MySQL Notes

Iñaki Lakunza

June 16, 2024

Contents

1 Introduction

MongoDB is a NO-SQL database management system. Unlike relational databse systems (like MySQL), MongoDB uses a NO-SQL format to retrieve data.

NO-SQL means not only SQL, the data is also stored in various formats besides the traditional sql table.

With MongoDB we store data in pairs similarly to JSON format, but it is technically Binary Javascript Object Notation, but it behaves very similarly to JSON.

The general idea is that data which is frequently accessed together is stored together rather than in separate tables.

We work with documents, **a DOCUMENT is a group of field-value pairs to represent an object**.

And a **COLLECTION is a group of one or more documents**.

And **a DATABASE is a group of one or more collections**.

After installing mongodb and its shell and adding the path of the shell to the local environments path, we can go to the mongodb shell.

If we write **mongodb** it will stablish the connection.

We can type **cls** to clean the terminal.

And we can type **exit** to exit.

But, instead of working this way, we can work using VSCode. We can just install the oficcial MongoDB extensions and work with it.

After we have connected it with our host, we can right click on the localhost connection and press *launch mongodb shell* to open up its shell.

If we want to see the databases we have we can type **show dbs**, and if we want ot use any of them we can write **use (name)**.

If we type the name of a database which does not exist it will be created, but, if we again type **show dbs** it will not appear, it will no visible yet since it is empty.

In order to show up, we will add a collection to it, this can be done using the following line:

```
db.createCollection("(name of the collection)")
```

Now it will appear if we try to find it.

Currently we are in the *school* database, and we can type the following line to **drop it**:

```
db.dropDatabase()
```

Now if we try to find it again it will not appear.

This is how we create and delete databases using the shell.

Doing these actions on the Compass is quite straightforward.

2 Insert

To insert a document **within the database we are working with** (so we have to access it using **use (name of the db)**).

If a collection is found withing the database we have to use its name, and otherwise it will be created:

```
db.students.insertOne({name: "Spongebob", age: 30, gpa: 3.2})
db.students.find()
```

In this example we have used the collection named *students* and we have introduced a document. **A document is enclosed using {}, same as a dictionary or json file**.

And with the second line we have looked at the documents inside that collection, in this case the one we have created.

We will see that apart from the field-value pairs that we have created another one will be there, named *_id*, which is created automatically, and works as a identifier.

We can insert more than a document at a time by using the **insertMany()** function:

```
db.students.insertMany([{name:"Patrick", age:38, gpa:1.5},
    {name:"Sandy", age:27, gpa:4.0}, {name:"Gary", age:18, gpa:2.5}])
```

We have to insert the documents within an array, an separated by commas.
And all the documents do NOT need to have the same fields, they can be different.

If we want to do this using Compass we have to first go to the database we want to work with in the left panel, then we have to go to the current collection, and select *ADD DATA*, we can import a json or we can do it manually.

3 Data Types

A **string** is a series of text within quotes, the quotes can be double or single.

An **integer** is a whole number. And **doubles** have decimals.

Booleans are either True or False, and **they are written as *true* or *false*, all in lowercase**.

Then we have **dates**, to create a date object we have to use the **new** keyboard followed by a **Date(constructor)**. If we do not pass any arguments to the constructor then it will use the current time in the UTC timezone, otherwise we can pass the date we want, and we can add a day too.

Then we have the **null** value, which is not a value, **when we create a null value all we are doing is to create a placeholder**.

Then we have a **arrays/lists**, which are enclosed withing [], and they allow one field to contain multiple values.

We also have **nested documents**, to create a nested document we use { and }.Withing the nested document we can insert a set of field-value pairs

```
db.students.insertOne({name: "Larry",
                        age: 32,
                        gpa: 2.8,
                        fullTime: false,
                        regirterDate: new Date("2023-01-02T00:10:05"),
                        birthDate: new Date("2000-05-20"),
                        gradutationDate: null,
                        courses: ["Biology", "Chemistry", "Calculus"],
                        address: {
                             street: "123 Fake St.",
                             city: "Bikiny Bottom"
                                 }
                        }
```

4 Sorting and limiting

As we know, we can type **db.(collection name).find()** to find the documents we have inside the chosen collection.

If we want to sort these documents in some sort of order we have to do the following, using the previous example, where the name of the collection was *students*:

```
db.students.find().sort({name:1}) # Alphabetical order
db.students.find().sort({name:-1}) # REVERSE alphabetical order
```

We can also limit the amount of documents that are returned to us:

```
db.students.find().limit(1)
```

In the above example we have chosen to just return one document, **and the order is kept the one we have chosen previously, or the one by default if we did not place any order**. And we can also combine **sort** and **limit**. For instance, if we want to find who is the student with the highest gpa:

```
db.students.find().sort({gpa:-1}).limit(1)
```

5 Find

As we know, by using **find** we will return all the documents in the collection.

But, we can find specific documents using arguments. **We first have to use a QUERY document and then a PROJECTION document (so they have to be inside curly brackets)**.

If we do the following, for example, we will return any document which has the inserted field-value pair (we can insert more than one pair):

```
db.students.find({name:"Spongebob"})
```

The query document is very similar to a *WHERE* clause in SQL.

With the **projection document** we can return specific fields. **Note that if we are not using the query document we have to pass it empty**.

With the following command, we will return all the documents just with the names:

```
db.students.find({}, {name:true})
```

By default, if we do not specify it MangoDB will also return the *_id* parameters, but we can splicitly choose not to get it typing **_id:false** in the projection document.

6 Update

Same way as creating documents, we can update one or more documents at the same time by differentiating *One* and *Many*.

There are two parameters that we have to select: **filter** and **update**.

filter is the selection criteria for the update, so withing the first curly braces we will pass the criteria.

In the **update** parameter we have to choose which updates we want to place, for it, we will utilize the **set operator**, which is preceeded by a dollar sign $. **The set operator replaces the value of a field**, so we have to add : and then another set of curly braces, and there we can make our changes.

If the field exists it will update it, and if it does not exist it will create it.

```
db.students.updateOne({name: "Spongebob"}, {$set:{fullTime:true}})
```

If we want to remove a field we can use the **unset** operator:

```
db.students.updateOne({_id: ObjectID("...")}, {$unset:{fullTime:""}})
```

And we will be able to do the same using **updateMany**, so we will do the changes to many documents.

For instance, if any student does not have a *full_time* field, we want to place it:

```
db.students.updateMany({fullTime:{$exists:false}}, {$set:{fullTime:true}})
```

7 Delete

In Compass we just have to select the trash buttom to delete a database, a collection or a file.

In the shell we have to use **delete** and we have to specify if we want to delete just one file or many:

```
db.students.deleteOne({name:"Larry"})
```

In the above example we are deleting the file with the chosen field-value pair.

```
db.students.deleteMany({fulltime:false})
```

8 Comparison Operators

Operators are denoted with a $ dollar sign and they return data based on value comparisons.

For example, if we want to find everybody that it is not spongebob:

```
db.students.find({name:{$ne:"Spongebob"}})
```

ne stands for **Not Equal**.
lt stands for **Less Than** and **lte** stands for **Less Than or Equal**.
gt stands for **Greather than** and **gte** stands for **Greater Than or Equal**.

We can also use comparison operators to find values within a certain range:

```
db.students.find({gpa:{$gte:3, $lte:4}})
```

The **in** operator returns all elements with a matching value:

```
db.students.find({name:{$in:["Spongebob", "Patric", "Sandy"]})
```

And **nin** stands for **Not IN**.

And in compass we can write these conditions in the finder bar.

9 Logical Operators

Logical operators return data based on expressions are either true or false. They are AND, NOT, NOR and OR.

And they also used with dollar signs $ since they are operators:

```
db.students.find({$and: [{fullTime:true}, {age:{$lte:22}}]})
```

So we are returning students who are doing fulltime and have an age less than or equal to 22.

A benefit of the NOT operator is that it will give us elements with Null values. So, for instance if we want to find students whose age is lower than 30 or that have a Null value in their age:

```
db.students.find({age:{$not:{$gte:30}}})
```

10 Indexes

Indexes allow for quick lookup of a field, however it takes up more memory and slows insert, update and remove operations.

By using indexes we are storing our data as a B-Tree.

Let's find anybody named Larry.
We can type *.explain("executionStats")* to give us more information of our query

```
db.students.find({name:"Larry"}).explain("executionStats")
```

We are performing a linear search on our collection. If it is small it will not take too long, but if it is large it will.
We can speed up the process by applying an index.

```
db.students.createIndex({name:1}) # Ascending order

db.students.createIndex({name:-1}) # Descending order
```

We are creating indexes using the name and using the wanted order.

The get all of our indexes we can use the following line:

```
db.students.getIndexes()
```

By default object ids will have an index.
If we want to drop an index, we have to use its name, which can be seen with the previous command:

```
db.students.dropIndex("name_1")
```

In Compass we would have to go to the Indexes section.

Indexes should be used when we do a lot of searching and almost no updating.

11 Collections

A collection is a group of documents and a database is a group of collections.

We can type **show collections** to show the collections on the current database.

For creating a collection we can type:

```
db.createCollection("teachers",
    {capped:true, size:1024000, max:100},
    {autoIndexID:true})
```

If we set the *capped* parameter true we have to specify a maximum size or a maximum number of documents.

In this case we have chosen the maximum size to be 10MB (1000x1024) and a maximum number of 100 documents.

And, **with the *autoIndexId* parameter we specify if by default we want to have indexing in the object IDs**.