using Query Operators

A query filter document can use the query operators to specify conditions in the following form:

```
{ <field1>: { <operator1>: <value1> }, ... }
```

The following example retrieves all documents from the inventory collection where status equals either "A" or "D":

```
const cursor = db.collection('inventory').find({
  status: { $in: ['A', 'D'] }
});
```

Note:

Although you can express this query using the $or operator, use the $in operator rather than the $or operator when performing equality checks on the same field.

The operation uses a filter predicate of { status: { $in: ["A", "D"] } }, which corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status in ("A", "D")
```

Refer to the Query and Projection Operators document for the complete list of MongoDB query operators.

Specify AND Conditions

A compound query can specify conditions for more than one field in the collection's documents. Implicitly, a logical AND conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

The following example retrieves all documents in the inventory collection where the status equals "A" **and** qty is less than ($lt) 30:

```
const cursor = db.collection('inventory').find({
  status: 'A',
  qty: { $lt: 30 }
});
```

The operation uses a filter predicate of `{ status: "A", qty: { $lt: 30 } }`, which corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" AND qty < 30
```

See comparison operators for other MongoDB comparison operators.

Specify `OR` Conditions

Using the `$or` operator, you can specify a compound query that joins each clause with a logical `OR` conjunction so that the query selects the documents in the collection that match at least one condition.

The following example retrieves all documents in the collection where the `status` equals `"A"` **or** `qty` is less than (`$lt`) `30`:

```
const cursor = db.collection('inventory').find({
  $or: [{ status: 'A' }, { qty: { $lt: 30 } }]
});
```

The operation uses a filter predicate of

`$or: [ { status: 'A' }, { qty: { $lt: 30 } } ] }`, which corresponds to

the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" OR qty < 30
```

Note:

Queries that use comparison operators are subject to Type Bracketing.

Specify AND as well as OR Conditions

In the following example, the compound query document selects all documents in the collection where the status equals "A" **and** *either* qty is less than ($lt) 30 *or* item starts with the character p:

```
const cursor = db.collection('inventory').find({
  status: 'A',
  $or: [{ qty: { $lt: 30 } }, { item: { $regex: '^p' } }]
});
```

The operation uses a filter predicate of:

```
{
   status: 'A',
   $or: [
     { qty: { $lt: 30 } }, { item: { $regex: '^p' } }
   ]
```

which corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR item LIKE "p%")
```

Note:

MongoDB supports regular expressions $regex queries to perform string pattern matches.

Bitwise Value:

Bitwise operators return data based on bit position conditions.

Note:

For details on a specific operator, including syntax and examples, click on the link to the operator's reference page.

Name	Description
$bitsAllClear	Matches numeric or binary values in which a set of bit positions *all* have a value of 0.
$bitsAllSet	Matches numeric or binary values in which a set of bit positions *all* have a value of 1.
$bitsAnyClear	Matches numeric or binary values in which *any* bit from a set of bit positions has a value of 0.
$bitsAnySet	Matches numeric or binary values in which *any* bit from a set of bit positions has a value of 1.

Query:

This page provides examples of query operations using the Collection.find() method in the MongoDB Node.js Driver.

The examples on this page use the inventory collection. Connect to a test database in your MongoDB instance then create the inventory collection:

```
await db.collection('inventory').insertMany([ {
   item: 'journal', qty: 25, size: { h: 14, w: 21, uom: 'cm' }, status: 'A' },
  { item: 'notebook', qty: 50, size: { h: 8.5, w: 11, uom: 'in' }, status: 'A' },
  { item: 'paper', qty: 100, size: { h: 8.5, w: 11, uom: 'in' }, status: 'D' },
  { item: 'planner', qty: 75, size: { h: 22.85, w: 30, uom: 'cm' }, status: 'D' },
  { item: 'postcard', qty: 45, size: { h: 10, w: 15.25, uom: 'cm' }, status: 'A' }
]);
```

Select All Documents in a Collection

To select all documents in the collection, pass an empty document as the query filter parameter to the find method. The query filter parameter determines the select criteria:

```
const cursor = db.collection('inventory').find({});
```

This operation uses a filter predicate of `{}`, which corresponds to the following SQL statement:

```
SELECT * FROM inventory
```

To see supported options for the `find()` method, see find().

Specify Equality Condition

To specify equality conditions, use `<field>:<value>` expressions in the query filter document:

```
{ <field1>: <value1>, ... }
```

The following example selects from the `inventory` collection all documents where the `status` equals `"D"`:

```
const cursor = db.collection('inventory').find({ status: 'D' });
```

This operation uses a filter predicate of `{ status: "D" }`, which corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "D"
```

Note:

The MongoDB Compass query bar autocompletes the current query based on the keys in your collection's documents, including keys in embedded sub-documents.

Specify Conditions Using Query Operators

A query filter document can use the query operators to specify conditions in the following form:

```
{ <field1>: { <operator1>: <value1> }, ... }
```

The following example retrieves all documents from the `inventory` collection where `status` equals either `"A"` or `"D"`:

```
const cursor = db.collection('inventory').find({
  status: { $in: ['A', 'D'] }
});
```

Note:

Although you can express this query using the `$or` operator, use the `$in` operator rather than the `$or` operator when performing equality checks on the same field.

The operation uses a filter predicate of `{ status: { $in: [ "A", "D" ] } }`, which corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status in ("A", "D")
```

Refer to the Query and Projection Operators document for the complete list of MongoDB query operators.

Specify `AND` Conditions

A compound query can specify conditions for more than one field in the collection's documents. Implicitly, a logical `AND` conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

The following example retrieves all documents in the `inventory` collection where the `status` equals `"A"` **and** `qty` is less than (`$lt`) `30`:

```
const cursor = db.collection('inventory').find({
  status: 'A',
  qty: { $lt: 30 }
});
```

The operation uses a filter predicate of `{ status: "A", qty: { $lt: 30 } }`, which corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" AND qty < 30
```

See comparison operators for other MongoDB comparison operators.

Specify OR Conditions

Using the $or operator, you can specify a compound query that joins each clause with a logical OR conjunction so that the query selects the documents in the collection that match at least one condition.

The following example retrieves all documents in the collection where the status equals "A" **or** qty is less than ($lt) 30:

```
const cursor = db.collection('inventory').find({
  $or: [{ status: 'A' }, { qty: { $lt: 30 } }]
});
```

The operation uses a filter predicate of { $or: [{ status: 'A' }, { qty: { $lt: 30 } }] }, which corresponds to the following SQL statement:

Note:

Queries that use comparison operators are subject to Type Bracketing.

Specify AND as well as OR Conditions

In the following example, the compound query document selects all documents in the collection where the status equals "A" **and** *either* qty is less than ($lt) 30 *or* item starts with the character p:

```
const cursor = db.collection('inventory').find({
  status: 'A',
  $or: [{ qty: { $lt: 30 } }, { item: { $regex: '^p' } }]
});
```

The operation uses a filter predicate of:

```
{
  status: 'A', $or: [ { qty: { $lt: 30 } }, { item: { $regex: '^p' } }
  ] }
```

which corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR item LIKE "p%")
```

Note:

MongoDB supports regular expressions $regex queries to perform string pattern matches.

Geospatial Query:

MongoDB provides powerful capabilities for querying and manipulating geospatial data, which represents locations on Earth or other surfaces. This functionality is particularly useful for applications that involve finding nearby points of interest (POIs), calculating distances, or defining spatial boundaries.

Key Concepts:

1. **GeoJSON:** MongoDB stores geospatial data using GeoJSON (JavaScript Object Notation) format. GeoJSON defines various geometric shapes like points, lines, polygons, and collections.
2. **Geospatial Indexes:** To optimize geospatial queries, you create indexes on the fields containing GeoJSON data. MongoDB offers two types of geospatial indexes:
    1. 2dsphere: Designed for spherical queries on an Earth-like surface.
    2. 2d: Suitable for flat surface queries or some spherical queries (use with caution).

Example: Finding Restaurants Near a User

Imagine you have a MongoDB collection named restaurants that stores restaurant information, including a location field containing the restaurant's address as a GeoJSON point:

```
{
"_id": ObjectId("..."),
"name": "Pizza Palace",
"cuisine": "Italian",
 "location": { "type": "Point",
"coordinates": [-73.985, 40.748] // Example coordinates (longitude, latitude)
 } }
```

Query to Find Restaurants Within 5km of a User's Location:

Here's a MongoDB query that leverages the $geoWithin operator to find restaurants within a 5-kilometer radius of a user's specified location (longitude -74.009, latitude 40.712):

```
db.restaurants.find({
 location: { $geoWithin: {
$geometry: {
type: "Point",
 coordinates: [-74.009, 40.712] // User's location },
$radius: 5000 // Radius in meters (convert 5km to meters) } }
 });
```

OUTPUT:

```
{ "_id": ObjectId("..."), "name": "Pizza Palace", "cuisine": "Italian", ... } // Document for a restaurant within 5km { "_id": ObjectId("..."), "name": "French Bistro", "cuisine": "French", ... } // Another restaurant within 5km (if applicable) ... (more documents if there are other restaurants within 5km)
```

Data types and operations:

Data Type:

- Point
- Line String
- Polygen

Name	Description
`$geoIntersects`	Selects geometries that intersect with a GeoJSON geometry. The `2dsphere` index supports `$geoIntersects`.
`$geoWithin`	Selects geometries within a bounding GeoJSON geometry. The `2dsphere` and `2d` indexes support `$geoWithin`.
`$near`	Returns geospatial objects in proximity to a point. Requires a geospatial index. The `2dsphere` and `2d` indexes support `$near`.
`$nearSphere`	Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The `2dsphere` and `2d` indexes support `$nearSphere`.