03. Big Data with MongoDB

Contents

1. MongoDB

Read *What is MongoDB* and *Introduction to MongoDB* for an overview.

MongoDB is a database management system (DBMS). It uses client-server architecture. On the server side there are two *processes*:

- mongod[1] — this does all the data stuff
- mongos — controls query routing for distributed configurations (this is achieved by replication and sharding in MongoDB)

An *interactive MongoDB shell* client, called *mongo* or *mongosh* (depending on the MongoDB version installed), comes with the DBMS. In this course, we will use a more learner-friendly client called Robo3T.

1.1 Installation advice

MongoDB community server edition is free for download and it is open source:

- Follow this guide for Windows
  - It is **not recommended** to install mongod as a service
  - Feel free to install and try out Compass but it is not used in this course
- Follow this guide for macOS
  - The Homebrew method is the easiest way to install MongoDB

After MongoDB is installed, you need to set up the folder where its data files will be created. Use the default locations below. Create empty folders in these locations:

Windows	macOS
`C:\data\db`	`/data/db`

https://coady.tech/macos-command-line-tools-outdated/

It is recommended you add MongoDB's binaries folder (/bin under your installation path) to your OS's "PATH" variable. This allows mongod, mongoimport, mongoexport, mongorestore, and mongodump to be executed from anywhere in Command Prompt (cmd) / Terminal.

- Windows: If you have not changed the default installation path, your binaries folder should be similar to "C:\Program Files\MongoDB\Server**4.4**\bin". This is the path you need to add to Windows' PATH system variable
- macOS: If you used the Homebrew installation method, it would have automatically added the binaries folder to $PATH in Terminal (i.e. you do not need to do anything else)

Robo3T is free for download and it is open source.

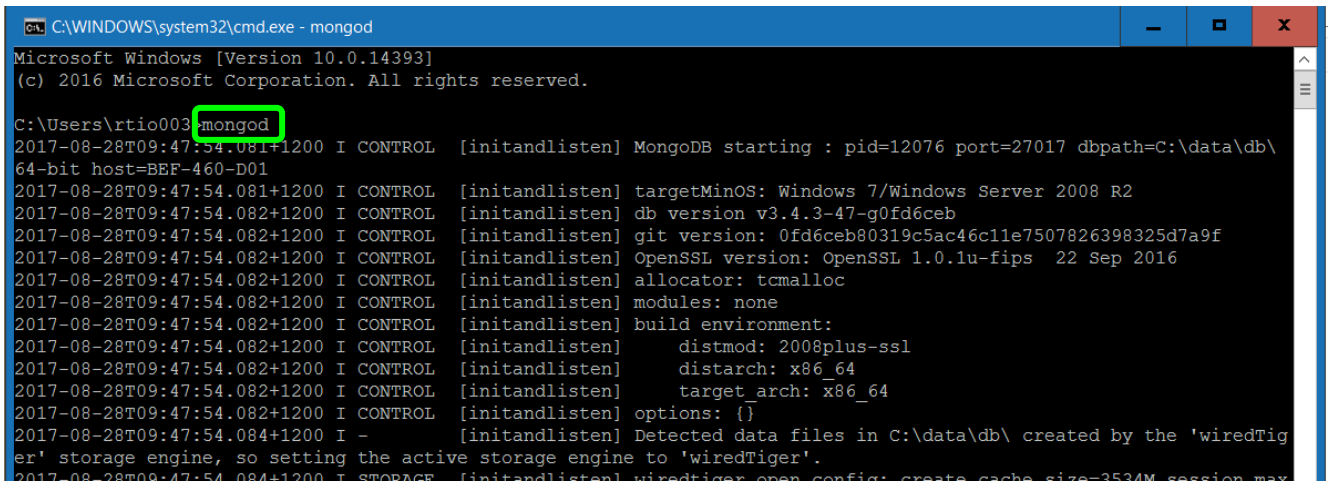
1.2 Server

Run the mongod process to start the MongoDB database server.

1.2.1 Windows

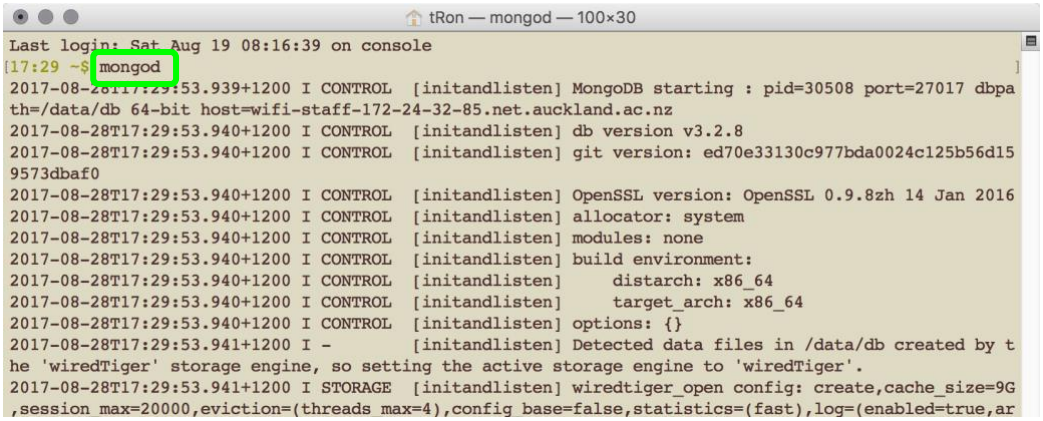
With a typical installation configuration and MongoDB's /bin folder added to your PATH, open command prompt and type "mongod" and press enter.

[1] The "d" means "daemon": https://en.wikipedia.org/wiki/Daemon_(computing). Daemon is pronounced "demon", and mongod is pronounced "mongo dee".



1.2.2 macOS

With MongoDB's /bin folder added to your PATH, open Terminal and type "mongod" and press enter.



1.2.3 Running mongod

This launches the mongod process with default parameters which are shown on the first line. Parameters to note are: port=27017 and dbpath=C:\data\db (Windows) or /data/db (macOS). You will need to know the port number to connect. The dbpath is where files to store data are created.

We will not configure any login or authentication settings for our use of MongoDB in this course. This should still be safe so long as port 27017 is not open on your computer's firewall (which it is not by default).

The MongoDB DBMS is up and running so long as the mongod process is running, so leave the cmd or Terminal window/tab opened while using MongoDB. Once you are done you can end the mongod process—the easiest way to do this is to press ctrl+c in the cmd or Terminal where mongod is running. Wait for it to exit completely lest you end up with corrupt data files.

1.3 Client

To connect to a running mongod process, use the mongoDB shell or Robo3T.

1.3.1 MongoDB shell

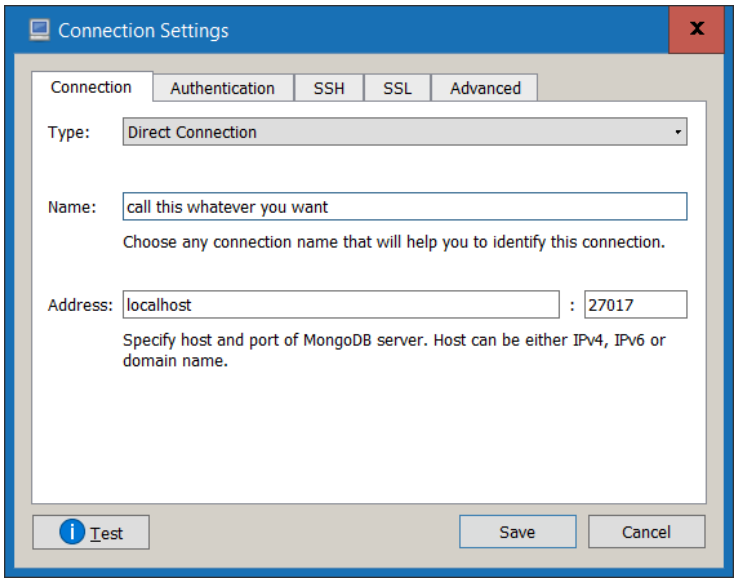
Open mongo or mongosh in cmd or Terminal. The default host is *localhost* and the default port is 27017, both of which match the default settings of the server, so it will connect to the server if it is running.

To end the session, press ctrl+c.

An introduction to the mongo shell can be found here, and the one for mongosh shell is here.

1.3.2 Robo3T

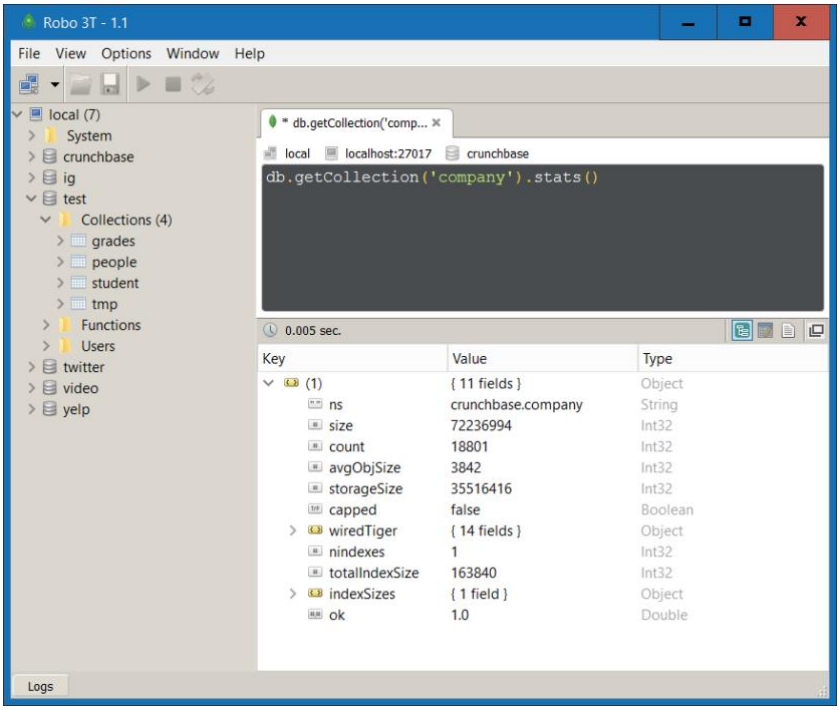
Connect to address *localhost* and port 27017.



Once connected you will see a list of databases on the left. This is a navigation menu for database objects (collections, indexes, etc.).

Right-clicking on a collection will provide some options such as viewing, inserting, or updating documents; and renaming, dropping, or seeing statistics on the collection.

In the "centre" of the window there is a space for tabs. Each tab has space for a query (where you type text), and the query's result (displayed as JSON text, a table, or tree view).



When you save a tab, it saves the plain text of the query.

Queries in Robo3T are in JavaScript. MongoDB's query language (MQL) is a JavaScript like language.

1.3.3 Other clients

There are many other MongoDB GUI clients for interactive querying or administrative uses. They are similar to Robo3T.

Another way to interact with MongoDB is by using code. This is achieved using MongoDB drivers. See the documentation. Drivers provide an API for interacting with a MongoDB server.

1.4 Data model

MongoDB, being a document store type of database, belongs to a family of NoSQL databases which use the *document* paradigm to store data.

MongoDB stores its data in **databases**, **collections**, **documents**, and **fields**. One server can have many databases; each database can have many collections; each collection stores many documents; each document is made up of fields containing values.

- Database: like a RDBMS, this is the *top level* object where data is stored
- Collection: comparable to a table. Collections store documents with fields of values (whereas tables store rows with columns of values)

Document is the unit for instances of data. The document structure is also used for queries.

The underlying storage format is a BSON (i.e. binary JSON) document. These are the datatypes supported by MongoDB (recall that JSON only supported string, number, boolean, array, and object).

1.5 Unused / unexplored features

Our focus is data wrangling, so we will only explore some aspects of MongoDB's data model and querying features. Many other features will not be explored; they typically fall into the realm of data administration or architecture.

Features **not** explored:

- Replication and sharding: high availability and scalability
- Indexing: query performance (non-sharded indexes are conceptually similar to RDBMS indexes)
- Schemas: data validation
- Geospatial query capabilities
- Text search capabilities: text indexing (different to Regular Expressions)
- Security, account management, performance management, and other administrative matters

2. Reference material

- Official reference: https://docs.mongodb.com/manual
- Quick reference cards: https://www.mongodb.com/collateral/quick-reference-cards
- Cheat sheet: https://www.mongodb.com/developer/quickstart/cheat-sheet
- SQL to MongoDB mapping chart: https://docs.mongodb.com/manual/reference/sql-comparison
- Glossary of terms: https://docs.mongodb.com/manual/reference/glossary
- MongoDB University: https://university.mongodb.com
- Stack Overflow: https://stackoverflow.com/questions/tagged/mongodb

3. Documents

Read MongoDB Documents and Thinking Documents Part 1.

A document is represented as a JSON object. We will use the terms document and object interchangeably.

Copy and paste the JSON document below into Robo3T. View it in tree mode, table mode, and text mode.

```
exampleDocument = {
    _id: ObjectId("59a3537f1e82d5bf85d54c15"),
    displayName: "Raymond Brothers",
    name: {
       first: "Raymond",
       last: "Brothers",
       other: [
          "RayRay",
          "BroBro" ] },
    birth: "1980-06-29",
    lastLogin: new Date("2017/08/01"),
    favStatuses: [
        "asleep",
        "awake",
        "gone" ],
    totalViews : 128053,
    recentActivity: [
        {
            comment: "yo",
            postId: 2001 },
        {
            activity: "logoff",
            codes: [
                0,
                2,
                4 ] },
        {
            postId: 2080,
            passive: true } ],
    level: 24
}
```

3.1 Data types

String, number, boolean, array, and object are no different to typical JSON use.

Noteworthy are:

- Number is stored by default as a 64-bit Double in mongo, and 32-bit Integer in mongosh
  - mongo: for decimal use Double; for huge decimal use NumberDecimal; for integer use NumberInt; for huge integer use NumberLong
  - mongosh: for decimal use Double; for huge decimal use Decimal128; for integer use Int32; for huge integer use Long
- Date is a datatype. Use new Date() to specify dates. Note that sometimes datetime data in a dataset could be stored as date, string or number.

Please read the reference document on data types for mongo or mongosh.

3.1.1 Exercises

In the same tab as where you pasted the `exampleDocument`, type the following code below the document and execute the query.

```
typeof exampleDocument.displayName
```

The query's result is *string*. This means the data type of the displayName field is string.

Task: for each field of the `exampleDocument` document, use the typeof keyword to discover its type.

3.2 _id and ObjectId type

Notice that documents must have an _id field. This is called the "underscore ID" field and acts as a primary key. If an _id is not provided, it will be automatically added with a unique ObjectId typed value provided by "`new ObjectId()`". The _id field may be any datatype and any value so long as it is unique within a collection.

3.2.1 Exercises

a) Use Robo3T to create a new database named **exercises**.
b) Open a new query tab for the exercises database.
c) Paste the commands below one by one and execute them. (At this stage, you do not need to know what the commands mean; we will cover them in the sections below.)

```
//"db" refers to the active database

//See how many documents are in the DatatypeExercises collection; note even
though this collection does not exist, no error is given.
db.DatatypeExercises.count()

// Insert documents one by one; the collection is automatically created if it
does not already exist
db.DatatypeExercises.insertOne({ _id: 1, doc: 1 })

db.DatatypeExercises.insertOne({ _id: 2, doc: 2 })

db.DatatypeExercises.insertOne({ _id: "2", doc: 3 })

db.DatatypeExercises.insertOne({ _id: new ObjectId(), doc: 4 })
```

```
// There is no fixed schema so documents can have variations in fields, and
within the same field variations of datatypes
db.DatatypeExercises.insertOne({ doc: "five", freedom: true, flexibility: ["so",
"much", "freedom", { oh: "yes"} ] })

// See how many documents are in the collection now
db.DatatypeExercises.count()

// Find the _id of the last inserted document
db.DatatypeExercises.findOne({ doc: "five" })._id
```

d) Insert a new document with the same _id found above. Use the template code below.

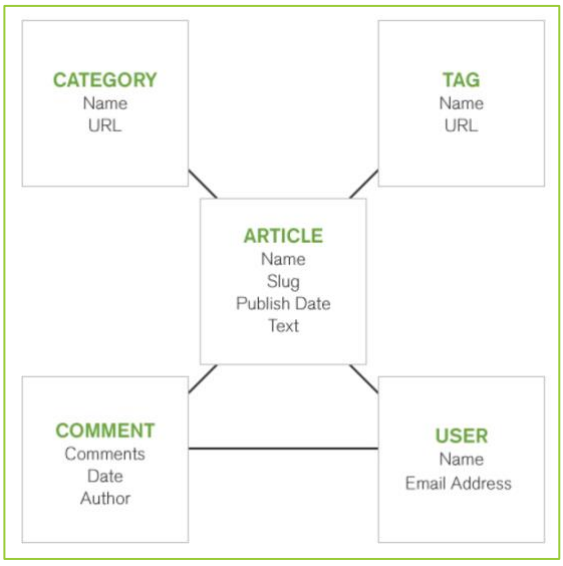
```
db.DatatypeExercises.insertOne({ _id: ObjectId("replace yours here"), doc: 6 })
```

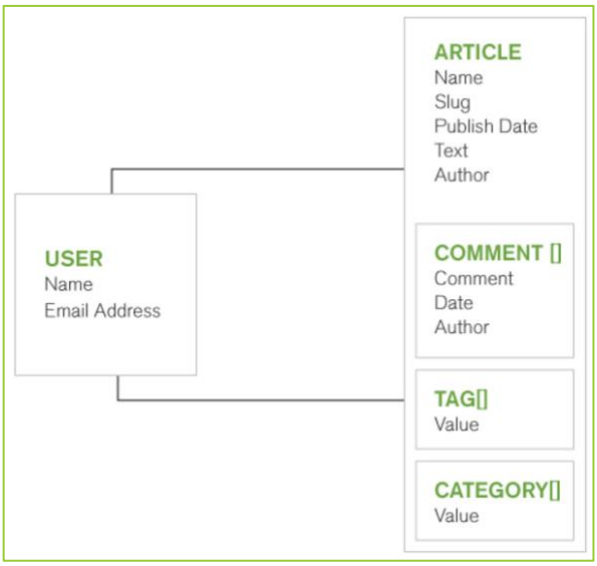
e) Insert a new document with the same _id as document with doc = 2.
f) Insert a new document with the same _id as document with doc = 3.

3.3 Listing and nesting

A document can contain multiple values for any of its fields. The tabular data rule of "atomic values for each field" does not apply. E.g. a single document representing an article can contain multiple comments and tags. This is achieved using arrays and/or nested[2] documents.

In the images[3] below, the left-hand image shows a relational model for articles. To show an article and its comments, a five-way join is needed(!). The right-hand image shows a document model where comments, tags, and categories are nested within each article, and users are linked to article.author and comment.author. The link here does not mean a join; it means two operations are needed to retrieve an article and its author: one for the article and another for the user (by using the userid from the article's author).





If we knew that:

- when displaying articles, we only show the author's name, and
- when displaying article comments we only show the commenter's name,

[2] Nested documents are also called "embedded documents".
[3] Source: https://www.mongodb.com/blog/post/thinking-documents-part-1

what could be done to reduce the operations of retrieving articles—and all required information, including authors and commenters—to only one operation?

We will explore arrays and nested documents in sections below.

3.4 Field and value existence (nulls, blanks, and undefined)

If a document contains a field, we say the field "exists" in the document, otherwise the field "does not exist" (also known as the field being "undefined").

If a field exists and contains no value, it is shown as `null`. Every other value is considered a value, including empty string, zero, empty array, and empty object.

3.4.1 Exercises

```
{
    _id: null,
    name: "name",
    title: [],
    "full name": {},
    titel: 0,
    a: [0],
    b: [null]
}
```

Consider the document above. For each field in the table below, state whether it exists in the document. If it exists state whether it has a value. If it has a value state what that value is.

Field name	Exists?	Has value?	Value
id			
_id			
Name			
name			
TITLE			
a			
b			
c			
A			
B			

4. Import, export, dump, and restore

MongoDB comes with four tools to move data in and out of a database using portable formats. They are all command line tools, so use them in cmd or Terminal.

4.1 mongoimport

To insert or update data from a JSON, CSV, or TSV file, use mongoimport. See https://docs.mongodb.com/manual/reference/program/mongoimport/

Parameters to note are:

Option	What it does
`db`	
`collection`	
`type`	
`mode`	
`jsonArray`	

A typical mongoimport to localhost:27017 will look like:

```
mongoimport -d myDatabaseName -c myCollectionName aFileFromMongoExport.json
```

4.1.1 Exercises

a) Download and expand the file twitter.json.zip, and import the JSON data into a collection named **tweet** in a database named **Twitter**.

b) The query below should show there are 51428 documents in the tweet collection.

```
use Twitter
db.tweet.count()
```

c) Import the data from the file restaurant.json.zip:
- There should be 25359 documents imported.
- Drop the collection; this dataset is not used later.

d) Import the Instagram data from the file ig-json-files.zip:
- Use the database name **ig**.
- For all files suffixed with "posts", import them into the **posts** collection.
- For all files suffixed with "profile", import them into the **profiles** collection.
- Verify there are 15838 documents in the posts collection, and 39 in the profiles collection.
- An easy way to generate the required 78 commands is to use Excel formulas (check ig-json-importer.xlsx)

4.2 mongoexport

To export data from a collection to a JSON or CSV file, use mongoexport. See https://docs.mongodb.com/manual/reference/program/mongoexport/

4.2.1 Exercises

a) Export the data from your DatatypeExercises collection created in section 3.2.1 above.

4.3 mongorestore

To restore a BSON dump of a MongoDB database, use mongorestore. See https://docs.mongodb.com/manual/reference/program/mongorestore/.

The dryRun and verbose options should be used to avoid accidentally overwriting data.

A typical mongorestore to localhost:27017 using the database and collection names from the BSON dump directory (named "dump") will look like:

```
mongorestore dump/
```

4.3.1 Exercises

a) The file ig-mongo-dump.zip contains the ig dataset [from 4.1.1.d) above] as a mongodump. Restore it using the drop option. This will provide you with a full restore of the ig dataset.
b) Verify the restored **ig** database has 15838 documents in the **posts** collection, and 39 in the **profiles** collection.
c) mongorestore the Twitter dataset from the file Twitter-dump.zip:
    - Either drop the Twitter database first, or use the drop option.
d) Verify the restored **Twitter** database contains a collection named **tweet** with 51428 documents.

4.4 mongodump

To create a BSON dump of a MongoDB database or collection, use mongodump. See https://docs.mongodb.com/manual/reference/program/mongodump/.

4.4.1 Exercises

a) Take a mongodump of the data from your DatatypeExercises collection created in section 3.2.1 above.

5. Find and MQL

MongoDB's find method is the mechanism by which documents are retrieved. Find is invoked on a single collection and you can specify a **query filter** parameter and a **projection** parameter. Parameters are formatted as JSON documents.

Find is like an SQL SELECT; query filters like WHERE clauses; projections like fields to select.

`SELECT * FROM coll` would look like:

```
db.coll.find({}, {}) //or
db.coll.find()
```

`db` represents the currently active database; `coll` is the name of a collection.

Where no filter document (or an empty document) is provided, all documents from the collection are returned.

Note that the find method returns a *cursor*, not any data itself. Robo3T automatically consumes the cursor to provide the data in batches of 50. It achieves this by performing cursor operations (next, limit, skip) behind the scenes.

5.1 Projection

Read:

- Project Fields to Return from Query
  - Dot Notation
- Projection Parameter

Where no projection document (or an empty document) is provided, all fields are returned for each document that matches the filter document. Note for the examples and exercises below we will use an empty filter document, which will result in all documents being returned.

5.1.1 Examples

Retrieve all documents from the ig.profiles collection, and show:

a) All fields
b) All fields except _id `{ _id: 0 }`
c) Only the _id field
d) All fields except media
e) Only the _id, username, and full_name fields `{ username: 1, full_name: 1 }`
f) Only the username and full_name fields
g) Only the media field
h) Only the count field of media
i) Only the nodes field of media `{ "media.nodes": 1 }`
j) Only the count fields of media, followed_by, and follows

5.1.2 Exercises

Retrieve all documents from the ig.posts collection, and show:

k) All fields
l) All fields except _id
m) Only the _id field
n) All fields except the comments array
o) Only the _id, date, and id fields

p) Only the date and id fields
q) Only the comments field
r) Only the id field of each comment (use dot notation)
s) Only the width dimension
t) Only the height dimension
u) Only the usernames of each usertag

5.2 Skip, count, limit methods

Skipping and limiting can be done behind the scenes by Robo3T by using the "arrows" on the results area. They can also be included in a query for the server to process before returning a cursor.

Add `.skip(n)` to the end of a `find()` method to skip the first n matching documents.

Add `.limit(n)` to the end of a `find()` method to only return the first n matching documents.

See: https://docs.mongodb.com/manual/reference/method/db.collection.find/#modify-the-cursor-behavior

skip, limit (and sort) can be chained:
https://docs.mongodb.com/manual/reference/method/db.collection.find/#combine-cursor-methods

Add `.count()` to the end of a `find()` method to return the number of matching documents:
https://docs.mongodb.com/manual/reference/method/cursor.count/#cursor.count

5.2.1 Examples

a) Use a find query to show how many documents are in the ig.posts collection
b) Use a find query to show how many documents are in the ig.profiles collection

Using the default document order in the ig.profiles collection, show:

c) The first 5 documents only `db.profiles.find().limit(5)`
d) The first 10 documents only
e) The 38th document onwards `db.profiles.find().skip(37)`
f) The 38th document only
g) The 10th to 20th documents only `db.profiles.find().skip(?).limit(?)`

5.2.2 Exercises

a) Use a find query to show how many documents are in the Twitter.tweet collection
b) Use a find query to show how many documents are in the crunchbase.company collection

Using the default document order in the Twitter.tweet collection, show:

c) The 50th document only
d) The 500th document only
e) The 5000th document only
f) The _id of the 5000th document
g) Only the user id, user name, created datetime, and tweet text of the first tweet
h) Only the user id, user name, created datetime, and tweet text of the 5000th tweet

5.3 Sorting

References:

- https://docs.mongodb.com/manual/reference/method/db.collection.find/#order-documents-in-the-result-set

- https://docs.mongodb.com/manual/reference/method/cursor.sort/#cursor.sort

MongoDB query results do not guarantee any order unless the `sort()` method is used. Add `.sort()` to the end of a `find()` method to sort the results returned.

Sort requires a single parameter: a sort document which lists which fields to sort by, in which order, and whether ascendingly or descendingly. E.g.

```
{ idNumber: 1, lastName: 1, registration: -1 }
```

means to sort by `idNumber` ascendingly first, then for documents with the same `idNumber`, sort them by `lastName` ascendingly, then for documents with the same `idNumber` and `lastName` , sort them descendingly by `registration`. Since the same fields can contain different data types, MongoDB follows the ordering shown here: https://docs.mongodb.com/manual/reference/bson-type-comparison-order/#bson-types-comparison-order

5.3.1 Examples

Retrieve all documents from the ig.profiles collection, and show:

a) Documents alphabetically sorted by full_name
b) Documents reverse-alphabetically sorted by full_name
c) Documents ascendingly sorted by followed_by count
d) Documents descendingly sorted by follows count
e) Verified profiles listed first

Retrieve all documents from the ig.posts collection, show documents without the comments and usertags fields, and sort by:

f) _id alphabetically
g) owner id ascendingly
h) owner id (asc) then code (asc)
i) date (desc) then code (desc)

Retrieve a single document which is:

j) The post with the largest like count `.sort({ "likes.count": -1 }).limit(1)`
k) The profile with the most followers
l) The profile with the fewest followers
m) The profile who follows the most users
n) The profile who follows the fewest users

5.3.2 Exercises

Use the Twitter db to write queries to retrieve:

a) The 390th tweet when sorted by the id field
    - id is not the same as _id
b) The 391st tweet when sorted by user screen_name (asc) then id (desc)
c) A tweet by the user with the most friends
d) A tweet by the user with the most tweets (according to their statuses_count)
e) A tweet by the user with the most followers
f) Check that for c), d), and e), there is only one user with the largest count

5.4 Filter: exact matching

So far, our queries have retrieved all documents from the collection because no filter was specified. We will now explore simple filters starting with the equality condition.

Query filters are specified as a document. Where values are provided for a field, it translates to an equality condition (i.e. exact match)[4]. E.g.

```
db.coll.find({ _id: 100 }) //or
db.coll.find({ _id: 100 }, {})
```

means to only return documents that match the condition where the _id field has the numeric value 100. The logic in SQL is `SELECT * FROM coll WHERE _id = 100`.

Where multiple conditions (on different fields) are given, they are combined using AND conjunctions, i.e. a document must match all the given conditions.

5.4.1 Examples

Find profiles in ig which match the conditions:

a) username is airnz
b) username is AirNz
c) _id is 218149346
d) full_name is qantas
e) media count is 630
f) followed_by count is 190508
g) followed_by count is 390
h) Have a country_block field with a value, but its value is empty.

Find posts in ig which match the conditions:

i) Owned by user with id = 21868725
j) Owned by user with id = 21868725 and is a video
    - How many posts match?
k) Owned by user with id = 21868725 and is not a video
    - How many posts match?
l) Tagged the location, "Auckland, New Zealand"
m) Tagged a location with a public page
n) Have a usertags field, but no users are tagged
o) Tagged the username: airnz
    - Note the difference between matching arrays exactly, and matching at least one element in the array: https://docs.mongodb.com/manual/tutorial/query-arrays/
p) Have a single comment with the comment's id equal to 17846969389114957. Note:
    - comment ids are strings,
    - the first comment is at index 0 of the comments array. If there is only 1 comment, then index 0 will always exist. (Use dot notation.)
q) The user with username airnz has made a comment on these posts

5.4.2 Exercises

a) Find a tweet made by the user with the most tweets. Note down the user's id. Use it to find all tweets made by the user

Find tweets (also called statuses) in the Twitter db which match the conditions:

b) Authored by user with screen_name = AucklandNZ

[4] This "style" of specifying values to match is a simplified way of specifying an equality condition. The lengthier way is to use the $eq and $and operators: https://docs.mongodb.com/manual/reference/operator/query/eq; https://docs.mongodb.com/manual/reference/operator/query/and

c) Have no user mentions
d) Have no user mentions and no hashtags
e) Have no user mentions and no hashtags and no url entities
f) Contain the hashtag, "mongodb" (without quotes), according to the entities field
g) Contain the hashtag, "love" (without quotes), according to the entities field
h) Authored by a user with no followers
i) Authored by a user with no followers, and is a retweet of the status id = 22558138123.
    - The original tweet which is retweeted is found in the retweeted_status field. E.g. look at tweet with _id = ObjectId("59a3808cb62fc471523217a5")
j) Created at this date/time: Thursday September 02 19:36:23 +0000 2010

Combining projection, skip, limit, count, and filters:

k) For all queries in 5.4.1, show how many documents match the condition instead of returning the documents
l) For all queries in 5.4.1, return the documents without the _id field, and sorted by id ascendingly
m) As in e), but only return the 1st document
n) As in e), but only return the 39th document

5.5 Filter: comparison operators

To specify non-equality conditions, we use comparison operators. As seen above, query filters are specified as a document. Instead of providing single values (which would result in an equality condition), we now provide a sub-document specifying the details of an operator. E.g

```
db.coll.find({ _id: { $gt: 100 } }) //or
db.coll.find({ _id: { $gt: 100 } }, {})
```

means to only return documents that match the condition where the _id field has a numeric value greater than 100. The logic in SQL is `SELECT * FROM coll WHERE _id > 100`.

Where multiple conditions are given in the sub-document (i.e. **on the same field**), they are combined using AND conjunctions such that the field must match all the given conditions. E.g.

```
db.coll.find({ _id: { $gt: 100, $lt: 200 } })
```

means the condition, "_id greater and 100 and _id less than 200". The logic in SQL is

```
SELECT * FROM coll WHERE 100 < _id AND _id < 200
```

Operator	Function (SQL equivalent)
`$gt`	
`$gte`	
`$lt`	
`$lte`	
`$ne`	
`$in`	IN (*item1, item2, …*)
`$nin`	NOT IN (*item1, item2, …*)

Comparison operator references: https://docs.mongodb.com/manual/reference/operator/query-comparison/

Note for `$in` and `$nin` the value provided is an array. E.g.

```
db.coll.find({ _id: { $in: [100, 200, 300] } })
```

```
db.coll.find({ _id: { $nin: [100, 200, 300] } })
```

Remember that projection, sorting, skipping, and limiting are still applicable; they "stack" on top of filtering.

5.5.1 Examples

Find posts in ig which match the conditions below. Only project fields which are used in the filter.

a) Have more than 1 like
b) Have at least 1 like
c) Have at least 200 likes
d) Have at most 200 likes
e) Have between 500 and 1000 likes (both inclusive)
    - How many posts match?
f) Have between 500 and 1000 likes (both exclusive)
    - How many posts match? Comparing with your answer to e), what can you deduce about how many posts have either exactly 500 or 1000 likes?

Only project fields which are used in the filter. How many posts in ig contain an image that:

g) Is at least 720 pixels wide?
h) Is more than 720 pixels wide?
i) Is wider than 1080 pixels?
j) Is exactly 720 pixels in height and at least 720 pixels wide?
k) Is exactly 1080 by 1080 with at least 1000 likes on the post?

Using ig:

l) Project only the username field for profiles with the full names: "Air New Zealand", "Virgin Australia", or "Qantas"
m) Find how many profiles exist apart from those with the usernames found in question l)
n) Apart from the user with the fewest posts and the one with the most posts (i.e. exclude these 2), list the post counts of all other users in ascending order
    - Use a query to find the user with the fewest posts; note down its _id
    - Use another query to find the user with the most posts; note down its _id
    - Using the _id values found, write another query to list the desired counts

5.5.2 Exercises

Using the Twitter db:

a) Find all tweets made by the users with these screen names: `luucastro_`, `The_CopyEditor`, `DANIGGAUNEED_IB`, `NZucker`
b) How many tweets do not have a null retweet_count?
c) How many retweets (which are contained in the retweeted_status field) do not have a null retweet_count?
    - What does this mean about the retweet_count field for the database?
d) Are there any retweets which contain retweets?
e) Find tweets which have a "place" tag (i.e. is not null)
f) Find tweets which have a "place" tag and a "geo" tag (i.e. both fields are not null)
g) Find tweets which have no url, no hashtag, and no mention entities, and that were made by a user with more than 100 followers
h) As above, and add the condition that the user also has made between 3000 and 4000 tweets (inclusive)

5.6 Filter: field comparisons

Comparisons operators in 5.5 compared a field value to a specified value. To compare field values against another field value within the same document, we can use the `$expr` operator. E.g.

```
db.coll.find( { $expr: { $gt: [ "$spent" , "$budget" ] } } )
```

means to only return documents that match the condition where the document's `spent` field value is greater than its `budget` field value. The logic in SQL is `SELECT * FROM coll WHERE spent > budget`.

Notes when using `$expr`:

- field names are "field path variables". The syntax is such that they must be provided as strings prefixed with the dollar sign. This is explained here under "Field Path and System Variables".
- Available operators in `$expr` are aggregation pipeline operators: https://docs.mongodb.com/manual/reference/operator/aggregation/#comparison-expression-operators This is why `$gt` when used with `$expr` has different syntax to what we saw in 5.5.

`$expr` reference: https://docs.mongodb.com/manual/reference/operator/query/expr/

Remember that projection, sorting, skipping, and limiting are still applicable; they "stack" on top of filtering.

5.6.1 Examples

Find profiles in ig which match the conditions below.

a) Are followed by more users than they follow
b) Follow more users than they are followed by
c) Have the same URLs for their standard profile picture and HD profile picture
d) Have different URLs for their standard profile picture and HD profile picture

5.6.2 Exercises

Use the Twitter db:

a) Find all tweets in which a user replied to themselves.

Use the ig db: find how many posts in ig contain an image that:

b) are square
c) are not square?

5.7 Filter: logical operators

When specifying multiple conditions, we need to use logical operators to say how those conditions should be combined. We have seen earlier that if no logical operator is specified, then conditions are joined using AND.

As with all other filters, logical operators are provided as documents. The values for logical operators must be either be a single condition, or a list of conditions (in an array).

Operator	When to use

`$or`	At least one condition is true of the documents to return.
	Conditions are on separate fields. Where conditions are on the same field, use `$in`.
	
	Examples using the ig db:
	`db.profiles.find({`
	`  $or: [`
	`  { "followed_by.count": 0 },`
	`  { "follows.count": 0 }`
	`  ]`
	`})`
	
	`db.profiles.find({`
	`  $or: [`
	`  { _id: { $in: ["70", "21868725", "218149346"] } },`
	`  { username: { $in: ["qantas", "airnz", "scobleizer"] } }`
	`  ]`
	`})`

`$not`	Negate a condition. Negate a Regular Expression match. `$not` will also match documents which do not contain the field. This is the same as `$ne`, but not the same as `$lt`, `$lte`, `$gt`, `$gte`. See the example in the documentation.

`$and`	Since
	`db.coll.find({`
	`  $and: [`
	`  { _id: { $gt: 100 } },`
	`  { _id: { $lt: 200 } }`
	`  ]`
	`})`
	is the same as
	`db.coll.find({ _id: { $gt: 100, $lt: 200 } })`
	
	`$and` should only be used in "complex" cases, e.g.
	`db.coll.find({`
	`  $and: [`
	`  { $or: [ { mon: "Jan" }, { discount: true } ] },`
	`  { $or: [ { size: "L" }, { weight: { $gte: 100 } } ] }`
	`  ]`
	`})`

Logical operator references: https://docs.mongodb.com/manual/reference/operator/query-logical/. (We will not cover the `$nor` operator.)

Remember that projection, sorting, skipping, and limiting are still applicable; they "stack" on top of filtering.

5.7.1 Examples

Use `$or` to find ig profiles:

a) With fewer than 100 followers or follow no one
b) With no followers or follow fewer than 100 users
c) With more than 5000 media posts or fewer than 100 followers
d) With no posts or no followers
e) Which satisfy at least one of these three conditions: have no posts; have no followers; follow no one
f) Which either:
    - have an external url (non-blank) and are verified, or
    - do not have an external url (blank) and are not verified

Use the ig posts collection.

g) Find how many posts have at least 5000 likes. Use `$gte`.
    - Note this number as the "g-count"
h) Find how many posts have 4999 or fewer likes. Use `$lt`.
    - Note this number as the "h-count"
i) Find how many posts do not have at least 5000 likes. Use `$not` to negate your condition in g).
    - Note this number as the "i-count"
j) Given there are 15838 posts in total, what can you conclude about the `likes.count` field in the collection? Note that:
    - g-count + h-count is less than 15838.
    - g-count + i-count = 15938
k) The query below returns all captions which contain a hashtag containing "NZ" or "NewZealand" (without quotes).

```
db.posts.find(
  { caption: /#[\w]*(NewZealand|NZ)/},
  { caption: 1, _id: 0 }
)
```

Use the `$not` operator to find all posts that do not contain a hashtag containing "NZ" or "NewZealand".

5.7.2 Exercises

Using the Twitter db, write queries using `$or` or `$in` to find:

a) Tweets which have a place and/or a geo tag (i.e. at least one field is not null)
b) Tweets which are either in reply to a tweet or in reply to a user
c) Tweets which were made by a user with no location detail, i.e. location is either null or an empty string
d) Tweets which are either:
    - retweets (i.e. contain a retweeted_status document) and are truncated, or
    - not retweets and are not truncated
e) The query below returns all tweets created in 2010. Modify it to find tweets that were created in other years (negate the condition given)

```
db.tweet.find({ created_at: / 2010$/ })
```

f) The query below returns all tweets created on Thursdays. Modify it to find tweets that were created on other days of the week (negate the condition given)

```
db.tweet.find({ created_at: /^thu /i })
```

g) What can you thus conclude about the created datetimes of tweets in the Twitter db?

5.8 Filter: arrays

Recall earlier that when querying fields with array, we can either match the array exactly, or match at least one element in the array. (See https://docs.mongodb.com/manual/tutorial/query-arrays/.)

To experiment with the below example queries, set up the ArrayExercises collection by running the commands in the file, ArrayExercises.js (https://drive.google.com/file/d/1Daxmkwz4ERbYrjRC8sevz_w4aEeFH2dO/view?usp=sharing).

Where an array value is specified…

The condition is to find an **array** that looks **exactly** like the array provided, e.g.

```
db.ArrayExercises.find({ scores: [30, 20, 10.0] })
```

means to find all documents whose score field is an array of exactly 3 elements, the first of which is the number 30.0, the second is 20.0, and the third element is 10.0 (in that exact order).

Where the value is NOT an array…

The condition is to find an **array** that **contains at least one element** that matches the value given, e.g.

```
db.ArrayExercises.find({ scores: 20.0 })
```

means to find all documents whose score field (if it is an array) contains at least one element that is the number 20.0 (or if the field is not an array, its value is 20.0).

Where multiple conditions are specified…

For example,

```
db.ArrayExercises.find({ scores: { $gt: 30, $lt: 20 } })
```

means to find an array where at least one element can satisfy each condition (not necessarily the same element). In the example above, it means to find all documents whose score field (if it is an array) contains at least one element that is greater than 30, and at least one element that is less than 20 (or if the field is not an array, its value is greater than 30 and less than 20).

Dot notation

Using dot notation allows us to query for values in a specified location of an array, e.g.

```
db.ArrayExercises.find(
{ $or: [
    { "scores.0": 10 },              //array indexes begin at 0
    { "authors.1.name": "Betsy" }
  ]
})
```

means to find an array either:

- named `scores`, in which the **first** element (at index 0) is 10, or
- named `authors`, in which the **second** element (at index 1) is a document where its name field is "Betsy"

Remember that documents do not have to have the same datatype for fields with the same name.

There are 3 array operators which provide additional querying options.

Operator	What it matches
`$all`	An array which contains all the values provided. E.g. `db.ArrayExercises.find({ scores: { $all: [30, 20, 10] } })` means to match an array which contains 30 and 20 and 10 in any order (and it can contain other elements too).

`$elemMatch`	An array which contains at least one element that matches all the conditions provided.
	
	Where scores is an array of numbers, consider the difference between:
	`db.ArrayExercises.find({`
	`  scores: { $gt: 30, $lt: 20 }`
	`})`
	`//vs`
	`db.ArrayExercises.find({`
	`  scores: { $elemMatch: { $gt: 30, $lt: 20 } }`
	`})`
	
	Where authors is an array of documents, consider the difference between:
	`db.ArrayExercises.find({`
	`  "authors.gender": "f",`
	`  "authors.current": true`
	`}) //vs`
	`db. ArrayExercises.find({`
	`  authors: { $elemMatch: { gender: "f", current: true } }`
	`})`

`$size`	An array with a specified number of elements. E.g. `db.ArrayExercises.find({ scores: { $size: 3 } })` means to match an array which has exactly 3 elements.

Array operator references: https://docs.mongodb.com/manual/reference/operator/query-array/

Remember that projection, sorting, skipping, and limiting are still applicable; they "stack" on top of filtering.

5.8.1 Exercises

No exercises require `$elemMatch`.

Use the Twitter db to find tweets:

a) Which contain the hashtags #music and #nowplaying
b) Which contain the hashtags #job and #jobs and #hiring
c) Which contain the hashtags #job or #jobs
d) Which contain the hashtags #job or #jobs or #hiring
e) With exactly 1 url tag, and any number of user mentions and hashtags
f) With exactly 5 hashtags, and any number of user mentions and url tags
g) With exactly 1 url tag, 1 user mention, and any number of hashtags
h) With exactly 1 url tag, and 1 user mention, and 1 hashtag
i) Which have 1 url tag and/or 1 user mention and/or 1 hashtag
j) Where the first hashtag is #news (there may be other hashtags)
k) Where the first hashtag is #news and the second hashtag is #russia (there may be other hashtags)

Use the ig db to find posts:

l) Which contain no comments
m) Which contain exactly 10 comments
n) Which contain a comment written by the user whose username is airnz
o) For which the first comment is written by airnz
p) For which the first comment is written by airnz or qantas
q) For which the tenth comment is written by airnz
r) Which contain at least one comment written by airnz and one by unknown_aircraft

5.9 Filter: null and $exists

Auditing your data will reveal what fields may be undefined, null, or an empty array, empty string etc.

To match null fields, treat null as a value, e.g.

```
db.coll.find({ aField: null }) //also matches if the field does not exist
```

To match non-null, use `$ne`: null, e.g.

```
db.coll.find({ aField: { $ne: null } })
```

To determine if a field exists or not, use `$exists`, e.g.

```
db.coll.find({ aField: { $exists: true } })    //true or 1
db.coll.find({ aField: { $exists: false } })   //false or 0
```

Since some operators match if the field does not satisfy the condition OR if the field does not exist, sometimes it is necessary to explicitly check that the field exists. The condition below finds documents where the field exists and is not equal to 10. Without checking for existence, all documents that do not have the field would also match `$ne`: 10.

```
{ field: { $exists: 1, $ne: 10 } }
```

See https://docs.mongodb.com/manual/tutorial/query-for-null-fields/ and https://docs.mongodb.com/manual/reference/operator/query/exists.

5.9.1 Examples

Use the ig db. Determine if the fields i) always exist, ii) are ever null

a) profile.country_block
   - What value represents "no country block"?
b) profile.media.nodes
   - What value represents "no media nodes"?
c) posts.comments
   - What value(s) represent(s) "no comments"?

In ig.posts:

d) Are there any posts which has usertags but not usertags.nodes?
e) Are there any posts which have usertags.nodes but no users tagged?
f) Are there any posts with a comment which does not have all 4 fields: `created_at`, `id`, `text`, `user`?
   - Use `$nin` to check that a post has comments
   - Use `$or` to check if any of the 4 fields do not exist

Using a trick that combines `$exists` and array index querying (using dot notation) we can query if an array has "at least n" elements, i.e. size is at least n. See https://stackoverflow.com/questions/7811163/query-for-

documents-where-array-size-is-greater-than-1#15224544. Remember that **array indexes are numbered starting from 0**.

g) How many posts have 10 or more comments?
h) How many posts have more than 100 comments?

5.9.2 Exercises

Use the Twitter db.

a) How many tweets do not contain a retweeted status?
b) How many tweets contain a url tag with an expanded url?
c) Are any of these fields ever null? `source`, `truncated`, `user`, `in_reply_to_user_id`, `user.location`
d) How many tweets have at least 10 user mentions?
e) How many tweets have at least 5 urls?
f) How many tweets have at least 10 hashtags?
g) How many tweets have more than 10 user mentions and hashtags?
h) How many tweets have more than 10 user mentions and/or hashtags?
i) How many tweets were made by a user with no location?
   - First consider: does user.location always exist? Can user.location can be null? Can user.location be blank?
   - Then write a query which captures all cases where no user location is provided
j) How many retweeted statuses were made by a user with no location?
k) How many tweets were made by a user with no url?
   - Note that "http://null" is not considered a valid url

5.10 Filter: $type

Unless a collection implements schema validation, any of its document can have any field, and any of those fields can have values of any data type. So a helpful way to audit or profile data is to examine the data types of values of a field. We can do this manually by querying using the `$type` operator. (MongoDB Compass is a tool to profile data less manually.)

"Querying by data type is useful when dealing with highly unstructured data where data types are not predictable." https://docs.mongodb.com/manual/reference/operator/query/type/

5.10.1 Examples/Exercises

Use the Twitter db.

a) Are all values for `id` 64-bit integers?
   - Use $not to negate a filter condition.
b) Are user.id and id values different in type?
   - Are they both numeric, though? What "type" can be used to confirm this?

Note there is an aggregation operator also called `$type` which *returns* the data type for a document.

```
db.tweet.aggregate(
[
    {
        $project: {
           type_id: { $type: "$id" },
           type_user_id: { $type: "$user.id" }
        }
    },
```

```
    {
        $group: {
            _id: { type_id: "$type_id",
                   type_user_id: "$type_user_id" },
            num: { $sum: 1 }
        }
    }
])
```

5.11 Dates

Note that:

- `new Date('2017-01-01')` is the SAME as
  `new Date('2017-01-01T00:00:00Z')`
  - i.e. dates without times default to UTC midnight

- `new Date('2017-01-01T05:00:00')` is NOT the same as
  `new Date('2017-01-01T05:00:00Z')`
  - The first is in local time (+1200 or +1300 in New Zealand) which gets converted to UTC (+0000) and the second is specified in UTC (that is what the "`Z`" means). `Z` is the same as `+0000`, i.e. `T12:34:56Z` is the same as `T12:34:56+0000`.

See https://docs.mongodb.com/manual/reference/method/Date/

For this course, assume all dates are in UTC. Where you must specify a time, ensure you include the "Z" at the end of the timestamp.

6. Insert, update, and delete

To create, update, and delete data, use the appropriate method on a collection.

See:

- https://docs.mongodb.com/manual/tutorial/insert-documents/
- https://docs.mongodb.com/manual/tutorial/update-documents/
- https://docs.mongodb.com/manual/tutorial/remove-documents/

There are variations of write queries, e.g. `findOneAndReplace()`, `findAndModify()` which are not covered here.

Note that this section is not examinable this semester.

6.1 insert

Where `{}` are documents to add to a collection named coll:

```
db.coll.insertOne( {} ) //inserts one document
db.coll.insertMany( [ {}, {}, … ] ) //inserts multiple documents
```

6.1.1 Exercises

Use exercises.WriteExercises:

a) Insert an empty document
b) Insert a document with only the _id field specified. Use any value
c) Insert another document with only the _id field specified. Use any value
d) Insert 3 empty documents with a single query
e) Run `db.WriteExercises.insert( { e: true } )`
f) Run `db.WriteExercises.insert(`

```
[
  { f: true },
  { f: true },
  { f: true }
])
```

6.2 update

```
db.coll.updateOne( filterDoc, updateMethodsDoc )
db.coll.updateMany( filterDoc, updateMethodsDoc )
```

Where `filterDoc` is a document containing a filter (same as `find()`) so that only matching documents will be updated, and `updateMethodsDoc` is a document containing update operators. See https://docs.mongodb.com/manual/reference/operator/update-field/
The only difference between `updateOne` and `updateMany` is that the former will only update the first matching document, whereas the latter will update all matching documents.

Example:

```
db.WriteExercises.insert({
    _id: "updateExample",
    aNum: 10,
    anoNum: 10,
    aNull: null,
    anArray: [1,2,3],
    yo: "sup"
```

```
})

db.WriteExercises.updateMany(
    { _id: "updateExample" },          //only update this doc
    {
        $inc: { aNum: 5, anoNum: -5 }, //plus minus numbers
        $set: { anArray: null },       //set to null
        $rename: { yo: "sup" },        //rename field
        $unset: { aNull: "" }          //remove field
    }
)
```

Note that in MongoDB, an "upsert" means to update the document if it can be found, otherwise insert it.

6.3 delete

Delete means to remove documents from a collection, whereas "drop" means to remove the collection itself. As in SQL where we delete rows and drop tables, MongoDB uses the same terminology, so we also delete documents and drop collections.

```
db.coll.deleteOne( filterDoc )
db.coll.deleteMany( filterDoc )
```

Where `filterDoc` is a document containing a filter (same as `find()`); only matching documents will be deleted.

The only difference between `deleteOne` and `deleteMany` is that the former will only delete the first matching document, whereas the latter will delete all matching documents.

6.3.1 Exercises

Use exercises.WriteExercises:

a) Find the _id of the document added in 6.1.1.a). Use the _id value to delete the document
b) Delete the documents added in 6.1.1.f) without specifying their _ids
c) Delete the document added in 6.1.1.e) without specifying its _id
d) Delete all remaining documents in a single query

7. Aggregation pipeline

MongoDB provides aggregation operations in three ways. See https://docs.mongodb.com/manual/aggregation/

(1) Collection aggregation methods

We have already used the `collection.count()` method, which aggregates documents in a collection and returns the number of documents.

Another method is `distinct()`. Below is an example on the ig db to show all users that have been tagged:

```
db.posts.distinct("usertags.nodes.user.username")
```

See https://docs.mongodb.com/manual/reference/method/db.collection.distinct/#examples for other examples.

(2) Map Reduce

See https://docs.mongodb.com/manual/core/map-reduce/

(3) Aggregation pipeline

See https://docs.mongodb.com/manual/core/aggregation-pipeline/

The pipeline accepts a sequence of steps which result in staged-transformations of documents in a collection. It is called a pipeline because the output of one step is the input into the next step. Examples using ig:

```
db.posts.aggregate([
    { $unwind: '$usertags.nodes' },                        //flatten array
    { $group: {
        _id: '$usertags.nodes.user.username',              //for each username
        numTimesTagged: { $sum: 1 }                        //count +1
      } },
    { $match: { numTimesTagged: { $gt: 1 } } },            //filter count > 1
    { $sort: { numTimesTagged: -1 } },                     //sort descending
    { $limit: 5}                                           //top 5
])

db.posts.aggregate([
    { $match: {                                            //filter posts
        'likes.count': { $gte: 500 },                      //>= 500 likes
        comments: { $ne: null }                            //has comments
      } },
    { $project: {
        commentsCount: { $size: '$comments' }              //project comment count
      } },
    { $group: {
        _id: null,                                         //group by null = all
        avgCommentsCount: { $avg: '$commentsCount' }       //find average count
      } }
])
```

Notes:

- The `$size` operator used above is not the array operator introduced in 5.8 above. The aggregation `$size` operator used here returns the number of elements in an array: https://docs.mongodb.com/manual/reference/operator/aggregation/size/
- In aggregation pipelines, the `$` prefix on a field name means to use the value of the field. E.g. "`$comments`" means to get the value of the `comments` field, whereas "comments" means to use the string "comments". This is explained here under "Field Path and System Variables".

There are similarities in functionality between MongoDB's aggregation pipeline and SQL GROUP BY queries. See https://docs.mongodb.com/manual/reference/sql-aggregation-comparison/.

Note that this section is not examinable this semester.

7.1.1 Exercises

Using the example queries from the SQL comparison reference page as a starting point, write queries to calculate the results below.

Using ig.posts:

a) The number of posts per user (use owner.id)
b) The number of posts per user (use owner.id) for users that have more than 100 posts
c) The total (sum) number of likes per user
d) The average number of likes per user
e) The total number of usertags per user
f) For a) to e), only show the result with the highest count/total/average.
    - To achieve this, add two more steps to your pipeline: one to sort the count/total/average descending, and one to limit to 1 document.
g) For each user, find the largest like count across their posts
h) For each user, find the smallest like count across their posts
i) All unique values of location
    - `$group` will produce one group per unique value

Using Twitter.tweet:

j) The number of tweets per user
k) The total number of hashtags per user
l) For each user, find their latest tweet id. Assume ids are chronologically sequential, i.e. tweets with larger id numbers are created after those with smaller id numbers
m) How many users made retweets, i.e. made a tweet where retweeted_status exists and is not null?
n) Which user made the most retweets?
o) Using the logic below to find duplicates, how many duplicate tweets exist based on their id field? (Note that _id cannot be duplicated.) A duplicate on any field exists where a value for the field exists more than once. Duplicates can be discovered by:
    - grouping on the values for a field, and
    - counting the number of documents in the group, i.e.
      `$group: { _id: '$fieldName', count: { $sum: 1 } }`
    - then filtering to see groups with more than 1 document in the group, i.e.
      `$match: { count: { $gt: 1 } }`

l) Find how many duplicate ids exist in ig.posts. What do you observe about these "duplicated" documents?

8. Compass

MongoDB provides a profiling tool named MongoDB Compass. Download it here: https://www.mongodb.com/download-center#compass

Compass provides several functions:

- Data schema profiling: what does your data look like? What fields exist? What datatypes are used? What values are possible?
  Profile stats are based on a random sample of the collection (n = 1000)
- Performance profiling: resource consumption stats and long-running queries
- Basic CRUD GUI

Note that this section is not examinable this semester.

8.1.1 Exercises

Use Compass to profile:

a) Twitter.tweet:
   - How many documents are in the collection?
   - How much disk space does the collection use?

   Based on the 1000 documents sampled:
   - Which fields are always null?
   - Which fields are always false?
   - On average, how many hashtags are used per tweet?
   - Where a place is tagged, what fields does it contain?
   - Which fields have different datatypes across documents?
   - In the users sub-document, are there more nulls or empty strings for the description field? What about location?

b) ig.posts

   Based on the 1000 documents sampled:
   - What is the datatype of the comments field? List the full hierarchy of sub-documents
   - Which fields are rarely present? (Depending on the results of your random sample, Compass may not pick up that there are some documents that have no id field; these documents are not actual posts and contain different fields.)
   - What is the datatype of the usertags field? List the full hierarchy of sub-documents
   - In what circumstance does the video_viewed field exist?

9. Document change history

v1.1	2022-04-05	Updating downloadable files
v1.0	2022-01-27	Initial release