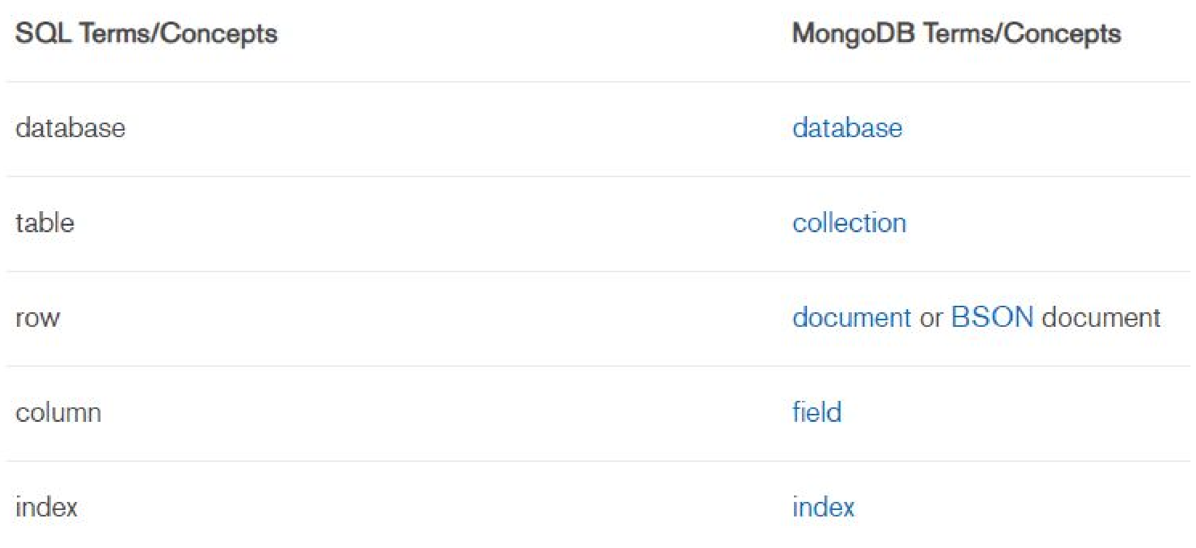
Exercise-1

Before we start, recall this jargon:



SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index

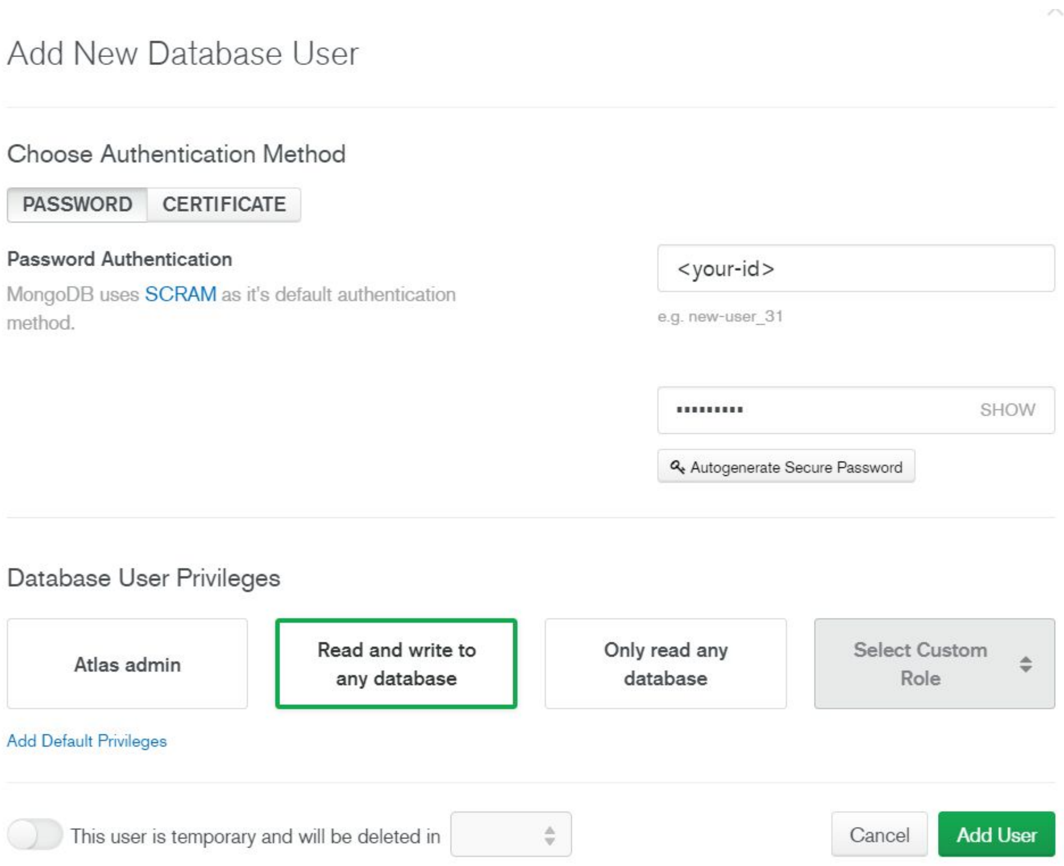
This exercise involves using the **mongo shell**. Mongo shell is already installed on your device. First complete MongoDB installation on your device by following the instructions written in **`\MongoDB Setup.pdf.`** Install client for MongoDB- MongoDB compass from here. Now register yourself at https://www.mongodb.com/cloud/atlas. Go for a free account. Sign up from google account is recommended.

Creating a cluster:

Select the shared cluster option as your choice.
Now select the following options:
Cloud Provider & Region: aws
Region : Singapore(ap-southeast-1)(recommended)
Give a name to your cluster- Mycluster (say)
After a successful cluster creation, do the following:

Database Access: Select 'Database Access' option from the options on the left side of the window.

Select +ADD NEW DATABASE USER. Now select the following options for the User:


Add New Database User

Choose Authentication Method

PASSWORD CERTIFICATE

Password Authentication

MongoDB uses SCRAM as it's default authentication method.

<your-id>

e.g. new-user_31

••••••••• SHOW

Autogenerate Secure Password

Database User Privileges

Atlas admin | Read and write to any database | Only read any database | Select Custom Role

Add Default Privileges

This user is temporary and will be deleted in

Cancel Add User

For ease, write user-id and password as the same. Now select Add User.

Network Access:

Go to the section 'Network Access' and click + ADD IP ADDRESS option. Click on allow access from anywhere and click confirm.

In the 'Clusters' section, go to connect option under your cluster name. Select Connect using mongo shell and copy the string displayed.

Instructions to open the shell are given below:

- open terminal
- paste the connection string.
    - (MongoDB uses default port as 27017 across devices)

A prompt '>' should appear indicating the mongo shell.
We will also be using a GUI client for MongoDB to view and edit data. To connect to the database from the client follow these steps:

- open application 'MongoDB Compass Community'

You will see the new connection window.

- copy the connection string for 'Connect using MongoDB compass' and paste it in the compass connection window.
- Don't forget to replace <password> with the actual password in the string. If the password has any special characters like '@' make sure they are 'URLsafe'. URLsafe passwords are encoded differently. For help see this.

Now, type `use <your-id>` in the terminal. You can also type 'db' to get the database name on which you are working.

Authenticate using `db.auth('<your-id>','<your-id>')`

CRUD operations- Create, Read, Update, Delete

In case of any clarifications in syntax, visit here

Create:

In the database that we have been provided, multiple collections can be created. Explicit creation of a collection can be done using:
`db.createCollection('posts')`
Creation of a collection is implicit upon insertion of a document in the collection. Use a text editor to type your commands with indentation and then copy them to shell for ease of use.

```
db.posts.insert({

        title:'Biden Takes Command of Race,
Winning Three States Including Michigan',
        body:'Joseph R. Biden Jr. widened his
advantage over Bernie Sanders in the Democratic
primary race, capturing Mississippi and Missouri
as well as Michigan, with support from both black
and white voters.',
        category: 'Politics',
        views:300,
        tags:['news','events'],
        user:{
            name: 'John Doe',
            status: 'author',
            },
        date: Date(),

})
```

Notice the aggregation of **user** collection within the **posts** collection. Also, notice the creation of the **_id field** for each inserted document.

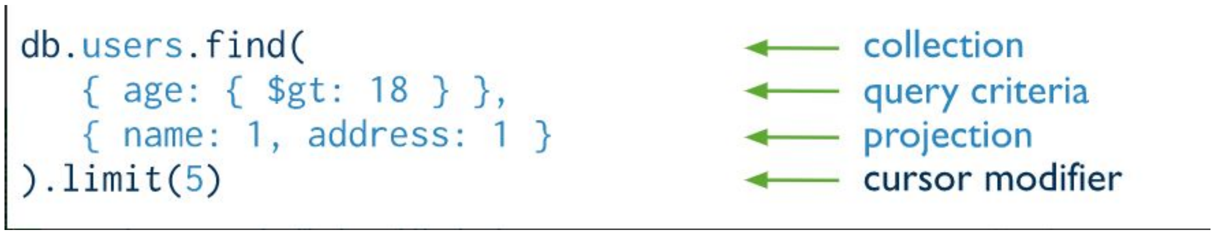
`show collections` is another useful command to view all the collections created. Don't forget to view the changes in the Compass Client as we type in commands.

`db.myCollection.insertMany(),db.myCollection.insertOne()` can be used to insert multiple,single documents.

Question 1: Write one mongo shell command to insert the posts given in `/data/posts/posts.txt`.

Read:

Read operations retrieves documents from a collection; i.e. queries a collection for documents. Mongo provides
`db.myCollection.find()`
This is equivalent to **select * from myCollection**
use **`.pretty()`** to get the output in indented form
`find()` takes 2 arguments



```
db.users.find(
   { age: { $gt: 18 } },
   { name: 1, address: 1 }
).limit(5)
```

Query criteria:

Query criteria contain the list of conditions on various fields of a document.

Equality conditions on a field:

These are specified as **`{<field>:<value>}`**
For example, if we want all the posts which belong to the category Politics, we write it as:

```
db.posts.find(
{
  category:'Politics'
}).pretty()
```

Other Conditions on a field:

Other conditions like >, <, in, etc. on a field are supported as **query operators.** They are specified as:

```
{
  <field1>:{
                <operator1>:<value1>
                },
  <field2>:{
                <operator2>:<value2>
                },
           ...
  }
```
For example, if we want all the posts with views > 5:
we write it as:
```
db.posts.find(
{
  views:{$gt: 5}
}).pretty()
```
Mongo also supports other operators like `$in,$lt,$ne`
The full list can be found here

There can be two or more such conditions discussed above in conjunction by ‘or’ logic or ‘and’ logic

OR condition:

These are specified as `{$or:[{<field1>:<value1>},{<field2>:value2}]}`
For example, if we want all the posts which either belong to the category ‘Politics’ or have views >= 10, we write it as:
```
  db.posts.find({
$or:[ {
      views:{$gte: 10}
      },
      {
      category:'Politics'}]}).pretty()
```

AND condition:

AND conditions are simply specified as:

`{<field1>:<value1>,<field2>:<value2>,...}`

Conditions on Embedded Documents:

The nested fields of a document are referred to using the dot '.' operator as `field.nestedfield`. A way around this is that the entire document can be specified with query conditions for each nested field. For example, the following are two equivalent ways of getting all the posts with the author - Vincent Vega :

```
db.posts.find({
'user.name':'Vincent Vega',
'user.status':'author'
}).pretty()
```

```
db.posts.find({
user:{
      name:'Vincent Vega',
      status:'author'
  }
}).pretty()
```

Question 2: Write a mongo shell command equivalent to the query: `SELECT * FROM posts WHERE category= "Politics" OR ( views <= 10 AND user_name== 'Jules Winnfield')`

Querying an Array:

An array field can be queried in several ways:

Matching entire array:

The entire array can be written as value to the field in the query expression as :

`{<arrayfield>:<array>,...}`

Query by element:

Query operators can be specified for an array field the same way it is specified to any other field.

For Example:

```
db.collection.find( { <arrayfield>: {
                          $gt: 15,
                          $lt: 20 }
                          } )
```

returns all documents in which the array field has at least one element satisfying first condition and other element satisfying the second condition.

Other ways of querying an array can be found here in detail. The same can be extended to an array of nested documents as well.

Question 3: Write a mongo shell command to get all the posts which have their first tag as news and has one other tag.

Projection:

projection is about the SELECT part of a query. It is specified in either in full inclusion or full exclusion as follows:

For the posts collection:

```
db.posts.find({},
{title:1,body:1,category:1,date:1},
).pretty()

db.posts.find({},
{views:0,tags:0,user:0},
).pretty()
```

return the same columns. However, there is an option to suppress the `_id` field by specifying `_id:0` while using full inclusion.

`.limit()` can be used to limit the no. of documents in the query output. Remember that the query is always a cursor and is expanded with a limit of 20 always. In cases of many documents, the cursor needs to be iterated.

Update:

`db.collection.updateOne()`, `db.collection.updateMany()` are the functions here. MongoDB has a concept of update operators just like query operators. Update queries are written as:

```
{<update_operator1>:
{<field11>:<value11>,<field21>:<value21>,...},
<update_operator2>:
{<field21>:<value21>,<field22>:<value22>,...}...}
```

`$set` is a commonly used operator. Full list is available here

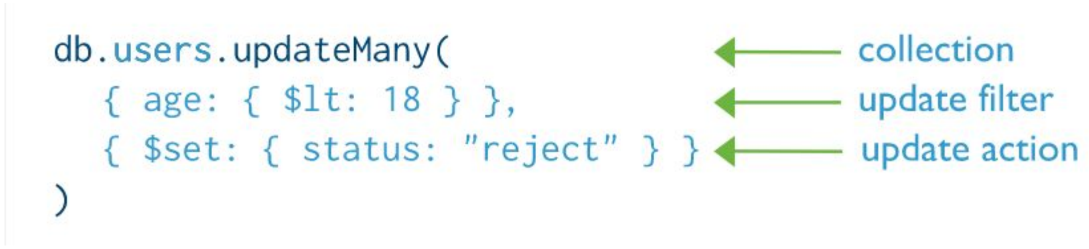
Update functions also take the query criteria as input and so the documents can be selectively updated.

Question 4: Add a new tag ‘UnitedStates’ to the post written by ‘Jules Winnfield’

MongoDB also provides the functionality of complete replacement of a document through **`db.collection.replaceOne()`**

Both replace and update functions have the option of **`{upsert: true}`**(upsert is short for update + insert. It creates a new document with the given field values)

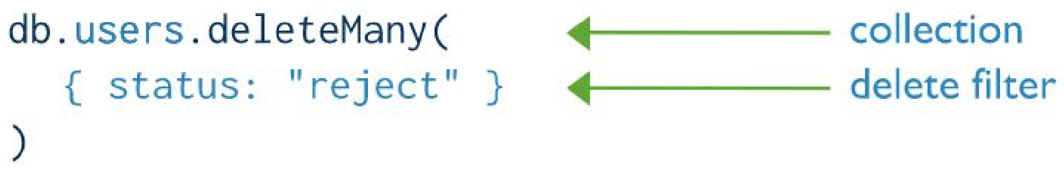
To show a general update function:



```
db.users.updateMany(
   { age: { $lt: 18 } },
   { $set: { status: "reject" } }
)
```

Delete:

Delete operations are provided by the db.collections.deleteOne() or deleteMany() functions. One has to specify the conditions through query operator for different fields to selectively delete those documents that satisfy the conditions. Structure of delete function.



```
db.users.deleteMany(
   { status: "reject" }
)
```

Bulk Write Operations:

MongoDB supports multiple operations at once through the **`db.collection.bulkWrite()`** function. The best part is that one can chain together insertOne, update, replaceOne, delete operations. There are two ways to execute bulk operations- ordered or unordered. This option is provided because MongoDB may try to optimise the writes in different sharded collections if the order of execution is not important. By default operations in **bulkWrite()** are **ordered.** An example to add a post, update the same, replacing it and then deleting it is given in

`/exercise1/soln/id_ex1.txt` assuming the title field is unique for every post.

Question 5: Write a single mongo shell command to do the following actions:

- **Increase the view count of each post by 1**
- **Insert the post given in the file `/exercise1/soln/id_ex1.txt (only the insert part)`**
- **delete all posts that do not have an author**
- **update all tags named ‘Facebook’ to ‘facebook’**

in that order.
Check the difference in case of unordered writes as well.

Strategies for Bulk Inserts to a Sharded Collection:

MongoDB uses the shard key to distribute the collection's documents across shards. The shard key consists of a field or fields that exist in every document in the target collection. You choose the shard key when sharding a collection. The choice of shard key cannot be changed after sharding.

- **Pre-Split the Collection**

  Specifying the shard key and splitting the collection into multiple chunks before insertion.

- **Unordered Writes to `mongos`**
  using `ordered: false` field in `bulkWrite()`. This will cause the mongos to send writes to different shards simultaneously.
- **Avoid Monotonic Throttling**

If your shard key increases monotonically during an insert, then all inserted data goes to the last chunk in the collection, which will always end up on a single shard.

Therefore, the insert capacity of the cluster will never exceed the insert capacity of that single shard.

If your insert volume is larger than what a single shard can process, and if you cannot avoid a monotonically increasing shard key, then consider the following modifications to your application:

- Reverse the binary bits of the shard key. This preserves the information and avoids correlating insertion order with an increasing sequence of values.
- Swap the first and last 16-bit words to “shuffle” the inserts.

Aggregation:

MongoDB has options for aggregation modelled via a data processing pipeline called the aggregation pipeline.

Aggregation Pipeline:

The Aggregation pipeline has stages each of which are represented by an aggregation operator. The **`match`** operator filters the documents on which aggregation is applied and the **`group`** operator groups documents to form the resultant document.

In the example,

```
db.orders.aggregate([
   { $match: { status: "A" } },
   { $group: { _id: "$cust_id", total: { $sum:
"$amount" } } }
])
```

First Stage: The **`$match`** stage filters the documents by the status field and passes to the next stage those documents that have **`status`** equal to **`"A"`**.

Second Stage: The `$group` stage groups the documents by the `cust_id` field to calculate the `sum` of the amount for each unique `cust_id`.

The match stage can be written in the same way the update filter and delete filters are provided. A generic group stage looks like this:

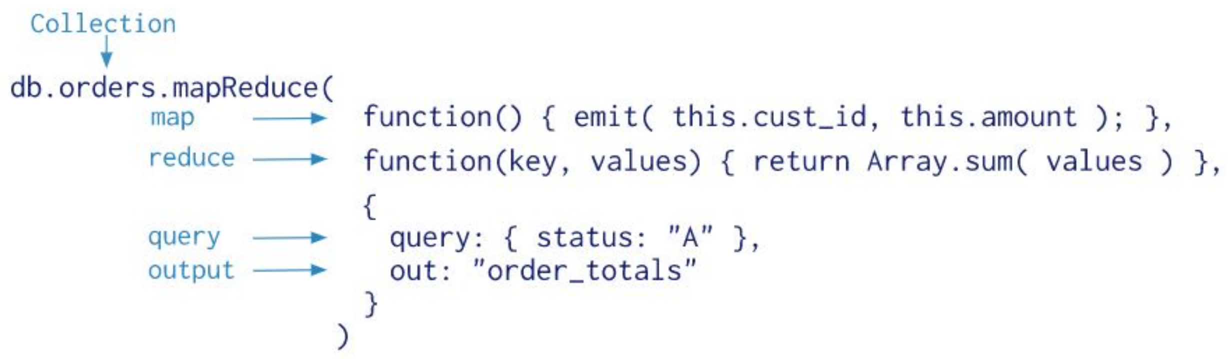
```
{
  $group:
    {
      _id: <expression>, // Group By Expression
      <field1>: { <accumulator1> : <expression1>
},
      ...
    }
 }
```

Question 6:Write a mongo shell command to get the category-wise sum of total views of the posts.

There are many other stages in the aggregation pipeline other than match and group, details of which are given here.
MongoDB supports a vast number of aggregation operators for elementary data types, arrays, sets etc. details of which are given here. Check out $unwind which is useful for iterating through arrays.
Note: A pipeline can have more than two stages and hence is represented as a list. Aggregation here is very much different than in SQL and is much more flexible.

Map-Reduce:

For map-reduce operations, MongoDB provides the **`db.collection.mapReduce()`** database command. Structure of a mapReduce command is as follows:



```
db.orders.mapReduce(
                    function() { emit( this.cust_id, this.amount ); },
                    function(key, values) { return Array.sum( values ) },
                    {
                      query: { status: "A" },
                      out: "order_totals"
                    }
                  )
```

map and reduce functions are just javascript functions. So, if you do not know already, lookup how basic javascript functions are written. Map function emits <k,v> pairs and reducer returns a value for every <k,v> pair.

Query operator is like a filter. The MapReduce operation will only act on documents satisfying this condition(s).

Output field defines where the output of the MapReduce operation is written down- A new collection or an existing collection or just inline on the shell.

The snapshot of the mapReduce function shown above dumps the output into the collection ‘order_totals’. If it exists already, the collection is overwritten.

mongo shell also supports the declaration of variables, functions programmatically. One can declare functions, variables beforehand and use them like this:

```
var myfunction=function(key,values)
            {
                var i;
                var max;
                for(i=0;i<values.length;i++)
                {
                    ...
                };
                return max;
            }
```

Question 7: Write a mongo shell command which lists the total number of posts, total number of views for each tag using mapReduce()

Concurrency:

All operations like insert, update, delete etc. in MongoDB are atomic for a document. During the mapReduce operation, map-reduce takes the following locks:

- The read phase takes a read lock. It yields every 100 documents.
- The insert into the temporary collection takes a write lock for a single write.
- If the output collection does not exist, the creation of the output collection takes a write lock.
- If the output collection exists, then the output actions (i.e. merge, replace, reduce) take a write lock. This write lock is global, and blocks all operations on the mongod instance.

In the case of multiple MapReduce jobs, subsequent jobs can be performed on the resultant collection. MongoDB also provides the option of specifying any further aggregation to the `out` parameter by specifying a function to the `finalize` parameter as follows:

```
db.sessions.mapReduce( mapFunction,
                       reduceFunction,
                       { query: { ts: { $gt:
ISODate('2011-11-05 00:00:00') } },
             out: { reduce: "session_stat" },
                   finalize: finalizeFunction
} );
```

Indexes:

MongoDB, just like most databases, allows indexing and that too on any field or subfield in the document. Indexes speed up ranged based and equality-based searches. All the update, delete, query operations can make use of the indexes including the aggregation pipeline. Indexes once created are used automatically by MongoDB. The query optimiser of MongoDB decides which index/indices to use. It might also use an intersection of multiple indices.

Indexes are created on collections and a default **unique** index is created on the `_id` field. Indexes with `unique: True` enforce the uniqueness of the field in the collection

To create an index, MongoDB has the command:
`db.collection.createIndex( <key and index type specification>, <options> )`
Indexes can also be created on multiple fields combined(known as compound Index) and can also be given human-readable names. For Example:

```
db.products.createIndex(
  { item: 1, quantity: -1 } ,
  { name: "query for inventory" }
)
```
creates a compound index on fields item and quantity in the collection products. The index is constructed by sorting the item field in ascending order (represented by a ‘1’) and the documents with the same item field are sorted in descending order of quantity (represented by a ‘-1’)

There are many different Index types available for creation in MongoDB, specialised for a given data type. All further details can b found here

MongoDB boasts off on its rich query API as you might have noticed. As a result, there are options provided for each and every case and user preference. Only the common ones are mentioned here. More can be found at the official documentation site. This Concludes the Mongo Shell exercise.