**Introduction to MongoDB: Part 2**

**Contents**

[**1. Further MongoDB 1**](#_g0x0uh31n4gn)

[1.1 Import twitter data 1](#_esnv7hh4uuvx)

[1.2 Start MongoDB 6](#_h5bd8vizrum9)

[1.2 Explore the documents 7](#_webwyzhps68y)

[1.3 Search for specific field value 8](#_qw48lneqvt3w)

[1.4 Filter Fields returned by query 8](#_b5ol66cvzyn5)

[1.5 Pattern Matching 9](#_245vhtlynyf2)

[1.5.1 $regex 10](#_qlj616ijp77g)

[1.5.2 Regular Expression Options 11](#_33hp0rmlleex)

[1.6 $regex vs. /pattern/ Syntax 12](#_ixz4afx5sqo0)

[1.6.1 $in Expressions 12](#_b5rlxr7i0nom)

[1.6.2 x and s options 12](#_rwu3y85n0zq1)

[**2. Indexes 13**](#_jkf9w7rzpcm)

[2.1 Creating an Index 13](#_wctuaj74e0ho)

[2.2 Searching with an Index 13](#_4n948vqn7wvs)

[2.3 Show Indexes 13](#_1zq0gxkv5dzw)

[2.4 Dropping an Index 14](#_qlfdr610wdp8)

[**3. Further Queries 14**](#_ogzyilxrq82k)

[3.1 Operators 14](#_g6ht7wudnyp0)

[3.2 Null and Boolean data 15](#_rwbyna2vvm7a)

[3.3 Nested documents 15](#_70xx7gh4z205)

[3.4 Where Clause 15](#_2vja8hgg7g6w)

[**4. Group by Commands and Aggregation 16**](#_xz34lluumlr8)

[4.1 Further Examples on Aggregation (Orders collection) 17](#_dqp63fflvh11)

[4.1.1 Example 1 : Total Revenue for Each Pizza Type 18](#_wyivuvsoo5uk)

[4.1.2 Example 2 : Most Ordered Pizza Size 19](#_9xos41h614xh)

[4.1.2 Example 3 : Top 3 most Expensive Pepperoni Pizza Orders 20](#_e8mziori48uz)

[**5. Error Checking 21**](#_b8dcq6mc3zcz)

[**6. Cleaning Data 22**](#_towjbkgzpoe3)

[6.1 Updating One Document 22](#_kxcf5r8a6szf)

[6.2 Undefined Language 25](#_q4lnjxglbwp7)

[6.3 Nested Documents 26](#_e05pidyhexki)

[6.4 Updating Tweets 27](#_5bxrbqjwjlvp)

[6.5 Reshaping the Documents 27](#_mi0yawoztaff)

[**7. Reloading the Collection 29**](#_b7uqfhm7yt6)

[**8. Further Tweets 29**](#_ngdutdkpwvj)

[**9. Python Notebook 29**](#_5vmvnvfdxarm)

[9.1 PyMongo 30](#_8hrocnqg6jzr)

# Further MongoDB

This workbook assumes that you have carried out both tutorial and workshop tasks from the previous week.

For this workbook, we are going to use **twitter data (weather.json)** to carry out some guided tasks.

For this task you are to use **MongoDB Compass(Mongosh)** for writing the queries.

## 1.1 Import twitter data

MongoDB provides special tools to import/export data in/from MongoDB, which are

**MongoImport** and **MongoExport.**

In earlier versions, MongoImport and MongoExport were available with MongoDB

Server. But now they are available separately, so you should download them

separately.

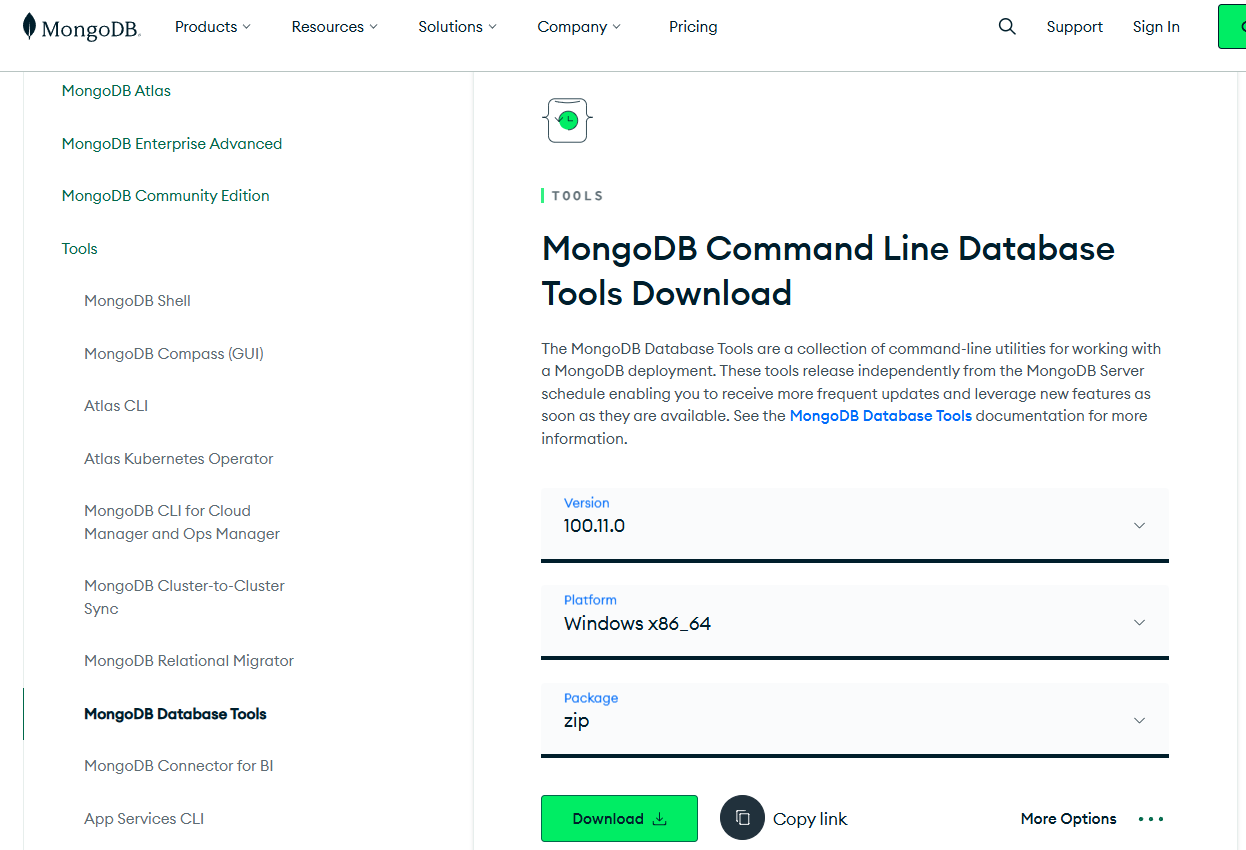
***Note:*** Always remember that MongoImport and MongoExport do not run in Mongo

Shell. You can directly use them in the command prompt (Windows) or terminal

(macOS).

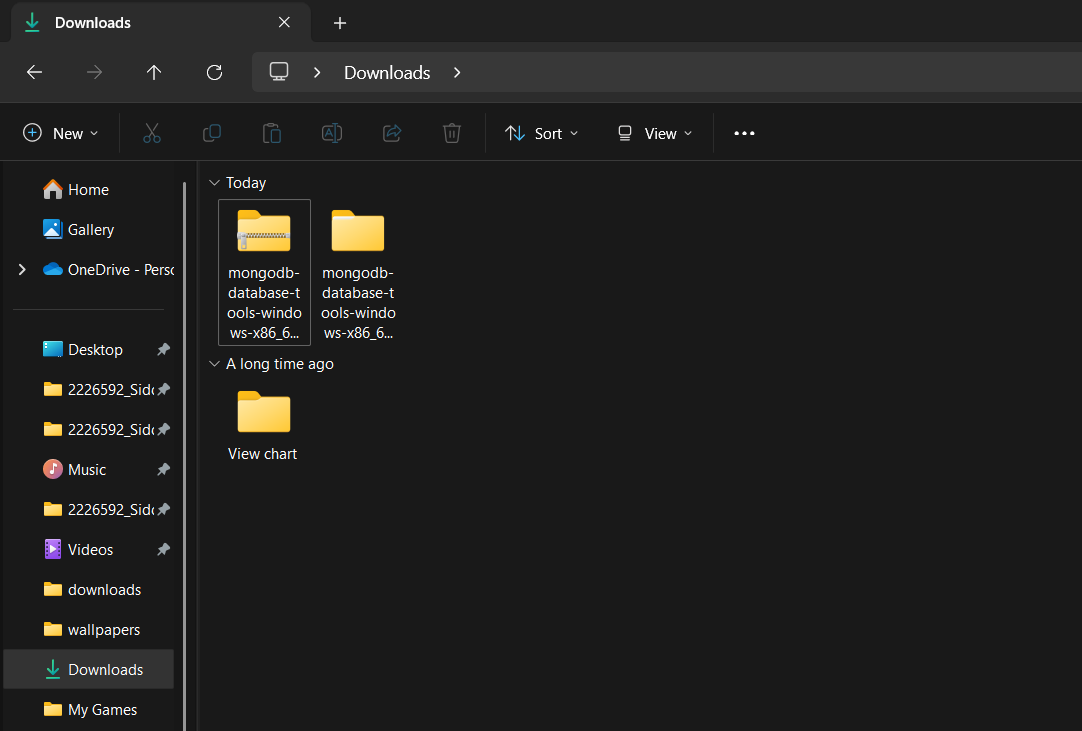
Follow the below given steps to **install MongoImport and MongoExport:**

1. Open the browser and go to the link: [Download MongoDB Command Line Database Tools](https://www.mongodb.com/try/download/database-tools)
2. Go to “**MongoDB Command Line Database Tools Download**”.

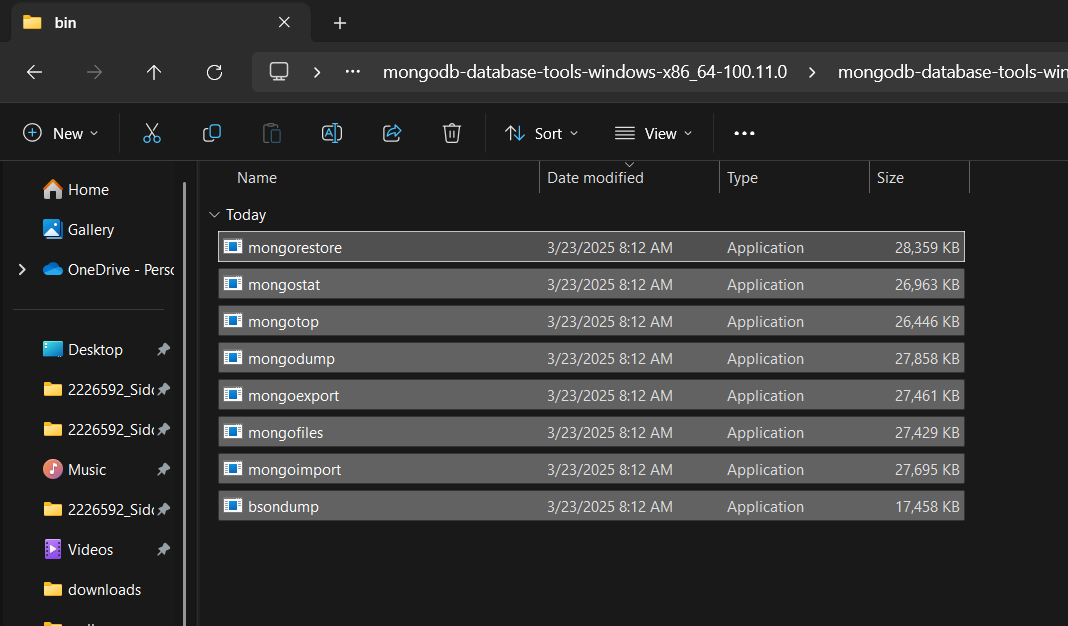


Now, you can choose the version, platform, and package.

1. After downloading, **unzip/extract** the file.

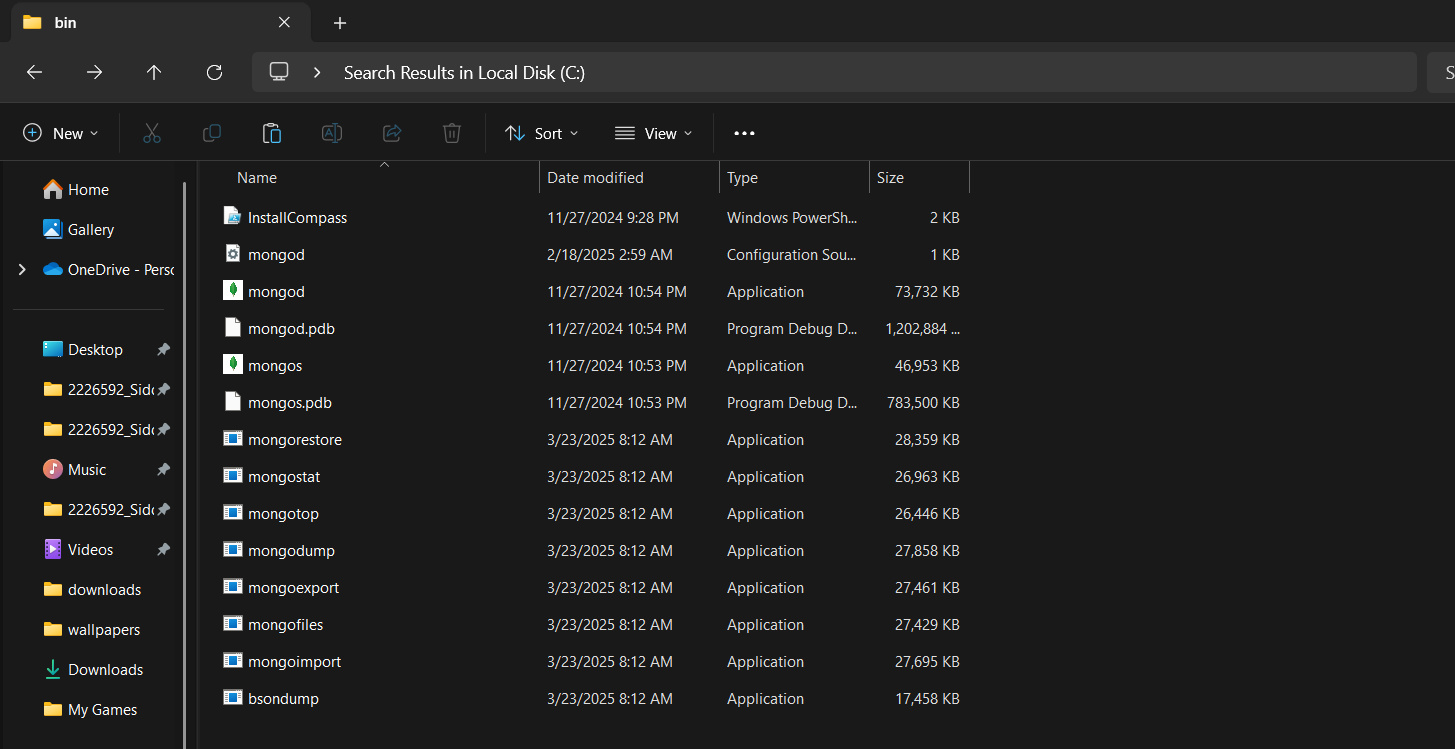


1. Now go to the bin folder of the unzipped file and **copy all .exe files**.



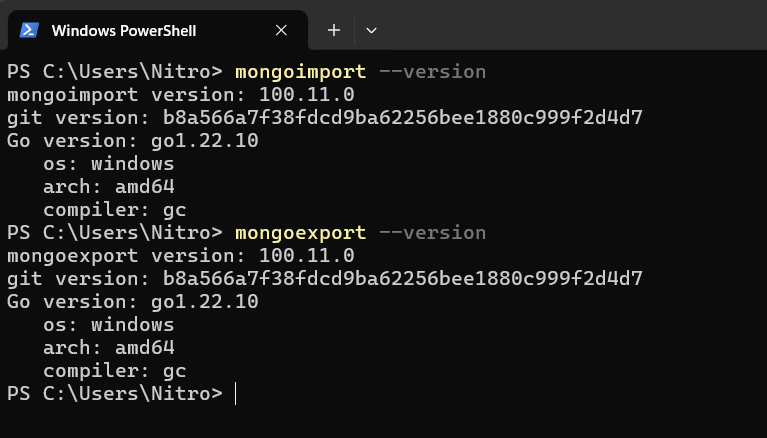
1. Paste all the copied .exe files into the **MongoDB bin folder**.

*My MongoDB bin folder path :* ***C:\Program Files\MongoDB\Server\8.0\bin***

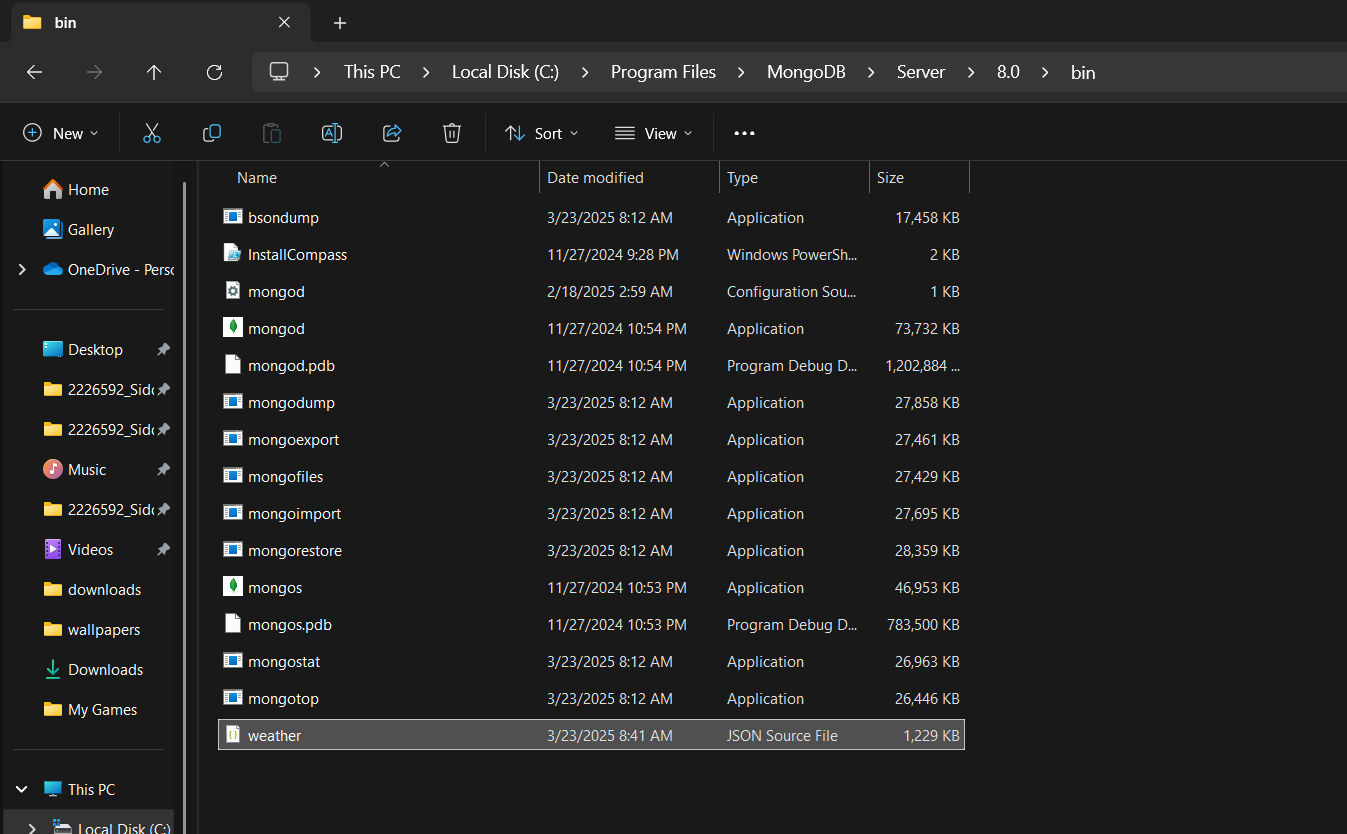


***Note :*** Your MongoDB bin folder path may differ.

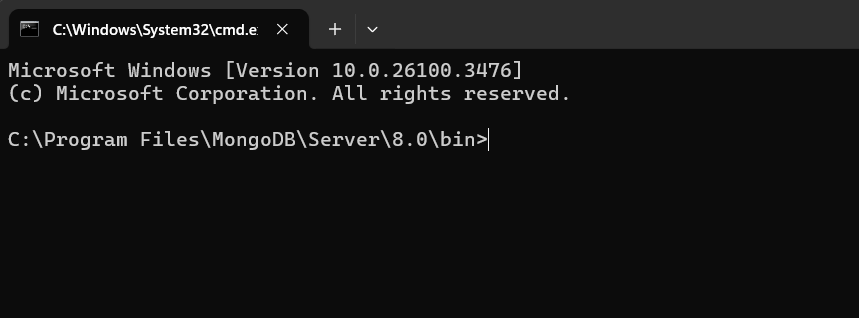
1. Open **command prompt (cmd)** and check if both mongoimport and mongoexport has been successfully installed or not, by running the below commands:



1. After successful installation, now you need to **download twitter data (weather.json)** provided to you from the shared drive.
2. Copy the downloaded **twitter data (weather.json file)** into the **MongoDB bin folder.**



1. Now from the **same bin folder** open the command prompt (cmd). Your **current working directory** should look something like this :

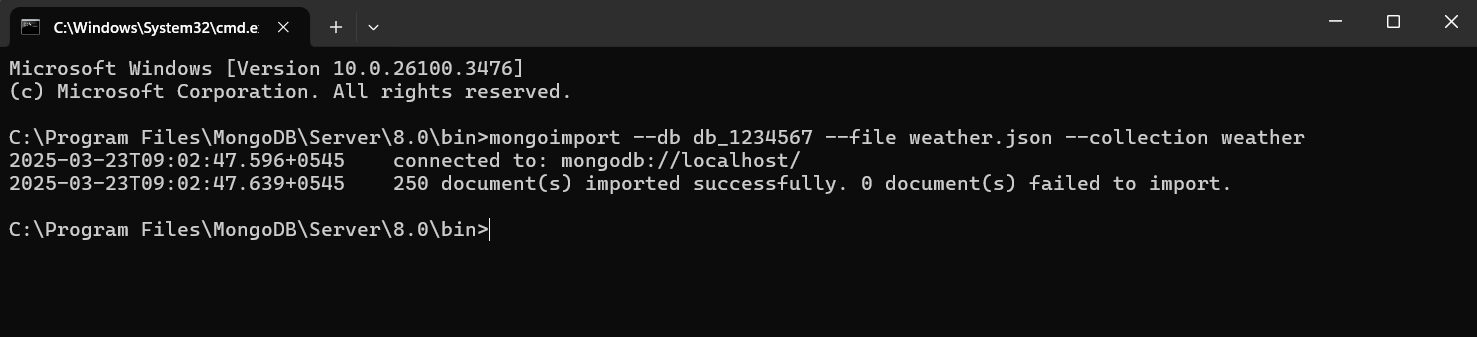


1. Now you need to use the below ***mongoimport*** command first to load the twitter data.

***Note :*** Name your database on the basis of your **uni id**. Example: if your id is **1234567** your database name should be **db\_1234567**.

|  |
| --- |
| mongoimport --db **db\_1234567** --file weather.json --collection weather |

Your output will look something like this :

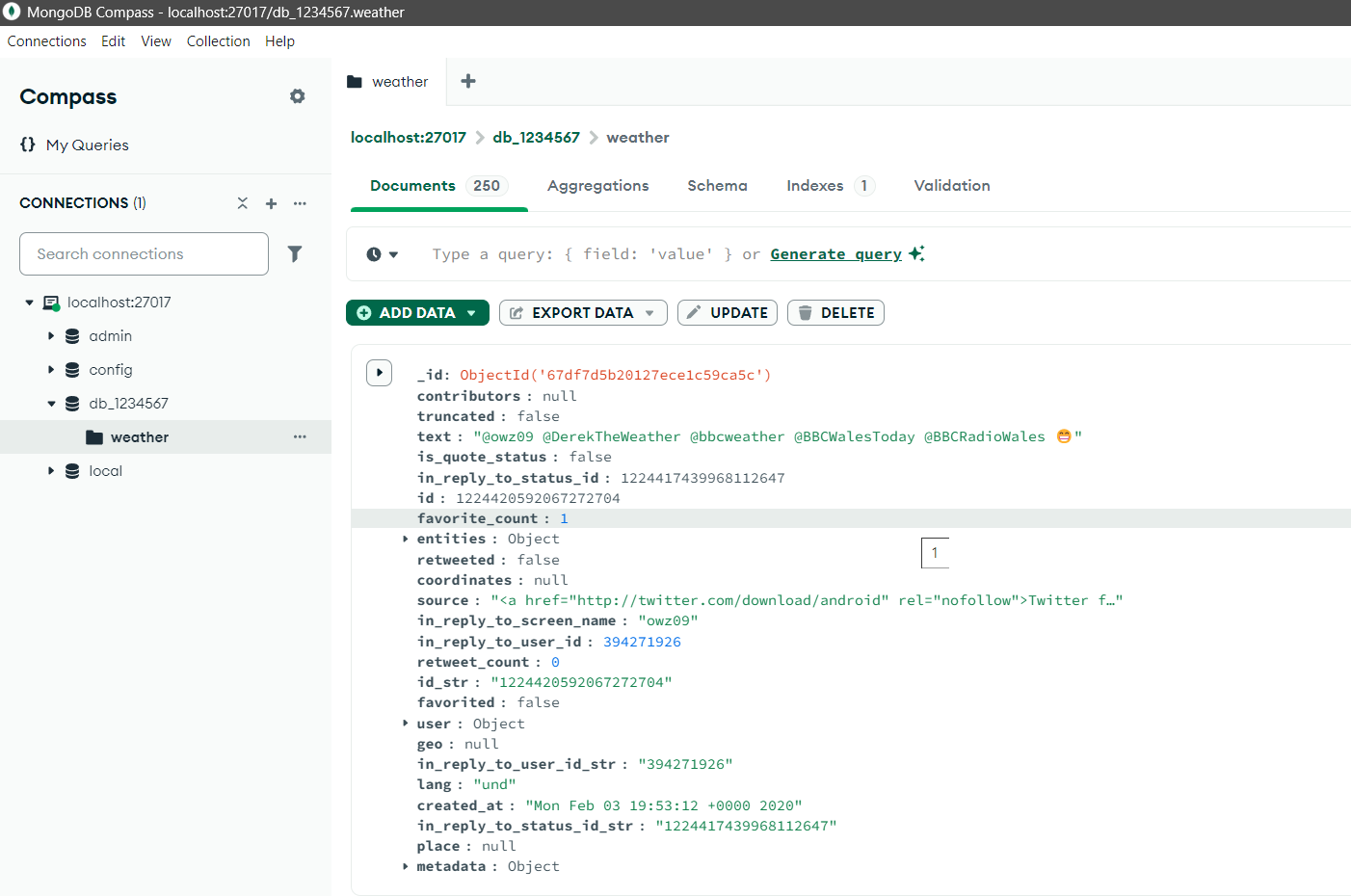


My cmd:

A screen shot of a computer

AI-generated content may be incorrect.

1. Finally, open your **MongoDB Compass** to see if the data has been imported to your newly created database.



My cmd:

A screenshot of a computer

AI-generated content may be incorrect.

## 1.2 Start MongoDB

First, switch to the database you previously created on the basis of your uni id from the

mongosh :

|  |
| --- |
|  |

***Note :*** Remember to type in your own university id while switching to the database you

created.

Check that the film tweets have been imported :

|  |
| --- |
| show collections |

A black background with white text

AI-generated content may be incorrect.

A black background with white text

AI-generated content may be incorrect.

The twitter data is stored in the weather collection. We can run the below command to count the number of documents:

|  |
| --- |
| db.weather.countDocuments() |

A black background with white text

AI-generated content may be incorrect.

## 1.2 Explore the documents

The twitter data is stored in the weather collection. We can run the below command to count the number of documents:

|  |
| --- |
| db.weather.findOne() |



A screenshot of a computer screen

AI-generated content may be incorrect.

The document is quite complex and has several fields, such as: name, retweet\_count, tweet\_ID, etc.. There are also nested fields under user, e.g., screen\_name, friends\_count, location, etc.

We can find the distinct values for a specific field by using the distinct() command. For example, let's find the distinct values for lang (language) :

|  |
| --- |
| db.weather.distinct("lang") |

A black background with white text

AI-generated content may be incorrect.

or find what the text of the tweets say :

|  |
| --- |
| db.weather.distinct("text") |

 A screenshot of a computer screen

AI-generated content may be incorrect.Finding unique values within a nested document needs the use of the dot notation, for example, find a user’s name:

|  |
| --- |
| db.weather.distinct("user.name") |

Some documents can have several layers of nesting, which can be accessed using the dot notation:

|  |
| --- |
| db.weather.distinct("extended\_entities.media.type") |

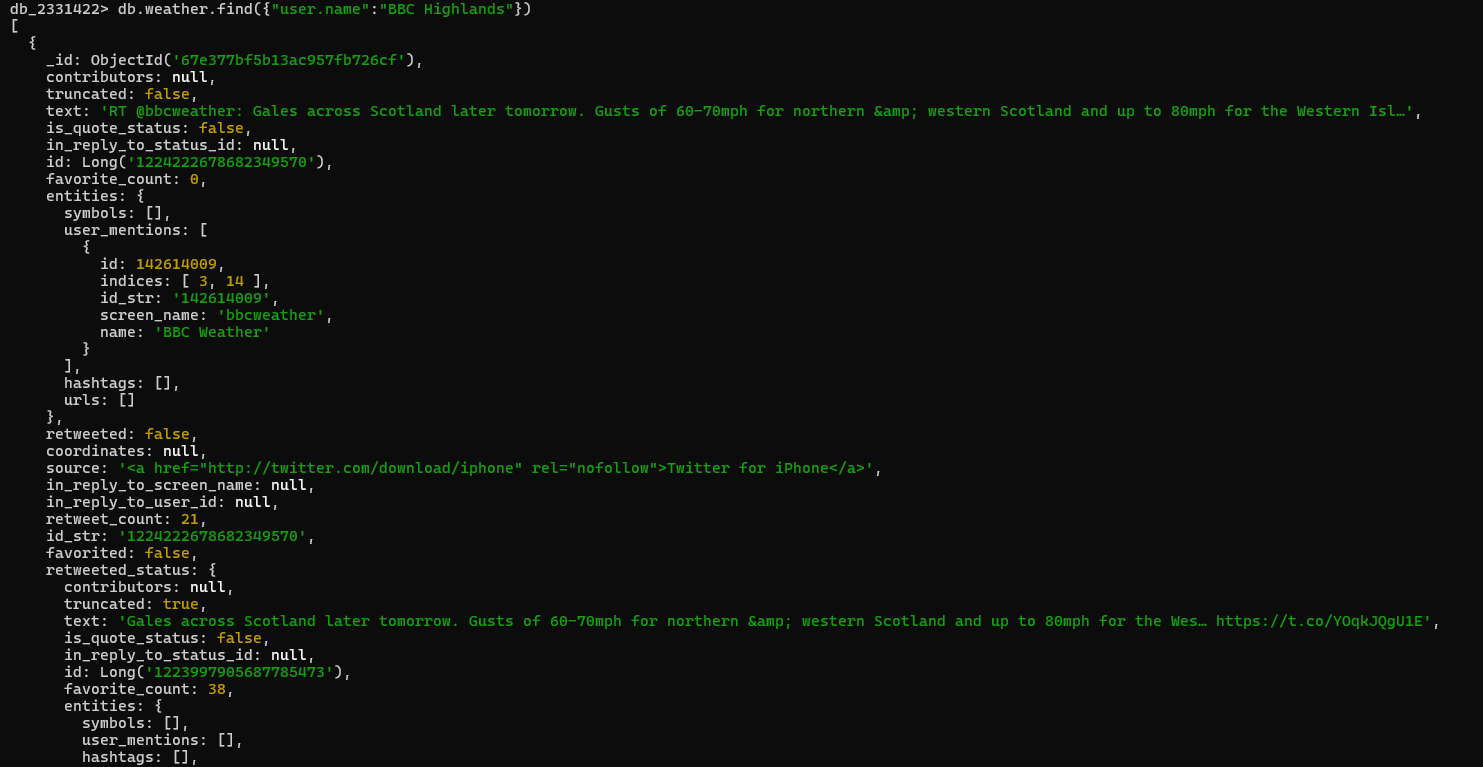
A screen shot of a video

AI-generated content may be incorrect.

## 1.3 Search for specific field value

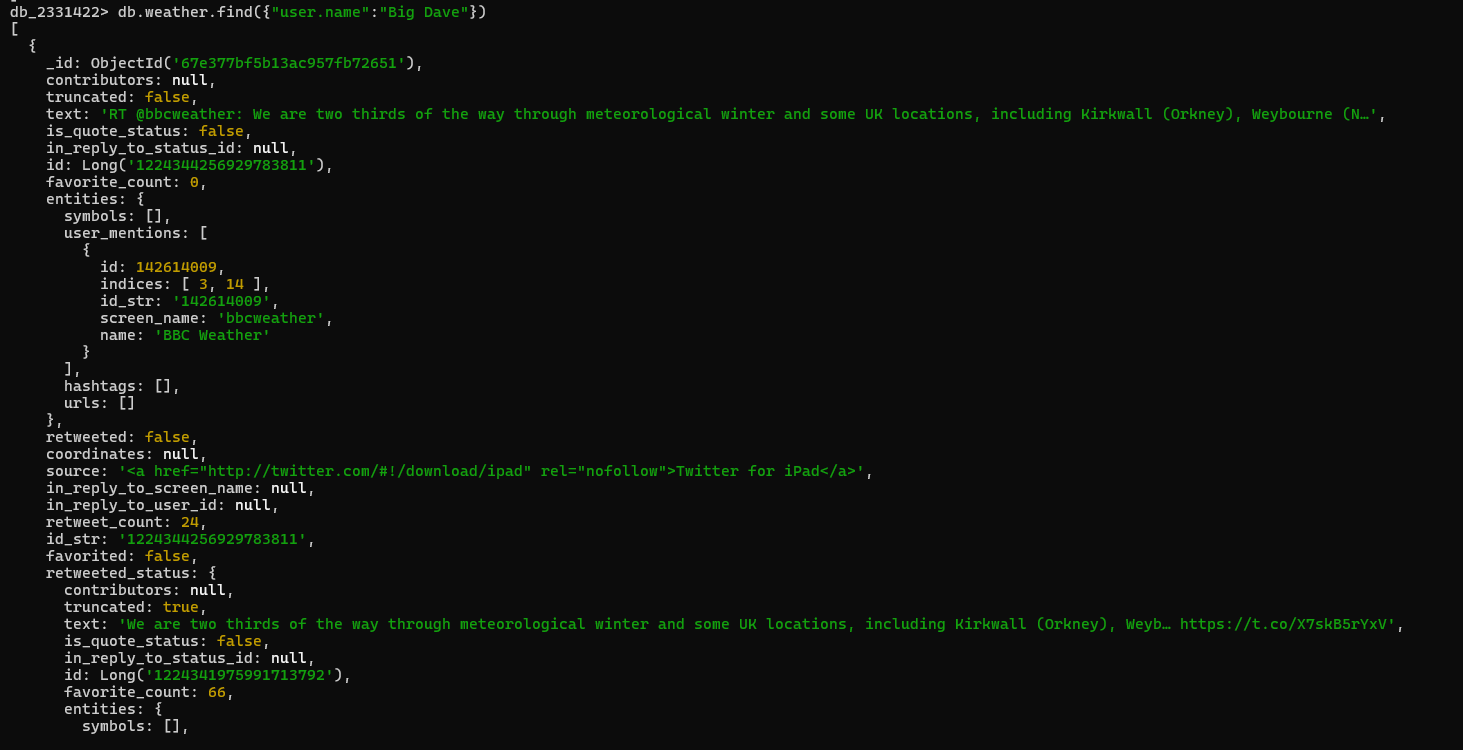
We can search for fields with a specific value using the find() command. For example, pick one of the names seen in the previous command and search for their tweets. For example:

|  |
| --- |
| db.weather.find({"user.name":"BBC Highlands"}) |

 A screenshot of a computer program

AI-generated content may be incorrect.

* Choose another user to view their tweets.



* Try viewing a different nested field, such as their screen\_name.

A screenshot of a computer

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.

Please note:

* Apologies in advance if there are any swear or offensive words in the text – do remember this is data produced by Twitter and is outside our control. Steps were taken to remove anything obvious, but offensive data could have slipped through given it is a large dataset.

## 1.4 Filter Fields returned by query

We can specify a second argument to the find() command to only show specific field(s) in the result. Let's repeat the previous search, but only show the tweet\_ID field:

|  |
| --- |
| db.weather.find({"user.name":"BBC Highlands"}, {id: 1}).pretty() |

***Output :***

|  |
| --- |
|  |

A computer screen with green text

AI-generated content may be incorrect.

The \_id field is objectID for every document and will be different for each

user. We can remove it from the results with the following filter:

|  |
| --- |
| db.weather.find({"user.name":"BBC Highlands"}, {id: 1, \_id:0}).pretty() |

***Output :***

|  |
| --- |
|  |

***Note:*** Remember the fields names are case sensitive, so if you get a null

return, it could be because you have used the wrong case.

## 1.5 Pattern Matching

You may not always want to search for an exact matching value, instead looking for patterns in the data, similar to the LIKE command in SQL. MongoDB supports pattern matching by using regular expressions.

The format for a regular expression is:

{ <field>: /pattern/<options> }

For example, if we search for the value sun in the tweet’s text field, there

are no results:

|  |
| --- |
| db.weather.find({text: "sun"}) |

However, if we search using a regular expression, there are many results:

|  |
| --- |
| db.weather.find({text: /sun/}, {text: 1}) |

***A screenshot of a computer program

AI-generated content may be incorrect.Note:*** The quotes have gone, otherwise it will try to search for “/sun/”.

The difference between the queries is that the first searched for where the text field value was exactly equal to sun, and the second searched for where the field value contained sun.

We can append .count() to the command to count the number of results :

|  |
| --- |
| db.weather.find({text: /sun/}).count() |



How many included rain in the text instead :

|  |
| --- |
| db.weather.find({text: /rain/}).count() |

### 

**1.5.1 $regex**

$regex also provides regular expression capabilities for pattern matching strings in queries.

$regex supports the following syntax :

{ <field>: { $regex: /pattern/, $options: '<options>' } }

{ <field>: { $regex: 'pattern', $options: '<options>' } }

{ <field>: { $regex: /pattern/<options> } }

For example, the previous search for sun in the text field can also be written using $regex :

|  |
| --- |
| db.weather.find({ text: { $regex: /sun/ } }).count() |



### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 1.5.2 Regular Expression Options

Options can be added to the regular expression.

These include:

|  |  |
| --- | --- |
| **i** | Case insensitivity to match upper and lower cases. |
| **m** | For patterns that include anchors (i.e. ^ for the start, $ for the end), match at the beginning or end of each line for strings with multiline values. Without this option, these anchors match at the beginning or end of the string.  If the pattern contains no anchors or if the string value has no newline characters (e.g. \n), the m option has no effect. |
| **x** | “Extended” capability to ignore all white space characters in the $regex pattern unless escaped or included in a character class. |
| **s** | Allows the dot character (i.e. .) to match all characters including newline characters. |

The previous query could be missing results if “sun” was actually written as “SUN”, “Sun” or other case variations. To make the search case insensitive use the i option too:

|  |
| --- |
| db.weather.find({ text: { $regex: /sun/i } }).count() |

 This time the count should be higher.

The above queries will find the value anywhere in the text field. If we want to search for

documents that start with a certain value, then use the start anchor, which is a claret (^).

For example, search for text values that are retweets (start with RT):

|  |
| --- |
| db.weather.find({ text: { $regex: /^RT/i } }).count() |

 The same can be done for documents that end with a certain value, this time using the dollar sign ($). For example, show all texts ending in snow :

|  |
| --- |
| db.weather.find({ text: { $regex: /snow$/i } }) |

## 1.6 $regex vs. /pattern/ Syntax

Depending on what type of query you are asking will determine whether you must use $regex or the /pattern/ syntax. In many cases either can be used, but some certain circumstances will dictate which must be used.

### 1.6.1 $in Expressions

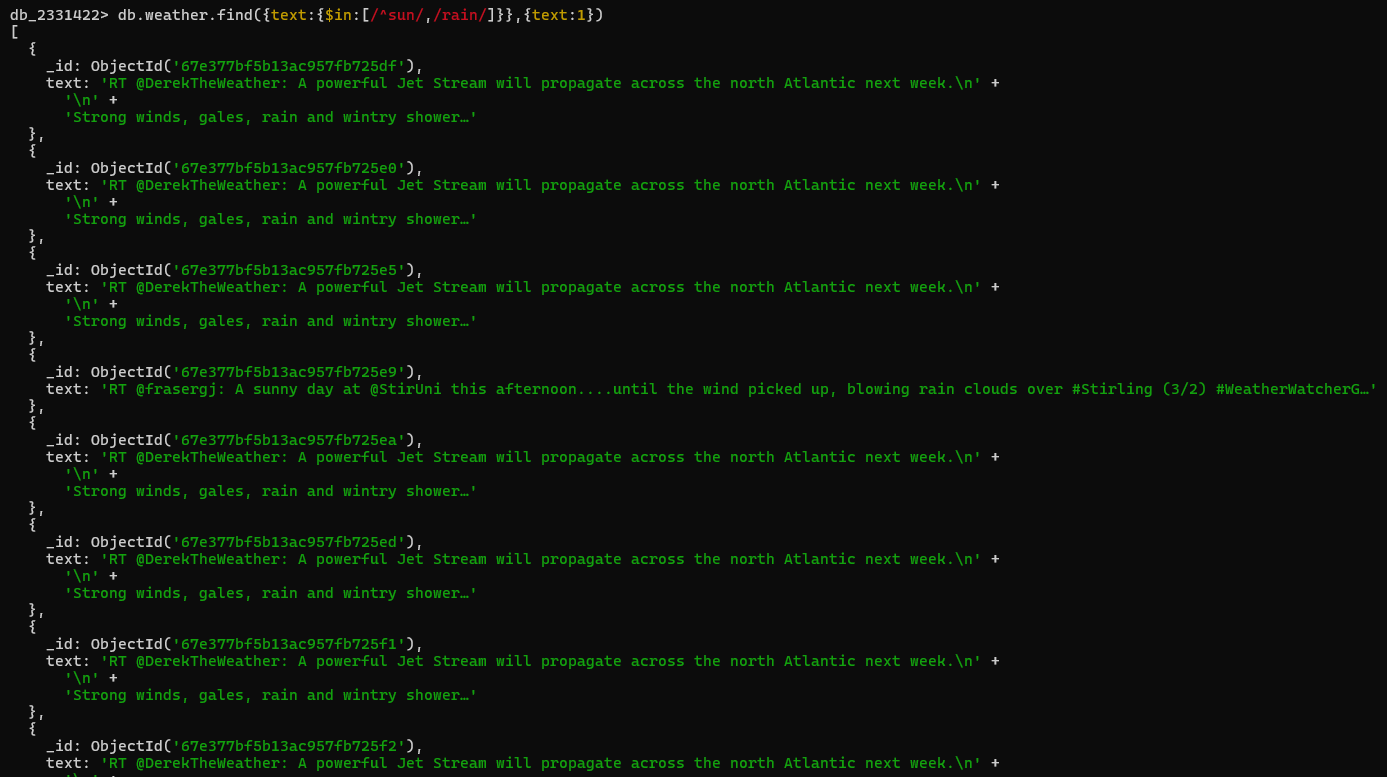
To include a regular expression in an $in query expression, you can only use JavaScript regular expression objects (i.e. /pattern/ ). $in is similar to the IN clause in SQL, where you can search for a value that matches any value in the list.

The syntax is:

{ field: { $in: [<value1>, <value2>, ... <valueN> ] } }

For example, find any texts that starts with sun in any case or contains rain:

|  |
| --- |
| db.weather.find({ text: { $in: [/^sun/i, /rain/] } }, {text: 1} ) |



### 1.6.2 x and s options

To use either the x option or s options, you must use the $regex operator expression with the $options operator. In such cases $options must also be used for i and m too.

For example, find any text that contains cold and night:

|  |
| --- |
| db.weather.find(  { text: { $regex: /cold.\*night/, $options: "si" } },  {\_id:0, text:1}  ) |

To summarise $regex cannot be used for $in queries, but must be used if using the i or m options.

A black screen with red text

AI-generated content may be incorrect.

# Indexes

A text index can be created to speed up searches and allows advanced searches with the $text query operator. Further details about $text can be found here:

👉 [$text (Self-Managed Deployments) - MongoDB Manual v8.0](https://docs.mongodb.com/manual/reference/operator/query/text/)

## 2.1 Creating an Index

Let's create the index using createIndex():

|  |
| --- |
| db.weather.createIndex({text: "text"}) |

The first argument text specifies the field on which to create the index, the second specifies that this is a text field (coincidentally the field we are indexing on is also called text).

## 2.2 Searching with an Index

Next, we can use the $text operator to search the collection. We can perform the previous query to find the documents containing snow:

|  |
| --- |
| db.weather.find({$text : {$search : "snow"}}).count() |

We can also search for documents not containing a specific value. For example, let's search for documents containing snow, but not ice:

|  |
| --- |
| db.weather.find({$text : {$search : "snow -ice"}}).count() |



## 2.3 Show Indexes

The function getIndexes() can be used to check what indexes exist on a collection:

|  |
| --- |
| db.weather.getIndexes() |

A computer screen shot of a computer code

AI-generated content may be incorrect.

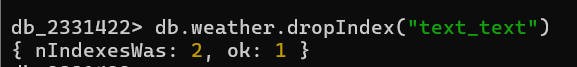
Make a note of the index name.

## 2.4 Dropping an Index

Indexes can also be dropped using the dropIndex() function.

For example, to drop the text index:

|  |
| --- |
| db.weather.dropIndex(‘text\_text’) |



or replace ‘text\_text’ by the name seen previously.

# 

# Further Queries

## 3.1 Operators

MongoDB can also search for field values matching a specific criteria. For example, we

can find where the retweet\_count is greater than 20:

|  |
| --- |
| db.weather.find(  {"retweeted\_status.retweet\_count": { $gt : 20 }}  ).count() |

A computer screen with green text

AI-generated content may be incorrect.

|  |  |
| --- | --- |
| **Operator Summary** | **Comment** |
| **$gt** | Searches for values greater than a specific value |
| **$lt** | Looks for values less than a given value |
| **$eq** | Equality search |
| **$gte** | Searches for values greater than, or equal to a specific value |
| **$lte** | Looks for values less than or equal to a given value |
| **$ne** | Not equals |

## 

## 

## 

## 3.2 Null and Boolean data

Missing data is represented by a null and Booleans by true/false values. For example,

count how many tweets have no user location defined:

|  |
| --- |
| db.weather.find({"user.location": { $eq : null }}).count() |

******

***Note:*** null is not in quotes, if you include quotes it will look for the value null, rather than

empty data.

How many users are geo\_enabled:

|  |
| --- |
| db.weather.find({"user.geo\_enabled": { $eq : true }}).count() |



How many users have 10 friends or less:

|  |
| --- |
| db.weather.find({"user.followers\_count": { $lte : 10 }}).count() |



## 3.3 Nested documents

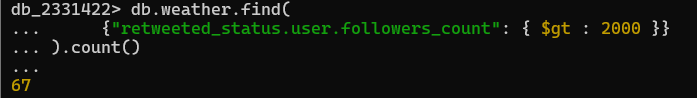
Note, syntax dictates that nested attributes have to be enclosed in double quotes, such

as retweet\_count which is a nested attribute of retweeted\_status.

There can be further nesting in a document, the dot notation can be used to traverse the

tree:

|  |
| --- |
| db.weather.find(  {"retweeted\_status.user.followers\_count": { $gt : 2000 }}  ).count() |



## 3.4 Where Clause

We can use the $where command to compare between fields in the same document. For

example, the following counts how many users have more followers than friends:

|  |
| --- |
| db.weather.find(  {$where :"this.user.followers\_count > this.user.friends\_count"}  ).count() |

***A black screen with green text

AI-generated content may be incorrect.Note:*** The field names for $where are required to be prefixed with this, which

represent the current document.

We can combine multiple searches by using $and. For example, let's count any tweets

containing ice where the retweet\_count is greater than fifty:

|  |
| --- |
| db.weather.find(  {$and : [ {text : /ice/},  { "retweeted\_status.retweet\_count":  {$gt: 50}}]}  ).count() |

A screen shot of a computer code

AI-generated content may be incorrect.

# Group by Commands and Aggregation

Carrying out statistical analysis of the data involves using the $group command and the

use of an aggregation pipeline. This is similar to the GROUP BY command in SQL.

For example, to produce the sum and count of all

|  |
| --- |
| db.weather.aggregate( [  {  $group: {  \_id : null,  totalAmount: { $sum: "$retweeted\_status.retweet\_count" },  count: { $sum: 1 }  }  }  ]) |

A screenshot of a computer code

AI-generated content may be incorrect.

See the MongoDB manual for further details:

👉 [Aggregation Stages - MongoDB Manual v8.0](https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/)

## 

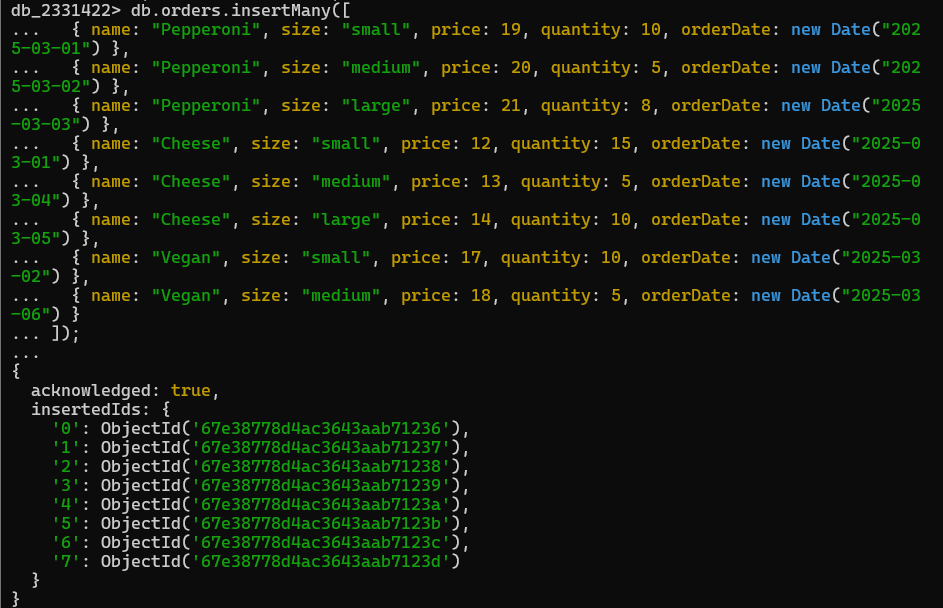
## 

## 

## 4.1 Further Examples on Aggregation (Orders collection)

Create a new collection (orders) :

|  |
| --- |
| db.**orders**.insertMany([  { name: "Pepperoni", size: "small", price: 19, quantity: 10, orderDate: new Date("2025-03-01") },  { name: "Pepperoni", size: "medium", price: 20, quantity: 5, orderDate: new Date("2025-03-02") },  { name: "Pepperoni", size: "large", price: 21, quantity: 8, orderDate: new Date("2025-03-03") },  { name: "Cheese", size: "small", price: 12, quantity: 15, orderDate: new Date("2025-03-01") },  { name: "Cheese", size: "medium", price: 13, quantity: 5, orderDate: new Date("2025-03-04") },  { name: "Cheese", size: "large", price: 14, quantity: 10, orderDate: new Date("2025-03-05") },  { name: "Vegan", size: "small", price: 17, quantity: 10, orderDate: new Date("2025-03-02") },  { name: "Vegan", size: "medium", price: 18, quantity: 5, orderDate: new Date("2025-03-06") }  ]); |



* The above code inserts multiple documents into the orders collection in the current/selected database.
* Each document represents an order with the following fields:
* **name** : Type of pizza ordered (e.g., "Pepperoni", "Cheese", "Vegan").
* **size** : Size of the pizza (e.g., "small", "medium", "large").
* **price** : Price of the pizza.
* **quantity** : Number of pizzas ordered.
* **orderDate** : Date the order was placed.

### 4.1.1 **Example 1 :** Total Revenue for Each Pizza Type

***Scenario :*** Calculate the total revenue generated for each pizza type.

|  |
| --- |
| db.orders.aggregate([  { $addFields: { totalRevenue: { $multiply: ["$price", "$quantity"] } } },  { $group: { \_id: "$name", totalRevenue: { $sum: "$totalRevenue" } } }  ]) |

***A screenshot of a computer code

AI-generated content may be incorrect.***

***Code Explanation :***

**First stage ($addFields)** : First, we calculated the total revenue for each

document by multiplying the price and quantity.

**Second stage ($group) :** Then, we group the documents by the name and

sum the total revenue.

***Output :*** It is a list of pizzas with their total revenue :

|  |
| --- |
|  |

### 

### 

### 

### 

### 

### 4.1.2 **Example 2 :** Most Ordered Pizza Size

***Scenario :*** Find out which pizza size was ordered the most.

|  |
| --- |
| db.**orders**.aggregate([  {  **$group**: {  \_id: "$size",  totalQuantity: { **$sum**: "$quantity" }  }  },  { **$sort**: { totalQuantity: -1 } },  { **$limit**: 1 }  ]) |

***Code Explanation :***

**First stage ($group)** : Groups by size and calculates the total quantity

ordered.

**Second stage ($sort)** : Sorts the results in descending order of quantity.

**Final stage ($limit)** : Returns only the most ordered pizza size.

***Output :***

|  |
| --- |
|  |

### 4.1.2 **Example 3 :** Top 3 most Expensive Pepperoni Pizza Orders

***Scenario :*** Find all orders containing "Pepperoni" and sort them by price in descending

order. Then, limit the results to the top 3 orders, and finally, project only the necessary

fields (name, price, and quantity).

|  |
| --- |
| # Try this yourself |

Your output will look something like this :

|  |
| --- |
|  |

# Error Checking

When you start using more complex commands you are more likely to get errors since

Most of them will involve a variety of brackets that need to be in the right order.

For example, if you typed in this for the last command:

|  |
| --- |
| db.weather.find(  {$and : [ {text : /ice/},  { "retweeted\_status.retweet\_count": {$gt: 50}}  }]).count() |

You would get something similar to this error :

|  |
| --- |
|  |

Why is this? If you count the brackets there appears to be the right number of opening and closing brackets, however look more carefully at the order they are in. There is a mixture of curly { }, square [ ] and round ( ) brackets. It is also important to get the order of them correct too!

Note the order of the final set is out of sync:

…{$gt: 50}}}]).count()

Should in fact be:

…{$gt: 50}}]}).count()

You can recall any last command in MongoDB by pressing the up arrow, then use the arrow keys to move to where you want to make any corrections. Press return when you have made the necessary changes.

If you find that you get 0’s for the counts and you were expecting some results, check that you have:

* Got the name of the attribute correct. Unlike relational databases, there is no schema for the database to check your query against, so it will not flag that the field name is wrong, it will assume it is just not found in the current set of documents.
* Same goes for the collection name and any other user defined names.
* Checking the nesting. Is the attribute you are interested in nested within a sub-document? For example, if the earlier query missed the retweet\_count nesting, it will just return 0:

|  |
| --- |
| db.weather.find(  { $and : [ {text : /box office/},  { "retweet\_count": {$gt: 50}}  ]}).count() |

# Cleaning Data

The documents generated by Twitter are quite complex and can be difficult to update.

When analysing data you may wish to get a consistent format to some documents, for

example, you may only be interested in the country of where the tweets came from,

rather than individual towns.

## 6.1 Updating One Document

By default, the update() method updates a single document, the multi parameter can be

set to true to update many, or use the updateMany() function.

See the manual for further details:

👉 <https://docs.mongodb.com/manual/reference/method/db.collection.update/>

👉 <https://docs.mongodb.com/manual/reference/method/db.collection.updateOne/>

👉 <https://docs.mongodb.com/manual/reference/method/db.collection.updateMany/>

When using the update() method with mu, the multi parameter set to false you cannot

guarantee which document will be updated. The only way to guarantee updating the

document you want is to use the Object Id: “\_id”.

***Note:*** everyone will have unique identifiers generated, so you will need to replace the

ObjectIds below with your own data.

Let's make the language code more user-friendly. See this page for the language

associated with each code:

👉 [Supported languages and browsers | Docs | Twitter Developer Platform](https://developer.twitter.com/en/docs/twitter-for-websites/twitter-for-websites-supported-languages/overview)

For example, en means English, which is the default language.

Let's change one of the languages to the full name, so for example, es would show as

Spanish. Check what languages are in your dataset:

|  |
| --- |
| db.weather.distinct("lang") |

Pick one of the languages and check the above website to see what the language

should be. Then find the documents with this language.

***Note:*** Replace **languageCode** with one of the values seen:

|  |
| --- |
| db.weather.find({ lang: **"languageCode"**}, {lang:1}) |

The results will be similar to below, but your Object Ids will be different:

|  |
| --- |
|  |

Pick one of your ObjectId values, e.g., **67de29a440e349f66b12dc57** seen above and

query just that document:

|  |
| --- |
| db.weather.find (  { \_id: ObjectId("**67de29a440e349f66b12dc57**")},  {lang: 1}  ) |

The results should be for a document for the language code given above.

If you get no results returned, check that you have entered a correct Id.

Using this Id, update the lang field so the code shows as the full name instead:

|  |
| --- |
| db.weather.update(  { \_id: ObjectId("**67de29a440e349f66b12dc57**") },  {$set: {'lang': '**languageName**'}}  ) |

Which should return :

|  |
| --- |
|  |

Since we have not set the multi option, only one document will be updated, but given we

have specified the ObjectId, which is unique, it should only update one documen

anyway.

Check the update has worked :

|  |
| --- |
| db.weather.find (  { \_id: ObjectId("**67de29a440e349f66b12dc57**")},  {lang: 1}  ) |

And the response should now show the new language name:

|  |
| --- |
|  |

## 6.2 Undefined Language

You may find that you have a language called “und” in your list. This is where the

language is unknown or underdetermined. To make this clearer, lets change und to say

unknown instead for all documents:

|  |
| --- |
| db.weather.updateMany({ lang: “und” }, {$set: {'lang': 'unknown'}}) |

This time many documents will be changed:

|  |
| --- |
|  |

## 6.3 Nested Documents

Nested documents can also be updated using the dot notation. For example, update the

collection so all locations containing England, just say UK, removing any town or county

information:

|  |
| --- |
| db.weather.updateMany(  { 'user.location': {$regex:'England', $options: 'i' }} ,  {$set: {'user.location': “UK”}}  ) |

Remember nested fields must be in quotes. This time updateMany() was used, so the

response should show several documents have been updated:

|  |
| --- |
|  |

To do :

* Write some code to check this has updated as expected.
* Check what other locations are in the collection and update any documents where the user’s location includes the value UK to just say UK.

## 6.4 Updating Tweets

Tweets can contain information you may want to update to be consistent, e.g., if

someone has used abbreviations and you want all tweets to say the same time. For

example, find all tweets that contain pic:

|  |
| --- |
| db.weather.find({'text': {'$regex':'pic', '$options': 'i' } }, {'text':1}) |

Update this so it says picture instead, this involves matching substrings:

|  |
| --- |
| db.weather.find({ 'text': { '$regex': 'pic' } }).forEach(function(e) {  e.text = e.text.replace("pic", "picture");  db.weather.replaceOne({ \_id: e.\_id }, e);  }); |

This code finds all documents in the weather collection where the text field contains the

substring "pic". For each matching document, it replaces "pic" with "picture" in the text

field and updates the document using replaceOne(). replaceOne() replaces the entire

document with the modified version, keeping the \_id unchanged.

Use of the function is based on information from this page:

👉 [How to replace substring in mongodb document - Stack Overflow](https://stackoverflow.com/questions/12589792/how-to-replace-substring-in-mongodb-document)

***Note:***

* Currently the function has no equivalent in Python.
* Tweets can contain Unicode characters to represent icons, etc. How they are represented can depend on how you view the data. For example, putty may show the characters as square boxes, whilst a Python notebook might show the Unicode character, or the icon.

## 6.5 Reshaping the Documents

You might only be interested in certain fields for your analysis and want create a

collection with just the required data. An aggregation pipeline using $project can be used

to pick out the required columns and the results saved to a new collection.

For example, say you just want to keep the information that shows the user’s location

and the text:

|  |
| --- |
| db.weather.find( { }, {"user.location":1, text:1}) |

To save these details into a new collection called projWeather:

|  |
| --- |
| db.weather.aggregate([  {$project:{"user.location":1, text:1}},  {$out:"projWeather"}  ]) |

Check that the new collection exists with the required data:

|  |
| --- |
| db.projWeather.find().pretty() |

You can also reduce the number of rows in a collection. This could be done with a query

that is based on some criteria, or if you just want to get a subset of the data, not based

on any particular criteria you can use an aggregation pipeline using $limit to restrict the

number of rows.

The syntax is:

{ $limit: <positive integer> }

For example, just keep 50 rows from the projWeather collection:

|  |
| --- |
| db.projWeather.aggregate([  {$limit: 50},  {$out:"newWeather"}  ]) |

Check it does indeed only have 50 rows:

|  |
| --- |
| db.newWeather.countDocuments() |

Compared to projWeather:

|  |
| --- |
| db.projWeather.countDocuments() |

# Reloading the Collection

MongoDB is not so good at transaction support compared to relational databases; updates will be made with no option to rollback the changes if you change your mind.

If you want to return to the original data, one option is to drop the collection completely (***do not do this unless you want to remove the collection completely!***):

|  |
| --- |
| db.weather.drop() |

Then reload from the original weather.json file.

# Further Tweets

To obtain your own Twitter data you need to register for a Developer’s account. See this page for further details:

👉 [apply for a Twitter developer account](https://developer.twitter.com/en/apply-for-access.html)

# Python Notebook

The examples in this workbook use the MongoDB shell. A python notebook version of this workbook called **mongo-Weather-Template.ipynb** is also available to the students.

This part we are going to cover in our upcoming workshop (week 5) and some of you may find using the notebooks easier to understand the data.

The notebook can be used be on various Python Notebooks:

|  |  |
| --- | --- |
|  | Anaconda |
|  | If you have a Google account you can use Google Colaboratory found here:  👉 <https://colab.research.google.com/> |

**Important:**

*For the upcoming week-5 workshop you are required to run your python notebook on your device locally. So make sure you have a jupyter notebook installed on your computer.*

You may use the following links to set up **jupyter notebook** on your computer :

|  |  |
| --- | --- |
| **Windows** | [How to Install Jupyter Notebook on Windows - GeeksforGeeks](https://www.geeksforgeeks.org/install-jupyter-notebook-in-windows/) |
| **Ubuntu** | [How to Install Jupyter Notebook in Linux - GeeksforGeeks](https://www.geeksforgeeks.org/how-to-install-jupyter-notebook-in-linux/) |
| **MacOS** | [How to Install Jupyter Notebook on MacOS - GeeksforGeeks](https://www.geeksforgeeks.org/how-to-install-jupyter-notebook-on-macos/?ref=next_article_top) |

## 9.1 PyMongo

The python workbook makes use of the PyMongo API, which contains tools for working

with MongoDB. Further details about this API can be found here:

👉 [PyMongo](https://pymongo.readthedocs.io/en/stable/)