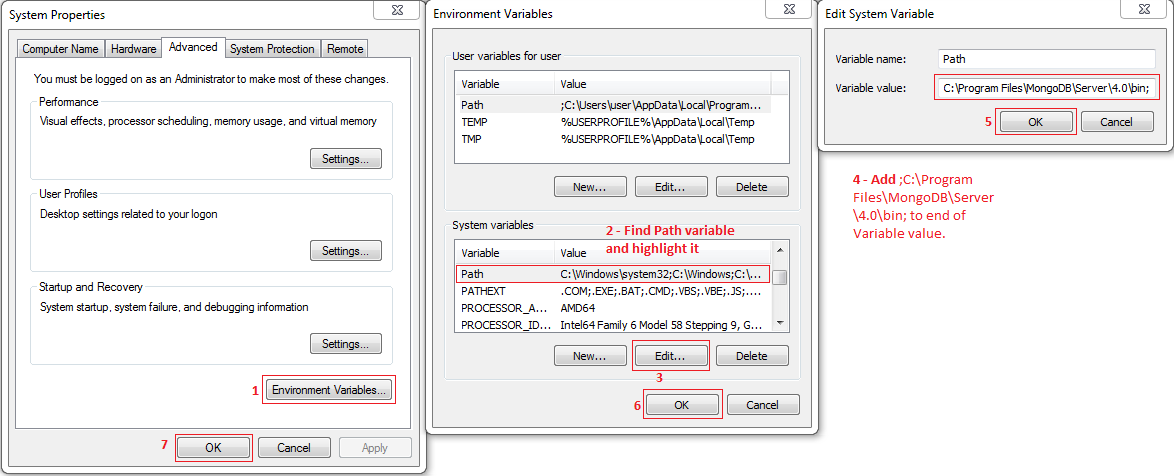
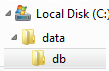
MongoDB

Installation,

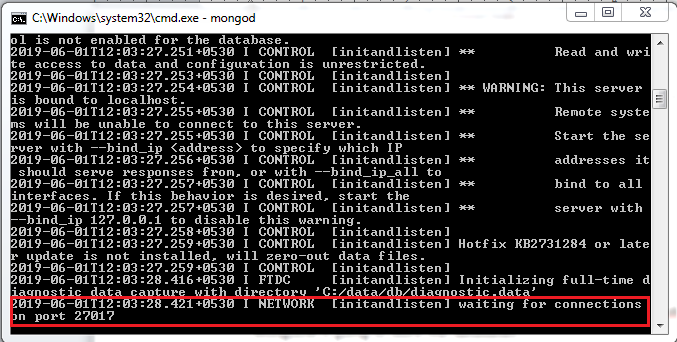
1. Go to <https://www.mongodb.com/download-center/community>, download MongoDB database server executable (Mongo Community Server).
2. Go to <https://www.mongodb.com/products/compass>, download the Compass – an interactive editor for MongoDB
3. Note this path, **;C:\Program Files\MongoDB\Server\4.0\bin;**
   1. Search for “edit the system environment variables”, then follow the steps as below.



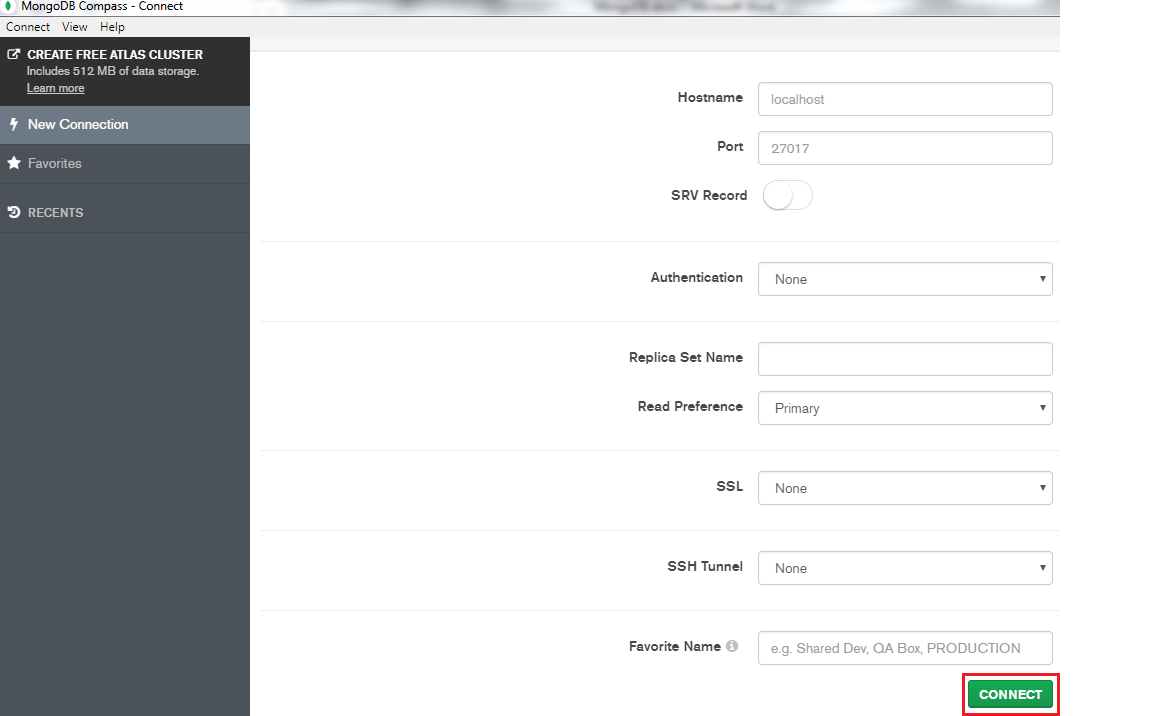
1. Create a nested folder in the C-drive like below, or cmd.exe “mkdir c:\\data\db”. This is where MongoDB is going to store the databases.



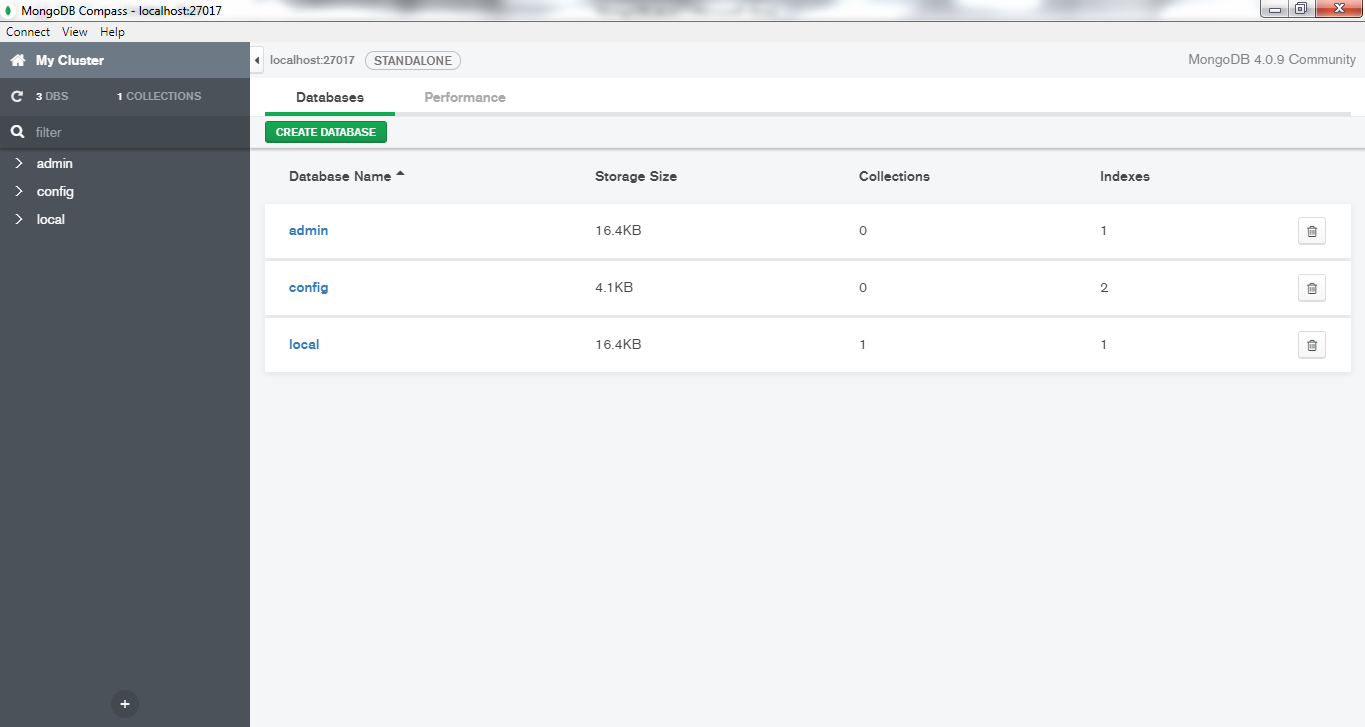
1. Open the command prompt and execute a command “**mongod**” daemon service which is nothing but MongoDB server. It should look like below,



1. Open MongoDB Compass (installed in step 2), accept all the default configuration and you will be shown below screen and click on “**Connect**”.

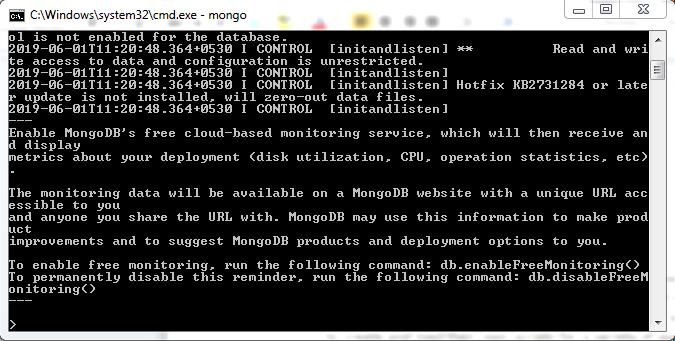


1. Final screen should look like below, with default databases created by MongoDB.



1. MongoDB is distributed with a simple but powerful tool called the mongo shell. The mongo shell provides built-in support for administering MongoDB instances and manipulating data using the MongoDB query language. It is also a fully functional JavaScript interpreter which enables users to create and load their own scripts for a variety of purposes.

Fire the command prompt and execute the command “**mongo**”. Then we should see below.



Part I. Introduction to MongoDB

Chapter 1. Introduction

MongoDB is a powerful, flexible, and scalable general purpose database. It combines the ability to scale out with features such as secondary indexes, range queries, sorting, aggregations, and geospatial indexes.

**MongoDB** is a *document oriented* database, not a relational one.

At the heart of MongoDB is the *document*: an ordered set of keys with associated values.

Ex:

{"greeting”: "Hello, world!"

MongoDB’s document format is based on JSON, a popular scheme for storing arbitrary data structures. JSON is an acronym **for *JavaScript Object Notation***

Ease of Use

Designed to Scale

Rich with Features...

Indexing

Aggregation

Special collection and index types

File storage

Without Sacrificing Speed

Chapter 2. Getting Started

Basic concepts of MongoDB:

* A ***document***is the basic unit of data for MongoDB and is roughly equivalent to a row in a relational database management system (but much more expressive).
* ***Collection***can be thought of as a table with a dynamic schema.
* A single instance of MongoDB can host multiple independent *databases*, each of which can have its own collections.
* Every document has a special key, **"\_id"**, that is unique within a collection.

Documents:

At the heart of MongoDB is the *document*: an ordered set of keys with associated values. In JavaScript, for example, **documents** are represented as objects:

{"greeting”: "Hello, world!"}

This simple document contains a single key, "greeting", with a value of "Hello, world.Most documents will be more complex than this simple one and often will contain multiple key/value pairs:

The keys in a document are strings. Any UTF8 character is allowed in a key, with a few notable exceptions:

* Keys **must not contain the null** character. This character is used to signify the end of a key.
* The **. and $** characters have some special properties and should be used only in certain circumstance. In general, they should be considered reserved, and drivers will complain if they are used inappropriately.
* MongoDB is **type-sensitive**, for example, below 2 statements are different.

{"count”: **5**}

{"count”: **"5"**}

* MongoDB is **case-sensitive**, for example, below 2 statements are different.

{"**c**ount”: 5}

{"**C**ount”: 5}

* MongoDB **cannot contain duplicate keys**. For example, the following is not a legal document:

{

"greeting": "Hello, world!",

"greeting": "Hello, MongoDB!"

}

Collections

A ***collection***is a group of documents. If a document is the MongoDB analog of a row in a relational database, then a collection can be thought of as the analog to a table.

Collections have ***dynamic schemas***. This means that the documents within a single collection can have any number of different “shapes.” For example, both of the following documents could be stored in a single collection:

{"greeting”: "Hello, world!", "views": 3}

{"signoff": "Good night and good luck"}

Note that the 1st documents have different keys, different numbers of keys, and values of different types compared to second document.

**But it’s recommended to create multiple collections of similar shapes like user collection, posts collection etc.**

A collection is identified by its name. Collection names can be any UTF8 string, with a few restrictions:

* The empty string ("") is not a valid collection name.
* Collection names may not contain the character \0 (the null character) because this delineates the end of a collection name.
* You should not create any collections that start with *system.*, a prefix reserved for internal collections. For example, the *system.users* collection contains the database’s users, and the *system.namespaces* collection contains information about all of the database’s collections.
* User created collections should not contain the reserved character $ in the name. The various drivers available for the database do support using $ in collection names because some system generated collections contain it. You should not use $ in a name unless you are accessing one of these collections.

SUBCOLLECTIONS

Databases

In addition to grouping documents by collection, **MongoDB groups collections into *databases***. A single instance of MongoDB can host several databases, each grouping together zero or more collections. A database has its own permissions, and each database is stored in separate files on disk. A good rule of thumb is to store all data for a single application in the same database.

Like collections, databases are identified by name. Database names can be any UTF8 string, with the following restrictions:

* The empty string ("") is not a valid database name.
* A database name cannot contain any of these characters: /, \, ., ", \*, <, >, :, |, ?, $, (a single space), or \0 (the null character). Basically, stick with alphanumeric ASCII.
* Database names are case-sensitive, even on non-case-sensitive file systems. To keep things simple, try to just use lowercase characters.
* Database names are limited to a maximum of 64 bytes.

One thing to remember about **database names is that they will actually end up as files on your file system**. This explains why many of the previous restrictions exist in the first place. There are also several reserved database names, which you can access but which have special semantics. These are as follows:

***admin***

The *admin* database plays a role in authentication and authorization. In addition, access to this database is required for some administrative operations.

***local***

This database stores data specific to a single server. In replica sets, *local* stores data used in the replication process. The local database itself is never replicated.

***config***

Sharded MongoDB clusters use the *config* database to store information about each shard.

By concatenating a database name with a collection in that database you can get a fully qualified collection name called a ***namespace***. For instance, if you are using the *blog.posts* collection in the *cms* database, the namespace of that collection would be cms.blog.posts.

Getting and Starting MongoDB:

MongoDB is almost always run as a network server (mongod.exe) that clients(mongo shell or Compass) can connect to and perform operations on so, make sure to start a server by running **mongod** command.

Introduction to the MongoDB Shell

Run **mongo** command to start mongo shell.

The shell is a full featured JavaScript interpreter, capable of running arbitrary JavaScript programs. To illustrate this, let’s perform some basic math:

> Math.random()

0.029327299126642448

A MongoDB Client

Mongo shell also a standalone MongoDB client. On startup, the shell connects to the *test* database on a MongoDB server and assigns this database connection to the global variable **db**.This variable is the primary access point to your MongoDB server through the shell.

For example,

> db

test

To select different database ( internally assign new database name to db global variable).

> use movies\_db

switched to db movies\_db

> db

movies\_db

Basic Operations with the Shell

CREATE

Create a local variable called movie that is a JavaScript object representing our document like below,

> movie = { "title":"Star Wars", "director":"George Locus", "year":1977 }

{ "title" : "Star Wars", "director" : "George Locus", "year" : 1977 }

Use **insertOne** method to insert to **movies** collection and to the database **movies\_db** (the one copied to db variable)

> db.**movies**.**insertOne**(movie);

{

"acknowledged" : true,

"insertedId" : ObjectId("5cf3c6eabc3a69b2b80780ea")

}

READ

**find** and **findOne** can be used to query a collection. If we just want to see one document from a collection, we can use findOne:

> db.movies.**findOne()**

{

"\_id" : ObjectId("5cf3fa87bc3a69b2b80780eb"),

"title" : "Star Wars",

"director" : "George Locus",

"year" : 1977

}

The **\_id** key & value is added by MongoDB & we can consider that as a **Primary Key**. The \_id will be auto generated, if we don’t supply one.

UPDATE

To modify our movie document, use **updateOne**. updateOne takes (at least) two parameters: the first is the criteria to find which document to update, and the second is the new document.

> db.movies.**updateOne({ "title":"Star Wars" },{ $set: { "director":"Nolan"} }**);

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }

> db.movies.findOne()

{

"\_id" : ObjectId("5cf3fa87bc3a69b2b80780eb"),

"title" : "Star Wars",

**"director" : "Nolan",**

"year" : 1977

}

DELETE

**deleteOne** and **deleteMany** permanently delete documents from the database. Both methods take filter document specifying criteria for removal.

> db.movies.deleteOne(**{ "title":"Star Wars" }**);

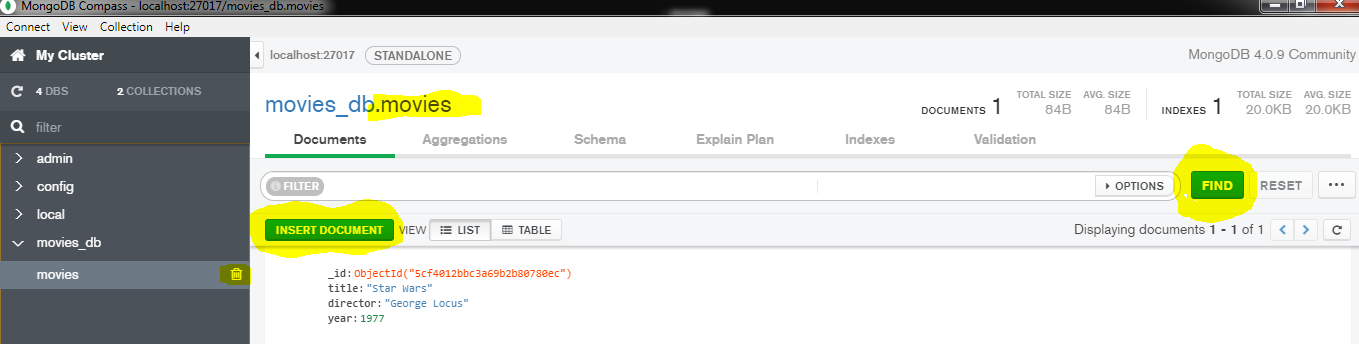
{ "acknowledged" : true, "deletedCount" : 1 }

> db.movies.findOne()

**null**

Use **deleteMany** to delete all documents matching a filter.

All above CURD operations can be done through MongoDB Compass too.



Data Types

Documents in MongoDB can be thought of as “JSON like” in that they are conceptually similar to objects in JavaScript. On the other hand, JSON’s expressive capabilities are limited because the only types are null, Boolean, numeric, string, array, and object.

For example, JSON has no date type, which makes working with dates even more annoying than it usually is. There is a number type, but only one—there is no way to differentiate floats and integers, never mind any distinction between 32-bit and 64- bit numbers. There is no way to represent other commonly used types, either, such as regular expressions or functions.

MongoDB adds support for a number of additional data types while keeping JSON’s essential key/value pair nature.

The most common types are:

**null**

Null can be used to represent both a null value and a nonexistent field:

{"x" : null}

**boolean**

There is a boolean type, which can be used for the values **true** and **false**:

{"x" : true}

**number**

The shell defaults to using **64bit floating point** numbers. Thus, these numbers look “normal” in the shell:

{"x" : 3.14}

Or

{"x" : 3}

**For integers**, use the **NumberInt** or **NumberLong** classes, which represent **4byte** or **8byte** signed integers, respectively.

{"x" : NumberInt("3")}

{"x" : NumberLong("3")}

**string**

Any string of UTF8 characters can be represented using the string type:

{"x" : "foobar"}

**date**

MongoDB stores dates as 64bit integers representing **milliseconds** since the Unix epoch

(January 1, 1970). **The time zone is not stored**:

{"x" : new Date()}

**regular expression**

Queries can use regular expressions using JavaScript’s regular expression syntax:

{"x" : /foobar/i}

**array**

Sets or lists of values can be represented as arrays:

{"x" : ["a", "b", "c"]}

**embedded document**

Documents can contain entire documents embedded as values in a parent document:

{"x" : {"foo" : "bar"}}

**object id**

An object id is a 12byte ID for documents.

{"x" : ObjectId()}

There are also a few less common types that you may need, including:

**binary data**

Binary data is a string of arbitrary bytes. It cannot be manipulated from the shell. Binary data is the only way to save non UTF8 strings to the database.

**code**

MongoDB also makes it possible to store arbitrary JavaScript in queries and documents.

{"x" : function() { */\* ... \*/* }}

Dates

In JavaScript, the Date class is used for MongoDB’s date type. **When creating a new Date object, always call new Date(...), not just Date(...).**

> Date()

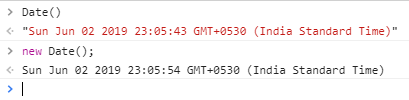
**Sun Jun 02 2019 23:07:26 GMT+0530 (India Standard Time)**

> new Date()

**ISODate("2019-06-02T17:37:34.478Z")**

**>**

**Or in Chrome Console**



The first one returns a string & second returns an object, 2nd one is preferred for MongoDB.

Arrays

In the following document, the key "things" has an array value:

{"things" : ["pie", 3.14]}

One of the great things about arrays in documents is that MongoDB “understands” their structure and knows how to reach inside of arrays to perform operations on their contents.

This allows us to query on arrays and build indexes using their contents. For instance, in the previous example, MongoDB can query for all documents where 3.14 is an element of the "things" array. If this is a common query, you can even create an index on the "things" key to improve the query’s speed.

Embedded Documents

Documents can be used as the *value* for a key.

{

"name" : "John Doe",

"address" : {

"street" : "123 Park Street",

"city" : "Anytown",

"state" : "NY"

}

}

In a relational database, the above document would probably be modeled as two separate rows in two different tables (one for “people” and one for “addresses”).

The flip side of this is that there can be more data repetition with MongoDB. Suppose “addresses” were a separate table in a relational database and we needed to fix a typo in an address. When we did a join with “people” and “addresses,” we’d get the updated address for everyone who shares it. With MongoDB, we’d need to fix the typo in each person’s document.

\_id and ObjectIds

Every document stored in MongoDB must have an "\_id" key. The "\_id" key’s value can be any type, but it defaults to an ObjectId.

In a single collection, every document must have a unique value for "\_id", which ensures that every document in a collection can be uniquely identified. That is, if you had two collections, each one could have a document where the value for "\_id" was 123. However, neither collection could contain more than one document with an "\_id" of 123.

OBJECTIDS

**ObjectIds** use 12 bytes of storage, which gives them a string representation that is 24 hexadecimal digits: 2 digits for each byte. This causes them to appear larger than they are, which makes some people nervous. It’s important to note that even though an ObjectId is often represented as a giant hexadecimal string, the string is actually twice as long as the data being stored.

Using the MongoDB Shell

Cutomization

Chapter 3. Creating, Updating, and Deleting Documents

CRUD API

Inserting Documents

To insert a single document, use the collection’s **insertOne** method.

> db.movies.insertOne({"title":"Stand by me", "\_id":"1234"})

{ "acknowledged" : true, "insertedId" : "1234" }

To insert multiple documents at once, use the **insertMany** method.

> db.movies.insertMany(**[**{"title":"Memento"},{"title":"Inception"},{"title":"Dunkirk"}])

{

"acknowledged" : true,

"insertedIds" : [

ObjectId("5cf5528bd490d55af15f9265"),

ObjectId("5cf5528bd490d55af15f9266"),

ObjectId("5cf5528bd490d55af15f9267")

]

}

This is far more efficient because your code will not make a round trip to the database for each document inserted, but will insert them in bulk.

Sending dozens, hundreds, or even thousands of documents at a time can make inserts **significantly faster**.

Current versions of MongoDB do not accept messages longer than **48 MB**, so there is a limit to how much can be inserted in a single batch insert. If you attempt to insert more than 48 MB, many drivers will split up the batch insert into multiple 48 MB batch inserts. Check your driver documentation for details.

What happens if 2nd document produces an error in-between while inserting many documents??

It depends on whether you have opted for **ordered** or **unordered** **operations**.

As the second parameter to insertMany you may specify options documents.

Specify **true** for the key, **"ordered"** in the options document to ensure documents are inserted in the order they are provided. Specify **false** and MongoDB may reorder the inserts to increase performance. **Ordered inserts is the default if no ordering is specified.**

* If a document produces an insertion error, no documents beyond that point in the array will be inserted.
* For unordered inserts, MongoDB will attempt to insert all documents, regardless of whether some insertions produce errors.

Ex:

db.movies.insertMany([

{"\_id" : 0, "title" : "Top Gun"},

{"\_id" : **1**, "title" : "Back to the Future"},

{"\_id" : **1**, "title" : "Gremlins"},

{"\_id" : 2, "title" : "Aliens"}])

2019-06-04T20:23:34.919+0530 E QUERY [js] BulkWriteError: write error at item 2 in bulk operation :

BulkWriteError({

"writeErrors" : [

{

"index" : 2,

"code" : 11000,

"errmsg" : "E11000 **duplicate key error collection**: movies\_db.movies index: \_id\_ dup key: { : 1.0 }",

"op" : {

"\_id" : 1,

"title" : "Gremlins"

}

}

],

"writeConcernErrors" : [ ],

"nInserted" : 2,

"nUpserted" : 0,

"nMatched" : 0,

"nModified" : 0,

"nRemoved" : 0,

"upserted" : [ ]

})

BulkWriteError@src/mongo/shell/bulk\_api.js:369:48

BulkWriteResult/this.toError@src/mongo/shell/bulk\_api.js:333:24

Bulk/this.execute@src/mongo/shell/bulk\_api.js:1173:1

DBCollection.prototype.insertMany@src/mongo/shell/crud\_api.js:314:5

@(shell):1:1

> ^

It has inserted only first 2 documents:

> db.movies.find()

{ "\_id" : **0**, "title" : "Top Gun" }

{ "\_id" : **1**, "title" : "Back to the Future" }

>

Let try with second parameter to insertMany(), i.e. **{ “ordered” : false}**

>db.movies.insertMany([

{"\_id" : 0, "title" : "Top Gun"},

{"\_id" : **1**, "title" : "Back to the Future"},

{"\_id" : **1**, "title" : "Gremlins"},

{"\_id" : 2, "title" : "Aliens"}], **{ “ordered” : false}**)

2019-06-04T20:31:34.495+0530 E QUERY [js] BulkWriteError: write error at item 2 in bulk operation :

BulkWriteError({

"writeErrors" : [

{

"index" : 2,

"code" : 11000,

"errmsg" : "**E11000 duplicate key error collection**: movies\_db.movies index: \_id\_ dup key: { : 1.0 }",

"op" : { "\_id" : 1,

"title" : "Gremlins"

}

}

],

"writeConcernErrors" : [ ],

**"nInserted" : 3,**

"nUpserted" : 0,

"nMatched" : 0,

"nModified" : 0,

"nRemoved" : 0,

"upserted" : [ ]

})

BulkWriteError@src/mongo/shell/bulk\_api.js:369:48

BulkWriteResult/this.toError@src/mongo/shell/bulk\_api.js:333:24

Bulk/this.execute@src/mongo/shell/bulk\_api.js:1173:1

DBCollection.prototype.insertMany@src/mongo/shell/crud\_api.js:314:5

@(shell):1:1

Let’s find out what happened in the db,

> db.movies.find()

{ "\_id" : 0, "title" : "Top Gun" }

{ "\_id" : 1, "title" : "Back to the Future" }

{ "\_id" : 2, "title" : "Aliens" }

>

It has inserted all documents except the error prone one.

Insert Validation

One of the basic structure checks is size: all documents must be smaller than **16 MB**.

To see the BSON size (in bytes) of the document *doc*, run **Object.bsonsize(*doc*)** from the shell.

> Object.bsonsize({"title":"Inception"})

26 bytes

insert()

In versions of MongoDB prior to 3.0, **insert()** was the primary method for inserting documents into MongoDB.

While methods such as insert() are still supported for backward compatibility, they should not be used in applications going forward.

Removing Documents

To delete the document(s), the MongoDB CRUD API provides **deleteOne** and **deleteMany.** Both of these methods take a filter document as their first parameter.

Ex:

> db.movies.find();

{ "\_id" : 0, "title" : "Top Gun" }

{ "\_id" : 1, "title" : "Back to the Future" }

**{ "\_id" : 2, "title" : "Aliens" }**

> db.movies.deleteOne(**{"\_id":2**});

{ "acknowledged" : true, **"deletedCount" : 1** }

> db.movies.find();

{ "\_id" : 0, "title" : "Top Gun" }

{ "\_id" : 1, "title" : "Back to the Future" }

>

We can also specify a filter that matches multiple documents in a collection. In these cases, deleteOne will **delete the first document found that matches the filter**.

**Note:** Which document is found first depends on several factors including the order in which documents were inserted, what updates were made to documents (for some storage engines), and what indexes are specified.

To delete more than one document matching a filter, use **deleteMany**.

> db.movies.find()

{ "\_id" : 0, "title" : "Top Gun", "year" : 1986 }

{ "\_id" : 1, "title" : "Back to the Future", "year" : 1985 }

{ "\_id" : 3, "title" : "Sixteen Candles", "year" : 1984 }

{ "\_id" : 4, "title" : "The Terminator", "year" : 1984 }

{ "\_id" : 5, "title" : "Scarface", "year" : 1983 }

> db.movies.deleteMany({ "year":1984});

{ "acknowledged" : true, **"deletedCount" : 2** }

> db.movies.find()

{ "\_id" : 0, "title" : "Top Gun", "year" : 1986 }

{ "\_id" : 1, "title" : "Back to the Future", "year" : 1985 }

{ "\_id" : 5, "title" : "Scarface", "year" : 1983 }

>

drop()

It is possible to use deleteMany to remove all documents in a collection.

> db.movies.find()

{ "\_id" : 0, "title" : "Top Gun", "year" : 1986 }

{ "\_id" : 1, "title" : "Back to the Future", "year" : 1985 }

{ "\_id" : 5, "title" : "Scarface", "year" : 1983 }

> db.movies.deleteMany(**{}**)

{ "acknowledged" : true, "deletedCount" : 3 }

> db.movies.find()

>

Removing documents is usually a fairly quick operation; However, if you want to clear an entire collection, **it is faster to *drop* it** and then recreate any indexes on the empty collection.

> db.movies.drop()

true

> db.movies.find()

>

Updating Documents

Documents can be updated in 3 ways,

updateOne

updateMany

replaceOne

updateOne and updateMany each take a filter document as their first parameter and as a second parameter the modifier document, which describes changes to make.

replaceOne takes a filter as the first parameter, but as the second parameter replaceOne expects a document with which it will replace the document matching the filter.

Document Replacement