MongoDB.

db.collection.find()

Open Interactive Tutorial

On this page

Definition

Compatibility

Syntax

Behavior

Examples

Learn More

MongoDB with drivers

This page documents a `mongosh` method. To see the equivalent method in a MongoDB driver, see the corresponding page for your programming language:

C# | Java Sync | Node.js | PyMongo

Show all

Definition

`db.collection.find(query, projection, options)`

Selects documents in a collection or view and returns a **cursor** to the selected documents.

Returns: A **cursor** to the documents that match the `query` criteria. When the `find()` method "returns documents," the method is actually returning a cursor to the documents.

Compatibility

This method is available in deployments hosted in the following environments:

- **MongoDB Atlas:** The fully managed service for MongoDB deployments in the cloud

NOTE

This command is supported in all MongoDB Atlas clusters. For information on Atlas support for all commands, see **Unsupported Commands.**

- **MongoDB Enterprise:** The subscription-based, self-managed version of MongoDB

- **MongoDB Community:** The source-available, free-to-use, and self-managed version of MongoDB

Syntax

The `find()` method has the following form:

```
db.collection.find( <query>, <proje
```

The `find()` method takes the following parameters:

Parameter	Type	Description
query	document	Optional. Specifies selection filter using **query operators.** To return all documents in a collection, omit this parameter or pass an empty document (`{}`).
projection	document	Optional. Specifies the fields to return in the documents that match the query filter. To return all fields in the matching documents, omit this parameter. For details, see **Projection.**
options	document	Optional. Specifies additional options for the query. These options modify query behavior and how results are returned. For details, see **Options.**

Behavior

Projection

IMPORTANT

Language Consistency

As part of making `find()` and `findAndModify()` projection consistent with aggregation's `$project` stage,

- The `find()` and `findAndModify()` projection can accept **aggregation expressions and syntax.**
- MongoDB enforces additional restrictions with regards to projections. See **Projection Restrictions** for details.

The `projection` parameter determines which fields are returned in the matching documents. The `projection` parameter takes a document of the following form:

```
{ <field1>: <value>, <field2>: <val
```

Projection	Description
`<field>: <1 or true>`	Specifies the inclusion of a field. If you specify a non-zero integer for the projection value, the operation treats the value as `true`.
`<field>: <0 or false>`	Specifies the exclusion of a field.

Projection	Description
`"<field>.$": <1 or true>`	Uses the `$` array projection operator to return the first element that matches the query condition on the array field. If you specify a non-zero integer for the projection value, the operation treats the value as `true`. Not available for **views.**
`<field>: <array projection>`	Uses the array projection operators (`$elemMatch`, `$slice`) to specify the array elements to include. Not available for **views.**
`<field>: <$meta expression>`	Uses the `$meta` operator expression to specify the inclusion of available `per-document metadata`. Not available for **views.**

Projection	Description
`<field>: <aggregation expression>`	Specifies the value of the projected field. With the use of **aggregation expressions and syntax**, including the use of literals and aggregation variables, you can project new fields or project existing fields with new values. • If you specify a non-numeric, non-boolean literal (such as a literal string or an array or an operator expression) for the projection value, the field is projected with the new value, for example: ◦ `{ field: [ 1, 2, 3, "$someExistingField" ] }` ◦ `{ field: "New String Value" }` ◦ `{ field: { status: "Active", total: { $sum: "$existingArray" } } }` • To project a literal value for a field, use the `$literal` aggregation expression; for example: ◦ `{ field: { $literal: 5 } }` ◦ `{ field: { $literal: true } }`

Projection	Description
	◦ `{ field: { $literal: { fieldWithValue0: 0, fieldWithValue1: 1 } } }`

Options

Option	Description
allowDiskUse	Whether or not pipelines that require more than 100 megabytes of memory to execute write to temporary files on disk. For details, see `cursor.allowDiskUse()`.
allowPartialResults	For queries against a sharded collection, allows the command (or subsequent getMore commands) to return partial results, rather than an error, if one or more queried shards are unavailable.
awaitData	If the cursor is a a tailable-await cursor. Requires `tailable` to be `true`.
collation	Collation settings for update operation.
comment	Adds a `$comment` to the query that shows in the **profiler** logs.
explain	Adds explain output based on the verbosity mode provided.
hint	Forces the query optimizer to use specific indexes in the query.
limit	Sets a limit of documents returned in the result set.

Option	Description
max	The exclusive upper bound for a specific index.
maxAwaitTimeMS	The maximum amount of time for the server to wait on new documents to satisfy a tailable cursor query. Requires `tailable` and `awaitData` to be `true`.
maxTimeMS	The maximum amount of time (in milliseconds) the server should allow the query to run.
min	The inclusive lower bound for a specific index.
noCursorTimeout	Whether the server should timeout the cursor after a period of inactivity (by default 10 minutes).
readConcern	Specifies the read concern level for the query.
readPreference	Specifies the read preference level for the query.
returnKey	Whether only the index keys are returned for a query.
showRecordId	If the `$recordId` field is added to the returned documents. The `$recordId` indicates the position of the document in the result set.
skip	How many documents to skip before returning the first document in the result set.
sort	The order of the documents returned in the result set. Fields specified in the sort, must have an index.

Option	Description
tailable	Indicates if the cursor is tailable. Tailable cursors remain open after the intial results of the query are exhausted. Tailable cursors are only available on **Capped Collections.**

Embedded Field Specification

For fields in an embedded documents, you can specify the field using either:

- **dot notation**, for example `"field.nestedfield": <value>`
- nested form, for example `{ field: { nestedfield: <value> } }`

`_id` Field Projection

The `_id` field is included in the returned documents by default unless you explicitly specify `_id: 0` in the projection to suppress the field.

Inclusion or Exclusion

A `projection` *cannot* contain *both* include and exclude specifications, with the exception of the `_id` field:

- In projections that *explicitly include* fields, the `_id` field is the only field that you can *explicitly exclude*.

- In projections that *explicitly excludes* fields, the `_id` field is the only field that you can *explicitly include*; however, the `_id` field is included by default.

See **Projection Examples.**

Cursor Handling

Executing `db.collection.find()` in `mongosh` automatically iterates the cursor to display up to the first 20 documents. Type `it` to continue iteration.

To access the returned documents with a driver, use the appropriate cursor handling mechanism for the **driver language.**

TIP

See also:

- **Iterate the Returned Cursor**
- **Modify the Cursor Behavior**
- **Available `mongosh` Cursor Methods**

Read Concern

To specify the **read concern** for `db.collection.find()`, use the `cursor.readConcern()` method.

Type Bracketing

MongoDB treats some data types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison. For most data types, however, **comparison**

operators only perform comparisons on documents where the **BSON type** of the target field matches the type of the query operand. Consider the following collection:

```
{ "_id": "apples", "qty": 5 }
{ "_id": "bananas", "qty": 7 }
{ "_id": "oranges", "qty": { "in st
{ "_id": "avocados", "qty": "fourte
```

The following query uses `$gt` to return documents where the value of `qty` is greater than `4`.

```
db.collection.find( { qty: { $gt: 4
```

The query returns the following documents:

```
{ "_id": "apples", "qty": 5 }
{ "_id": "bananas", "qty": 7 }
```

The document with `_id` equal to `"avocados"` is not returned because its `qty` value is of type `string` while the `$gt` operand is of type `integer`.

The document with `_id` equal to `"oranges"` is not returned because its `qty` value is of type `object`.

NOTE

To enforce data types in a collection, use **Schema Validation.**

Sessions

For cursors created inside a session, you cannot call `getMore` outside the session.

Similarly, for cursors created outside of a session, you cannot call `getMore` inside a session.

Session Idle Timeout

MongoDB drivers and `mongosh` associate all operations with a **server session**, with the exception of unacknowledged write operations. For operations not explicitly associated with a session (i.e. using `Mongo.startSession()`), MongoDB drivers and `mongosh` create an implicit session and associate it with the operation.

If a session is idle for longer than 30 minutes, the MongoDB server marks that session as expired and may close it at any time. When the MongoDB server closes the session, it also kills any in-progress operations and open cursors associated with the session. This includes cursors configured with `noCursorTimeout()` or a `maxTimeMS()` greater than 30 minutes.

For operations that may be idle for longer than 30 minutes, associate the operation with an explicit session using `Mongo.startSession()`

and periodically refresh the session using the `refreshSessions` command. See **Session Idle Timeout** for more information.

Transactions

`db.collection.find()` can be used inside **distributed transactions.**

- For cursors created outside of a transaction, you cannot call `getMore` inside the transaction.
- For cursors created in a transaction, you cannot call `getMore` outside the transaction.

IMPORTANT

In most cases, a distributed transaction incurs a greater performance cost over single document writes, and the availability of distributed transactions should not be a replacement for effective schema design. For many scenarios, the **denormalized data model (embedded documents and arrays)** will continue to be optimal for your data and use cases. That is, for many scenarios, modeling your data appropriately will minimize the need for distributed transactions.

For additional transactions usage considerations (such as runtime limit and oplog size limit), see also **Production Considerations.**

Client Disconnection

Starting in MongoDB 4.2, if the client that issued `db.collection.find()` disconnects before the operation completes, MongoDB marks `db.collection.find()` for termination using `killOp`.

New in version 8.0.

You can use query settings to set index hints, set **operation rejection filters**, and other fields. The settings apply to the **query shape** on the entire cluster. The cluster retains the settings after shutdown.

The **query optimizer** uses the query settings as an additional input during query planning, which affects the plan selected to run the query. You can also use query settings to block a query shape.

To add query settings and explore examples, see `setQuerySettings`.

You can add query settings for `find`, `distinct`, and `aggregate` commands.

Query settings have more functionality and are preferred over deprecated **index filters.**

To remove query settings, use `removeQuerySettings`. To obtain the query settings, use a `$querySettings` stage in an

db.collection.hideIn...
db.collection.insert
db.collection.insert...

aggregation pipeline.

Examples

The examples in this section use documents from the **bios collection** where the documents generally have the form:

```
{
    "_id" : <value>,
    "name" : { "first" : <string>,
    "birth" : <ISODate>,
    "death" : <ISODate>,
    "contribs" : [ <string>, ... ],
    "awards" : [
        { "award" : <string>, year:
        ...
    ]
}
```

To create and populate the `bios` collection, see **bios collection.**

Find All Documents in a Collection

The `find()` method with no parameters returns all documents from a collection and returns all fields for the documents. For example, the following operation returns all documents in the **bios collection:**

```
db.bios.find()
```

Find Documents that Match Query Criteria

Query for Equality

- The following operation returns documents in the **bios collection** where `_id` equals `5`:

```
db.bios.find( { _id: 5 } )
```

- The following operation returns documents in the **bios collection** where the field `last` in the `name` embedded document equals `"Hopper"`:

```
db.bios.find( { "name.last": "H
```

NOTE

To access fields in an embedded document, use **dot notation** (`"<embedded document>.<field>"`).

Query Using Operators

To find documents that match a set of selection criteria, call `find()` with the `<criteria>` parameter.

MongoDB provides various **query operators** to specify the criteria.

- The following operation uses the `$in` operator to return documents in the **bios collection** where `_id` equals either `5` or

`ObjectId("507c35dd8fada716c89d0013")`:

```
db.bios.find(
   { _id: { $in: [ 5, ObjectId(
)
```

- The following operation uses the `$gt` operator returns all the documents from the `bios` collection where `birth` is greater than `new Date('1950-01-01')`:

```
db.bios.find( { birth: { $gt: n
```

- The following operation uses the `$regex` operator to return documents in the **bios collection** where `name.last` field starts with the letter `N` (or is `"LIKE N%"`)

```
db.bios.find(
   { "name.last": { $regex: /^N
)
```

For a list of the query operators, see **Query Selectors.**

Query for Ranges

Combine comparison operators to specify ranges for a field. The following operation returns from the **bios collection** documents where `birth`

is between `new Date('1940-01-01')` and `new Date('1960-01-01')` (exclusive):

```
db.bios.find( { birth: { $gt: new D
```

For a list of the query operators, see **Query Selectors.**

Query for Multiple Conditions

The following operation returns all the documents from the **bios collection** where `birth` field is `greater than` `new Date('1950-01-01')` and `death` field does not exists:

```
db.bios.find( {
   birth: { $gt: new Date('1920-01-
   death: { $exists: false }
} )
```

For a list of the query operators, see **Query Selectors.**

Compare Two Fields from A Single Document

`$expr` can contain expressions that compare fields from the same document.

Create a `monthlyBudget` collection with these documents:

```
db.monthlyBudget.insertMany( [
   { _id : 1, category : "food", bu
   { _id : 2, category : "drinks",
   { _id : 3, category : "clothes",
   { _id : 4, category : "misc", bu
   { _id : 5, category : "travel",
] )
```

The following operation uses `$expr` to find documents where the `spent` amount exceeds the `budget`:

```
db.monthlyBudget.find( { $expr: { $
```

Output:

```
{ _id : 1, category : "food", budget :
{ _id : 2, category : "drinks", budget
{ _id : 5, category : "travel", budget
```

Query Embedded Documents

The following examples query the `name` embedded field in the **bios collection.**

Query Exact Matches on Embedded Documents

The following operation returns documents in the **bios collection** where the embedded document `name` is *exactly* `{ first: "Yukihiro", last: "Matsumoto" }`, including the order:

```
db.bios.find(
    { name: { first: "Yukihiro", la
)
```

The `name` field must match the embedded document exactly. The query does **not** match documents with the following `name` fields:

```
{
   first: "Yukihiro",
   aka: "Matz",
   last: "Matsumoto"
}

{
   last: "Matsumoto",
   first: "Yukihiro"
}
```

Query Fields of an Embedded Document

The following operation returns documents in the **bios collection** where the embedded document `name` contains a field `first` with the value `"Yukihiro"` and a field `last` with the value `"Matsumoto"`. The query uses **dot notation** to access fields in an embedded document:

```
db.bios.find(
   {
     "name.first": "Yukihiro",
     "name.last": "Matsumoto"
   }
)
```

The query matches the document where the `name` field contains an embedded document with the field `first` with the value `"Yukihiro"` and a field `last` with the value `"Matsumoto"`. For instance, the query would match documents with `name` fields that held either of the following values:

```
{
  first: "Yukihiro",
  aka: "Matz",
  last: "Matsumoto"
}

{
  last: "Matsumoto",
  first: "Yukihiro"
}
```

For more information and examples, see also **Query on Embedded/Nested Documents.**

Query Arrays

Query for an Array Element

The following examples query the `contribs` array in the **bios collection.**

- The following operation returns documents in the **bios collection** where the array field `contribs` contains the element `"UNIX"`:

```
db.bios.find( { contribs: "UNI
```

- The following operation returns documents in the **bios collection** where the array field `contribs` contains the element `"ALGOL"` or `"Lisp"`:

```
db.bios.find( { contribs: { $i
```

- The following operation use the `$all` query operator to return documents in the **bios collection** where the array field `contribs` contains both the elements `"ALGOL"` and `"Lisp"`:

```
db.bios.find( { contribs: { $a
```

  For more examples, see `$all`. See also `$elemMatch`.

- The following operation uses the `$size` operator to return documents in the **bios collection** where the array size of

`contribs` is 4:

```
db.bios.find( { contribs: { $s
```

For more information and examples of querying an array, see:

- **Query an Array**
- **Query an Array of Embedded Documents**

For a list of array specific query operators, see **Array.**

Query an Array of Documents

The following examples query the `awards` array in the **bios collection.**

- The following operation returns documents in the **bios collection** where the `awards` array contains an element with `award` field equals `"Turing Award"`:

```
db.bios.find(
   { "awards.award": "Turing Aw
)
```

- The following operation returns documents in the **bios collection** where the `awards` array contains at least one element with both the `award` field equals `"Turing Award"` and the `year` field greater than 1980:

```
db.bios.find(
   { awards: { $elemMatch: { aw
)
```

Use the `$elemMatch` operator to specify multiple criteria on an array element.

For more information and examples of querying an array, see:

- **Query an Array**
- **Query an Array of Embedded Documents**

For a list of array specific query operators, see **Array.**

Query for BSON Regular Expressions

To find documents that contain BSON regular expressions as values, call `find()` with the `bsonRegExp` option set to `true`. The `bsonRegExp` option allows you to return regular expressions that can't be represented as JavaScript regular expressions.

The following operation returns documents in a collection named `testbson` where the value of a field named `foo` is a `BSONRegExp` type:

```
db.testbson.find( {}, {}, { bsonReg
```

HIDE OUTPUT

```
[
  {
    _id: ObjectId('65e8ba8a4b3c33a76e6c
    foo: BSONRegExp('(?-i)AA_', 'i')
  }
]
```

Projections

The **projection** parameter specifies which fields to return. The parameter contains either include or exclude specifications, not both, unless the exclude is for the `_id` field.

NOTE

Unless the `_id` field is explicitly excluded in the projection document `_id: 0`, the `_id` field is returned.

Specify the Fields to Return

The following operation finds all documents in the **bios collection** and returns only the `name` field, `contribs` field and `_id` field:

```
db.bios.find( { }, { name: 1, contr
```

NOTE

Unless the `_id` field is explicitly excluded in the projection document `_id: 0`, the `_id` field is returned.

Explicitly Excluded Fields

The following operation queries the **bios collection** and returns all fields *except* the `first` field in the `name` embedded document and the `birth` field:

```
db.bios.find(
   { contribs: 'OOP' },
   { 'name.first': 0, birth: 0 }
)
```

Explicitly Exclude the `_id` Field

NOTE

Unless the `_id` field is explicitly excluded in the projection document `_id: 0`, the `_id` field is returned.

The following operation finds documents in the **bios collection** and returns only the `name` field and the `contribs` field:

```
db.bios.find(
   { },
   { name: 1, contribs: 1, _id: 0 }
)
```

On Arrays and Embedded Documents

The following operation queries the **bios collection** and returns the `last` field in the `name` embedded document and the first two elements in the `contribs` array:

```
db.bios.find(
   { },
   { _id: 0, 'name.last': 1, contri
```

You can also specify embedded fields using the nested form. For example:

```
db.bios.find(
   { },
   { _id: 0, name: { last: 1 }, con
)
```

Use Aggregation Expression

`db.collection.find()` projection can accept **aggregation expressions and syntax.**

With the use of aggregation expressions and syntax, you can project new fields or project existing fields with new values. For example, the following operation uses aggregation expressions to override the value of the `name` and `awards` fields as well as to include new fields `reportDate`, `reportBy`, and `reportNumber`.

```
db.bios.find(
   { },
   {
     _id: 0,
     name: {
        $concat: [
           { $ifNull: [ "$name.aka"
           " ",
           "$name.last"
        ]
     },
     birth: 1,
     contribs: 1,
     awards: { $cond: { if: { $isAr
     reportDate: { $dateToString: {
     reportBy: "hellouser123",
     reportNumber: { $literal: 1 }
   }
)
```

To set the `reportRun` field to the value `1` The operation returns the following documents:

```
{ "birth" : ISODate("1924-12-03T05:00:0
{ "birth" : ISODate("1927-09-04T04:00:0
{ "birth" : ISODate("1906-12-09T05:00:0
{ "birth" : ISODate("1926-08-27T04:00:0
{ "birth" : ISODate("1931-10-12T04:00:0
{ "birth" : ISODate("1956-01-31T05:00:0
{ "birth" : ISODate("1941-09-09T04:00:0
{ "birth" : ISODate("1965-04-14T04:00:0
{ "birth" : ISODate("1955-05-19T04:00:0
{ "contribs" : [ "Scala" ], "name" : "M
```

TIP

See also:

Project Fields to Return from Query

Iterate the Returned Cursor

The `find()` method returns a **cursor** to the results.

In `mongosh`, if the returned cursor is not assigned to a variable using the `var` keyword, the cursor is automatically iterated to access up to the first 20 documents that match the query. You can update the `displayBatchSize` variable to change the number of automatically iterated documents.

The following example sets the batch size to 3. Future `db.collection.find()` operations will only return 3 documents per cursor iteration.

```
config.set( "displayBatchSize", 3 )
```

To manually iterate over the results, assign the returned cursor to a variable with the `var` keyword, as shown in the following sections.

With Variable Name

The following example uses the variable `myCursor` to iterate over the cursor and print the matching documents:

```
var myCursor = db.bios.find( );

myCursor
```

With `next()` Method

The following example uses the cursor method `next()` to access the documents:

```
var myCursor = db.bios.find( );

var myDocument = myCursor.hasNext()

if (myDocument) {
    var myName = myDocument.name;
    print (tojson(myName));
}
```

To print, you can also use the `printjson()` method instead of `print(tojson())`:

```
if (myDocument) {
   var myName = myDocument.name;
   printjson(myName);
}
```

With `forEach()` Method

The following example uses the cursor method `forEach()` to iterate the cursor and access the documents:

```
var myCursor = db.bios.find( );

myCursor.forEach(printjson);
```

Modify the Cursor Behavior

`mongosh` and the **drivers** provide several cursor methods that call on the *cursor* returned by the `find()` method to modify its behavior.

Order Documents in the Result Set

The `sort()` method orders the documents in the result set. The following operation returns documents in the **bios collection** sorted in ascending order by the `name` field:

```
db.bios.find().sort( { name: 1 } )
```

`sort()` corresponds to the `ORDER BY` statement in SQL.

Limit the Number of Documents to Return

The `limit()` method limits the number of documents in the result set. The following operation returns at most `5` documents in the **bios collection:**

```
db.bios.find().limit( 5 )
```

`limit()` corresponds to the `LIMIT` statement in SQL.

Set the Starting Point of the Result Set

The `skip()` method controls the starting point of the results set. The following operation skips the first `5` documents in the **bios collection** and returns all remaining documents:

```
db.bios.find().skip( 5 )
```

Specify Collation

Collation allows users to specify language-specific rules for string comparison, such as rules for lettercase and accent marks.

The `collation()` method specifies the **collation** for the `db.collection.find()` operation.

```
db.bios.find( { "name.last": "hoppe
```

Combine Cursor Methods

The following statements chain cursor methods `limit()` and `sort()`:

```
db.bios.find().sort( { name: 1 } ).
db.bios.find().limit( 5 ).sort( { n
```

The two statements are equivalent; i.e. the order in which you chain the `limit()` and the `sort()` methods is not significant. Both statements

return the first five documents, as determined by the ascending sort order on 'name'.

Available `mongosh` Cursor Methods

- `cursor.allowDiskUse()`
- `cursor.allowPartialResults()`
- `cursor.batchSize()`
- `cursor.close()`
- `cursor.isClosed()`
- `cursor.collation()`
- `cursor.comment()`
- `cursor.count()`
- `cursor.explain()`
- `cursor.forEach()`
- `cursor.hasNext()`
- `cursor.hint()`
- `cursor.isExhausted()`
- `cursor.itcount()`
- `cursor.limit()`
- `cursor.map()`
- `cursor.max`
- `cursor.max`
- `cursor.mir`
- `cursor.nex`
- `cursor.noC`
- `cursor.obj`
- `cursor.pre`
- `cursor.rea`
- `cursor.rea`
- `cursor.ret`
- `cursor.sho`
- `cursor.siz`
- `cursor.ski`
- `cursor.sor`
- `cursor.tai`
- `cursor.toA`

Use Variables in `let` Option

You can specify query options to modify query behavior and indicate how results are returned.

For example, to define variables that you can access elsewhere in the `find` method, use the `let` option. To filter results using a variable, you must access the variable within the `$expr` operator.

Create a collection `cakeFlavors`:

```
db.cakeFlavors.insertMany( [
   { _id: 1, flavor: "chocolate" },
   { _id: 2, flavor: "strawberry" }
   { _id: 3, flavor: "cherry" }
] )
```

The following example defines a `targetFlavor` variable in `let` and uses the variable to retrieve the chocolate cake flavor:

```
db.cakeFlavors.find(
   { $expr: { $eq: [ "$flavor", "$$
   { _id: 0 },
   { let : { targetFlavor: "chocola
} )
```

Output:

```
[ { flavor: 'chocolate' } ]
```

Retrieve Documents for Roles Granted to the Current User

Starting in MongoDB 7.0, you can use the new `USER_ROLES` system variable to return user **roles.**

The scenario in this section shows users with various roles who have limited access to documents in a collection containing budget information.

The scenario shows one possible use of `USER_ROLES`. The `budget` collection contains documents with a field named `allowedRoles`. As you'll see in the following scenario, you can write queries that compare the user roles found in the `allowedRoles` field with the roles returned by the `USER_ROLES` system variable.

NOTE

For another `USER_ROLES` example scenario, see **Retrieve Medical Information for Roles Granted to the Current User.** That example doesn't store the user roles in the document fields, as is done in the following example.

For the budget scenario in this section, perform the following steps to create the roles, users, and `budget` collection:

1. **Create the roles**

   Run:

```
db.createRole( { role: "Market
db.createRole( { role: "Sales"
db.createRole( { role: "Develo
db.createRole( { role: "Operat
```

2 Create the users

Create users named `John` and `Jane` with the required roles. Replace the `test` database with your database name.

```
db.createUser( {
   user: "John",
   pwd: "jn008",
   roles: [
      { role: "Marketing", db:
      { role: "Development", d
      { role: "Operations", db
      { role: "read", db: "tes
   ]
} )

db.createUser( {
   user: "Jane",
   pwd: "je009",
   roles: [
      { role: "Sales", db: "te
      { role: "Operations", db
      { role: "read", db: "tes
   ]
} )
```

3 Create the collection

Run:

```
db.budget.insertMany( [
   {
      _id: 0,
      allowedRoles: [ "Marketi
      comment: "For marketing
      yearlyBudget: 15000
   },
   {
      _id: 1,
      allowedRoles: [ "Sales"
      comment: "For sales team
      yearlyBudget: 17000,
      salesEventsBudget: 1000
   },
   {
      _id: 2,
      allowedRoles: [ "Operati
      comment: "For operations
      yearlyBudget: 19000,
      cloudBudget: 12000
   },
   {
      _id: 3,
      allowedRoles: [ "Develop
      comment: "For developmen
      yearlyBudget: 27000
   }
] )
```

Perform the following steps to retrieve the documents accessible to `John`:

1. **Log in as John**

   Run:

```
db.auth( "John", "jn008" )
```

2 Retrieve the documents

To use a system variable, add `$$` to the start of the variable name. Specify the `USER_ROLES` system variable as `$$USER_ROLES`.

Run:

```
db.budget.find( {
   $expr: {
      $not: {
         $eq: [ { $setIntersec
      }
   }
} )
```

The previous example returns the documents from the `budget` collection that match at least one of the roles that the user who runs the example has. To do that, the example uses `$setIntersection` to return documents where the intersection between the `budget` document `allowedRoles` field and the set of user roles from `$$USER_ROLES` is not empty.

3 Examine the documents

`John` has the `Marketing`, `Operations`, and `Development` roles, and sees these documents:

```
[
   {
      _id: 0,
      allowedRoles: [ 'Marketing' ]
      comment: 'For marketing team
      yearlyBudget: 15000
   },
   {
      _id: 2,
      allowedRoles: [ 'Operations'
      comment: 'For operations tear
      yearlyBudget: 19000,
      cloudBudget: 12000
   },
   {
      _id: 3,
      allowedRoles: [ 'Development
      comment: 'For development tea
      yearlyBudget: 27000
   }
]
```

Perform the following steps to retrieve the documents accessible to `Jane`:

1. **Log in as Jane**

   Run:

   ```
   db.auth( "Jane", "je009" )
   ```

2 Retrieve the documents

Run:

```
db.budget.find( {
   $expr: {
      $not: {
         $eq: [ { $setIntersec
      }
   }
} )
```

3 Examine the documents

`Jane` has the `Sales` and `Operations` roles, and sees these documents:

```
[
   {
      _id: 1,
      allowedRoles: [ 'Sales' ],
      comment: 'For sales team',
      yearlyBudget: 17000,
      salesEventsBudget: 1000
   },
   {
      _id: 2,
      allowedRoles: [ 'Operations'
      comment: 'For operations tea
      yearlyBudget: 19000,
      cloudBudget: 12000
   }
]
```

NOTE

On a sharded cluster, a query can be run on a shard by another server node on behalf of the user. In those queries, `USER_ROLES` is still populated with the roles for the user.

Modify a Query with options

The following examples show how you can use the `options` field in a `find()` query. Use the following `insertMany()` to setup the `users` collection:

```
db.users.insertMany( [
   { username: "david", age: 27 },
   { username: "amanda", age: 25 },
   { username: "rajiv", age: 32 },
   { username: "rajiv", age: 90 }
] )
```

limit with options

The following query limits the number of documents in the result set with the `limit` options parameter:

```
db.users.find(
   { username : "rajiv"}, // query
   { age : 1 }, // projection
   { limit : 1 } // options
)
```

allowDiskUse with options

The following query uses the `options` parameter to enable `allowDiskUse`:

```
db.users.find(
   { username : "david" },
   { age : 1 },
   { allowDiskUse : true }
)
```

explain with options

The following query uses the `options` parameter to get the `executionStats` explain output:

```
var cursor = db.users.find(
   { username: "amanda" },
   { age : 1 },
   { explain : "executionStats" }
)
cursor.next()
```

Specify Multiple options in a query

The following query uses multiple `options` in a single query. This query uses `limit` set to `2` to return only two documents, and `showRecordId` set to `true` to return the position of the document in the result set:

```
db.users.find(
    {},
    { username: 1, age: 1 },
    {
      limit: 2,
      showRecordId: true
    }
)
```

Learn More

- `findOne()`
- `findAndModify()`
- `findOneAndDelete()`
- `findOneAndReplace()`

English

About

Careers

Investor Relations

Legal

GitHub

Security Information

Trust Center

Connect with Us

Support

Contact Us

Customer Portal

Atlas Status

Customer Support

Manage Cookies

Deployment Options

MongoDB Atlas

Enterprise Advanced

Community Edition