Cheatsheets / **Learn MongoDB**

codecademy

# MongoDB CRUD I

## Printing a List of Databases

In the `mongosh` shell, running the command `show dbs` prints a list of all the databases in a MongoDB instance.

## Connecting to a Database

In the `mongosh` shell, the command `use <db>` is used to navigate to the specified database.
For example, the following command will select the existing database called `store`:

```
use store
```

## Viewing a Selected Database

In the `mongosh` shell, the command `db` is used to verify which database is actively being used.
For example, if we switched to a database called `concerts`, then running the command `db` will display the following:

```
> db
concerts
```

## Cursors

Successful MongoDB queries will return a cursor. A cursor is an object that points to documents matched by a query filter. Cursors return results in batches and can be iterated to return subsequent documents.
If we run the following command for a collection called `storage_room`:

```
db.storage_room.find( { id: 1
})
```

The cursor will be the returned document. You can think of the cursor as a pointer that points to a specific record(s).

## Iterating Over a Cursor

In the `mongosh` shell, the command `it` (short for iterate) will iterate through a cursor to show the next batch of documents.

## The `.find()` Method

The `.find()` method is used to select documents in a collection and return a cursor to the selected documents.
The following example will print all documents from a collection named `posts`:

```
db.posts.find()
```

```
db.<collection>.find()
```

## Querying with `.find()`

When using the `.find()` method, a query can be passed in as a parameter to search for specific documents.
For example, consider a collection named `guitarists`.
To retrieve a specific document, query criteria can be passed in:

```
db.guitarists.find({
last_name: "Santana" })
```

```
db.<collection>.find({ <field>: <value>
})
```

> Note: If no document matches the filtering criteria, then no documents will be printed.

## The `$gt` and `$gte` Comparison Operators

The comparison operators `$gt` and `$gte` can be used to match documents where the value for a particular field is greater than, or greater than or equal to a specified value, respectively.
For example, consider a collection named `inventory` where you're trying to retrieve documents with a field named `quantity` with a value greater than or equal to `20`. This can be accomplished with the following command:

```
db.inventory.find( {
quantity: { $gte: 20 } } )
```

```
db.<collection>.find({ <field>: { $gte:
<value> } })
```

## The `.sort()` Method

The `.sort()` method can be appended to queries to order matching documents according to one or more fields. To specify sorting orders, a value of `1` and `-1` are used. `1` is used for ascending order while `-1` is used for descending order.

For example, consider a collection named `documentaries`. If we wanted to view a list of documents sorted in descending order by the field `release_year`, we would use the following command:

```
db.documentaries.find().sort(
{ release_year: -1 })
```

```
db.<collection>.find().sort( { <field>:
<sort_order> } )
```

## The `.sort()` Method with Duplicate Values

When sorting with `.sort()` on fields that have duplicate values, documents that have those values may be returned in any order. Additional fields can be included in the sort document to produce consistent results.

For example, consider a collection named `students` that holds two of the following documents:

```
{ first_name: "Lucy",
last_name: "Wang", age: 12,
grade: 7 },
{ first_name: "Lucy",
last_name: "Alfe", age: 12,
grade: 7 }
```

When running the following command, the duplicate values will not be returned in a sorted manner due to having duplicate values in the collection:

```
db.students.find().sort({
first_name: 1 })
```

## Query Projections with `.find()` (inclusive)

When using the `.find()` method, we can limit the amount of data returned by including a projection document to specify or restrict fields to return. For example, consider a collection named `restaurants`. Assuming each document includes several fields, if we want to view only the names and locations of restaurants that have a `grade` field, with the value of `"A"`, we can run the following command:

```
db.restaurants.find( { grade:
"A" }, { name: 1, location: 1
} )
```

If the field in the projection has the value of `1`, it will be included in the returned documents.

```
db.<collection>.find( { <field>: <value>
}, { <field>: 1 } )
```

## Query Projections with `.find()` (exclusive)

When running the `.find()` method, we can limit the amount of data returned by including a projection document to specify or restrict fields to return. Consider a collection named `inventory`. Assuming each document includes several fields, if we want to view all the fields of the documents and *only* exclude the field `origin` and `original_cost`, we can run the following command:

```
db.restaurants.find( {}, {
origin: 0, original_cost: 0 }
)
```

```
db.<collection>.find( { <field>: <value>
}, { <field>: 1 } )
```

## Query Projection Limitations

When using projection queries, it is not possible to combine inclusion and exclusion statements in a single projection document except for the `_id` field.
For example, consider a collection named `artists` with documents holding the following format:

```
{ _id: 1, first_name:
"Pablo", last_name:
"Picasso", movement: "Cubism"
},
{ _id: 2, first_name:
"Clause", last_name: "Monet",
movement: "Impressionism" },
{ _id: 3, first_name:
"Henri", last_name:
"Matisse", movement:
"Modernism" }
```

The only command that can be run that includes both exclusion and inclusion statements involves the `_id` field:

```
db.artists.find( { }, {
last_name: 1, movement: 1,
_id: 0 } )
```

## Embedded Documents

Embedded documents are documents nested inside of another. One parent document can have many embedded child documents and can help define relationships between the data.
For example, consider a collection named `courses` that holds documents describing a specific course. Each course document will hold a field `teacher` which will be a document in itself. The format for an embedded document could look like so:

```
{
  course: "CS101",
  course_number: 10401,
  teacher: { first_name:
"Sam", last_name: "Juarez" },
  branch: "Computer Science"
}
```

In the example above, the teacher is another document.

## Accessing Embedded Documents

An embedded document can be accessed by using the .find() method and using dot notation in our query filter to access the embedded field.

For example, consider a collection named `employees` with documents in the following format:

```
{
  firstName: "John",
  lastName: "King",
  department: { name:
"Finance" },
  address: {
    phone: { type: "Home",
number: "111-000-000" }
  }
}
```

To retrieve documents from the collection from the `"Finance"` department, we can run the following command:

```
db.employees.find({
"department.name": "Finance"
}
```

```
db.<collection>.find({ <"field1.field2">:
<value> })
```

## The `$lt` and `$lte` Comparison Operators

The comparison operators `$lt` and `$lte` can be used to match documents where the value for a particular field is less than, or less than or equal to the specified value, respectively.

For example, consider a collection named `inventory` where we want to retrieve documents with a field named `quantity` with a value less than or equal to `20`. This can be accomplished with the following command:

```
db.inventory.find( {
quantity: { $lte: 20 } } )
```

```
db.<collection>.find({ <field>: { $lte:
<value> } })
```

## Exact Array Match

The `.find()` method can query a collection for an array containing specific values by passing in a query argument that includes the field and array to match:

Check out the following example of a `.find()` operation on a collection named `interviewees`:

```
db.interviewees.find({
  languages: ["C#",
"Javascript", "Ruby"]
})
```

```
db.<collection>.find({
  <field>: [ <value1> , <value2> ... ]
})
```

## Single Array Element Match

The `.find()` method can query collections on a single array element by providing a query document containing the field name and the specific element we want to match as its value.
Check out the following example searching for a document containing `"Best Picture"` as a value in a field named `"awards"`:

```
db.academyawards.find({
  awards: ["Best Picture"]
})
```

```
db.<collection>.find({
  <field>: [ <value> ]
})
```

## The `$all` operator

The `.$all` operator selects the documents where the value of a field is an array that contains all the specified elements, in any order, without regard to additional elements in the array.
The query below will match documents that contain at least those specified values, in any order, in the field `awards`.

```
db.academyawards.find({
  awards: { $all: [ "Best
Picture", "Best Director",
"Best Director" ] }
})
```

```
db.<collection>.find({
  <field>: { $all: [ <value1> , <value2>
... ] }
})
```

## Comparison Operators on Array Fields

The `.find()` method can query a collection with comparison operators to match documents where the array contains one or more elements that satisfy the query conditions in some combination.
The following query will find a document in a `ufc_contestants` collections where the array field `championship_years` is greater than 2004 but less than 2008:

```
db.ufc_contestants.find({
  championship_years: { $gt:
2004, $lt: 2008 }
})
```

```
db.<collection>.find({
  <field>: { <comparison operator> }
})
```

## The `$elemMatch` operator

The `.find()` method can use the `$elemMatch` operator to find documents with at least one array field. The finding operation matches the data in the array field with the criteria mentioned with `$elemMatch`.
The following query looks for documents in which the array field `bands` contains values where `active` is true, and the `formation_year` is greater than 1990:

```
db.music_fest.find({
  bands: { $elemMatch: {
active: true, formation_year:
{ $gte: 1990} } }
})
```

```
db.<collection>.find({
  <field>: { $elemMatch: { <query1>,
<query2>, ... } }
})
```

## Array Fields with Embedded Documents

We can use `.find()` to query an array with embedded documents for an exact match by providing a query argument containing the specific fields and their associated values. The query will only match documents with the specific order we provide in our query.

Consider the following document as an example:

```
{
  _id: ObjectId(...),
  name: "Miyu Kato",
  country: "Japan",

wimbledon_doubles_placements:
  [{
    year: 2016,
    place: 2
  },
  {
    year: 2017,
    place: 1
  }]
}
```

We can query the embedded documents inside of the `wimbledon_doubles_placements` array field:

```
db.tennis_players.find( {
"wimbledon_doubles_placements
": { year: 2016, place: 2 } }
)
```

```
db.<collection>.find({
  <field>: { <array_field>: <array_value>
}
})
```

## Array Fields with Embedded Documents II

We can use `.find()` to query an array with embedded documents for a single field by providing a query argument for the specific field using dot ( `.` ) notation. The following query will query the array field `wimbledon_doubles_placements` for a embedded document with the field `year` with the value of `2016` :

```
db.tennis_players.find( {
"wimbledon_doubles_placements
.year": 2016 )
```

```
db.<collection>.find({
  "field.nestedfield": <value>
})
```

Print Share