

# MySQL Notes

Iñaki Lakunza

June 13, 2024

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Insert</b>	<b>3</b>
<b>3</b>	<b>Data Types</b>	<b>3</b>
<b>4</b>	<b>Sorting and limiting</b>	<b>4</b>
<b>5</b>	<b>Find</b>	<b>4</b>
<b>6</b>	<b>Update</b>	<b>5</b>
<b>7</b>	<b>Delete</b>	<b>5</b>

# 1 Introduction

MongoDB is a NO-SQL database management system. Unlike relational database 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 establish 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 official 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 to 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 not be 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"),
                        graduationDate: 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