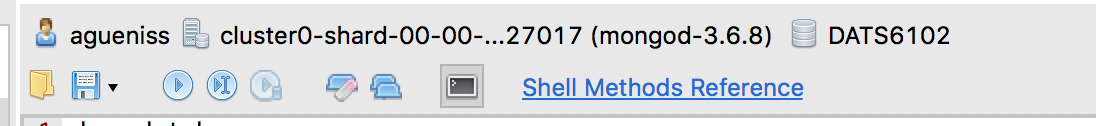
**MongoDB Week 2 Lab**

**Part I – Working with Collections in Studio 3T utilizing the shell**

**Important Note: Make sure you are executing statement at cursor and not entire script, or you will get duplicates by executing statements multiple times!!**

****

1. Navigate to IntelliShell and show databases running on your mongo server



show dbs

1. Navigate to a non-existing database **flights** and observe that you can switch to this database. Once you start entering data into it, mongoDB will create it implicitly

use flight\_data

show dbs

1. Observe how collection will be created implicitly once you start interacting with it (flight\_data collection does not exist yet). Observe unique id being automatically inserted. Execute show dbs again to see flights database created.

db.flight\_data.insertOne({

"departureAirport": "MUC",

"arrivalAirport": "SFO",

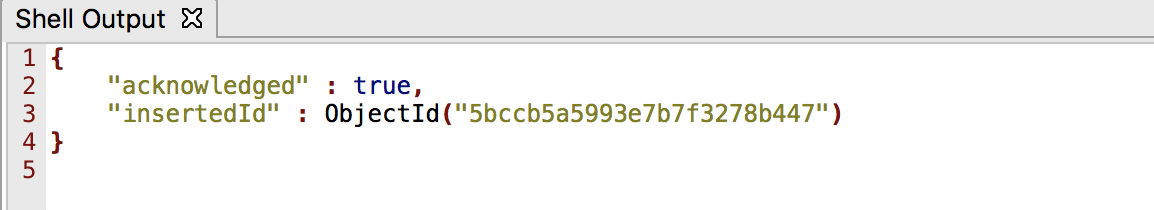
"aircraft": "Airbus A380",

"distance": 12000,

"intercontinental": true

})

You should see the following message indicating that insert was successful. Note that **\_id** was automatically added for you (your \_id will be different):



show dbs

1. Query flights data – a) select all records (although you only have 1 now) b) select using **pretty()** function, which produces output in formatted way:
2. db.flight\_data.find**()**
3. db.flight\_data.find**()**.pretty**()**
4. Re-execute the same insert and observe how MongoDB inserts different unique **\_id** with the same document:

db.flight\_data.insertOne({

"departureAirport": "MUC",

"arrivalAirport": "SFO",

"aircraft": "Airbus A380",

"distance": 12000,

"intercontinental": true

})

db.flight\_data.find**()**.pretty**()**



1. Delete extra document that we just inserted (your object id might be different), observe the deletion:

db.flight\_data.deleteOne**({**"\_id"**:** ObjectId**(**"5bcc9fe6467a552285f3e131"**)})**

db.flight\_data.find**()**.pretty**()**

1. Insert a document into the flight\_data collection with a different schema, observe the changes (two documents in the same collection don’t have to have the same schema!):

db.flight\_data.insertOne**({**"departureAirport"**:** "TXL"**,** "arrivalAirport"**:** "LHR"**})**

db.flight\_data.find**()**.pretty**()**

1. Insert a document into the flight\_data collection with your own \_id (do not have to use auto-generated one) and make it a String this time. Observe how this document got inserted:

db.flight\_data.insertOne**({**

"departureAirport"**:** "STM"**,**

"arrivalAirport"**:** "CHI"**,**

"aircraft"**:** "Airbus A382"**,**

"distance"**:** 16000**,**

"intercontinental"**:** false**,**

"\_id" **:** "anya-test-01"

**})**

db.flight\_data.find**()**.pretty**()**

1. Try to add the same document to flight\_data collection with the same \_id and observe MongoDB throwing errors:

db.flight\_data.insertOne**({**

"departureAirport"**:** "STM"**,**

"arrivalAirport"**:** "CHI"**,**

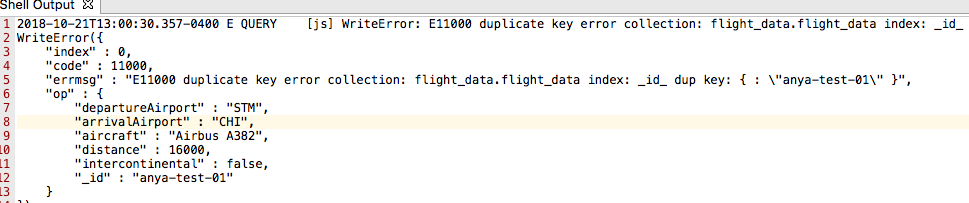
"aircraft"**:** "Airbus A382"**,**

"distance"**:** 16000**,**

"intercontinental"**:** false**,**

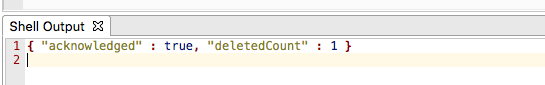
"\_id" **:** "anya-test-01"

**})**



1. Delete document with MUC airport, observe changes:

db.flight\_data.deleteOne**({**"departureAirport"**:** "MUC"**})**



db.flight\_data.find**()**.pretty**()**

1. Update document with distance filter 16000 and add a new field **marker with value “delete”.** Use $set operator to make this change. If the value for field **marker** exist, MongoDB would have updated it. Since it does not exist, it will insert it. Observe your update.

db.flight\_data.updateOne**({**"distance"**:** 16000**},{**$set**:** **{**"marker"**:** "delete"**}})**

db.flight\_data.find**()**.pretty**()**

1. Add **marker** field to all of the documents (yes, we only have two right now). Use empty curly braces {} to update all documents, so you are not passing any filters. Observe your update

db.flight\_data.updateMany**({},{**$set**:** **{**"marker"**:** "update"**}})**

1. Delete all documents where marker = update. Observe your delete, you should have no documents.

db.flight\_data.deleteMany**({**"marker"**:** "update"**})**

db.flight\_data.find**()**.pretty**()**

1. Insert multiple documents into flight\_data collection by passing an array (**[ ])** observe your insert:

db.flight\_data.insertMany**([**

**{**

"departureAirport"**:** "MUC"**,**

"arrivalAirport"**:** "SFO"**,**

"aircraft"**:** "Airbus A380"**,**

"distance"**:** 12000**,**

"intercontinental"**:** true

**},**

**{**

"departureAirport"**:** "LHR"**,**

"arrivalAirport"**:** "TXL"**,**

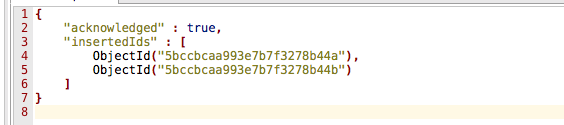
"aircraft"**:** "Airbus A320"**,**

"distance"**:** 950**,**

"intercontinental"**:** false

**}**

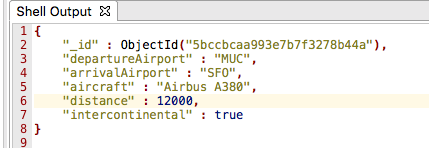
**])**



db.flight\_data.find**()**.pretty**()**

1. Query flight\_data collection by passing document to find() in order to filter the collection. We are getting all flights where intercontinential is set to true. We are also using pretty() function here.

db.flight\_data.find**({**"intercontinental" **:** true**})**.pretty**()**

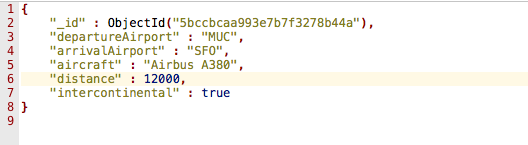


1. Find all data in flight\_data collection with distance of 12000:

db.flight\_data.find**({**"distance" **:** 12000**})**.pretty**()**

1. Find all data in flight\_data collection with distance > 10000. You will pass another object to find() as a value for distance by using **$gt** operator

db.flight\_data.find**({**"distance" **:** **{**$gt**:** 10000**}})**.pretty**()**

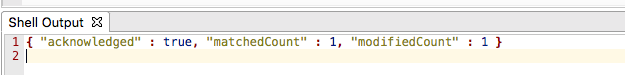


1. Utilize findOne() function to get first element from your search. Although both documents match distance > 900, only one is returned. Also, you cannot use pretty() function with findOne().

db.flight\_data.findOne**({**"distance" **:** **{**$gt**:** 900**}})**

1. Update flight\_data collection with ObjectId of your choice, add new field **delayed** and set it to **true.** Use updateOne() function and observe your update. **Your ObjectId will be different, so don’t just copy/paste this one, copy/paste YOURS**

db.flight\_data.updateOne**({**"\_id" **:** ObjectId**(**"5bccbcaa993e7b7f3278b44b"**)},** **{**$set**:** **{**delayed**:** true**}})**



db.flight\_data.find**()**.pretty**()**

1. Now use update() function to change value of **delayed** to **false**

db.flight\_data.update**({**"\_id" **:** ObjectId**(**"5bccbcaa993e7b7f3278b44b"**)},** **{**$set**:** **{**delayed**:** false**}})**

db.flight\_data.find**()**.pretty**()**

1. Now use **update()** function without **$set** syntax and see how it will overwrite all of the data in your document:
2. Observe that you have two documents

db.flight\_data.find**()**.pretty**()**

1. Issue update() without $set for one of them (grab one of the ObjectIds)

db.flight\_data.update**({**"\_id" **:** ObjectId**(**"5bccbcaa993e7b7f3278b44b"**)},** **{**delayed**:** false**})**

db.flight\_data.find**()**.pretty**()**

****

Notice how your first document only has \_id and delayed fields now. This is the behavior of update() vs updateOne() and updateMany(). Update() without $set operator will take existing object and replace it with new object (which in this case only has \_id and delayed fields). You have to use $set operator with updateOne() and updateMany(). **That is the difference between these functions.**

1. Use **replace()** function to replace a document in flight\_data collection. Observe your operation. Take ObjectId of the document from #21, the one that only has **delayed=false data**

db.flight\_data.replaceOne**({**"\_id" **:** ObjectId**(**"5bccbcaa993e7b7f3278b44b"**)},**

**{**

"departureAirport"**:** "LHR"**,**

"arrivalAirport"**:** "TXL"**,**

"aircraft"**:** "Airbus A320"**,**

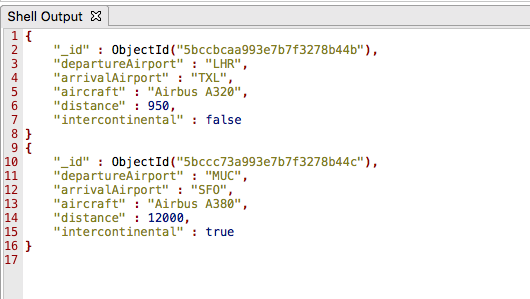
"distance"**:** 950**,**

"intercontinental"**:** false

**}** **)**



db.flight\_data.find**()**.pretty**()**



1. Insert data into a new collection **passengers** using insertMany() function as follows:

show dbs

use flight\_data

db.passengers.insertMany**(**

**[**

**{**

"name"**:** "Max Schwarzmueller"**,**

"age"**:** 29

**},**

**{**

"name"**:** "Manu Lorenz"**,**

"age"**:** 30

**},**

**{**

"name"**:** "Chris Hayton"**,**

"age"**:** 35

**},**

**{**

"name"**:** "Sandeep Kumar"**,**

"age"**:** 28

**},**

**{**

"name"**:** "Maria Jones"**,**

"age"**:** 30

**},**

**{**

"name"**:** "Alexandra Maier"**,**

"age"**:** 27

**},**

**{**

"name"**:** "Dr. Phil Evans"**,**

"age"**:** 47

**},**

**{**

"name"**:** "Sandra Brugge"**,**

"age"**:** 33

**},**

**{**

"name"**:** "Elisabeth Mayr"**,**

"age"**:** 29

**},**

**{**

"name"**:** "Frank Cube"**,**

"age"**:** 41

**},**

**{**

"name"**:** "Karandeep Alun"**,**

"age"**:** 48

**},**

**{**

"name"**:** "Michaela Drayer"**,**

"age"**:** 39

**},**

**{**

"name"**:** "Bernd Hoftstadt"**,**

"age"**:** 22

**},**

**{**

"name"**:** "Scott Tolib"**,**

"age"**:** 44

**},**

**{**

"name"**:** "Freddy Melver"**,**

"age"**:** 41

**},**

**{**

"name"**:** "Alexis Bohed"**,**

"age"**:** 35

**},**

**{**

"name"**:** "Melanie Palace"**,**

"age"**:** 27

**},**

**{**

"name"**:** "Armin Glutch"**,**

"age"**:** 35

**},**

**{**

"name"**:** "Klaus Arber"**,**

"age"**:** 53

**},**

**{**

"name"**:** "Albert Twostone"**,**

"age"**:** 68

**},**

**{**

"name"**:** "Gordon Black"**,**

"age"**:** 38

**}**

**]**

**)**

db.passengers.find**()**.pretty**()**

**Notice that at the end of your document you see the following:**



**Type “it” for more – find() brings back the cursor, not all of the data.**

**Now execute toArray() function, which will bring all the data**

db.passengers.find**()**.toArray**()**

1. Use projections to bring back **name** only from **passengers** collection by passing a) empty document to find() method by using **{}** as we want to get all documents and b) pass projection that we want to filter by – name. Here value 1 means data – include it in the data you are returning.

db.passengers.find**({},** **{**name**:** 1**})**.pretty**()**

**Notice that ObjectId is also returned with your data, that is default behavior**

**To exclude \_id use 0 instead of 1 to explicitly exclude ObjectId**

db.passengers.find**({},** **{**name**:** 1**,** \_id**:** 0**})**.pretty**()**

1. Update all documents in **flight\_data** collection (by passing empty array **{} )** to set **status** field to another (embedded) document. Observe the update:

db.flight\_data.updateMany**({},** **{**$set**:** **{**status**:** **{**description**:** "on-time"**,** lastUpdated**:** "2 hours ago" **}}})**

db.flight\_data.find**()**.pretty**()**

1. Add another nested document to **flight\_data** collection in the same way as in #25 with field **details**. Observe your update:

db.flight\_data.updateMany**({},** **{**$set**:** **{**status**:** **{**description**:** "on-time"**,** lastUpdated**:** "2 hours ago"**,** details**:{**responsible**:** "John Smith"**}** **}}})**

db.flight\_data.find**()**.pretty**()**

1. Utilize arrays by updating **passengers** collection, add hobbies field to specific name in a form of an array. Remember that an array can have any values in it, including other documents. Observe your update.

db.passengers.updateOne**({**name**:** "Manu Lorenz"**},** **{**$set**:** **{**hobbies**:** **[**"sports"**,** "cooking"**,** "movies"**]}})**

db.flight\_data.find**()**.pretty**()**

1. Query for a specific passenger – Albert Twostone utilizing find() method:

db.passengers.find**({**"name" **:** "Albert Twostone"**})**.pretty**()**

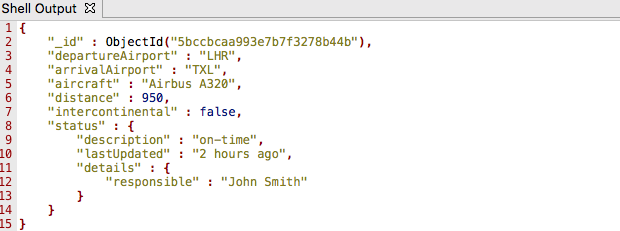
1. Query for a specific passenger’s hobbies (passenger name = Manu Lorenz):

db.passengers.findOne**({**"name" **:** "Manu Lorenz"**})**.hobbies

1. Query for all passengers with a hobby sports:

db.passengers.find**({**hobbies **:** "sports"**})**

1. Query for all data inside **flight\_data** collection with **description field = on\_time (**recall that description is a field in a nested document):



db.flight\_data.find**({**"status.description" **:** "on-time"**})**.pretty**()**

**Note that we access object status with dot notation as status.description and using “” around it.**

1. Drill down to the same status.details to get responsible field:

db.flight\_data.find**({**"status.details.responsible" **:** "John Smith"**})**.pretty**()**

**//RESTAURANTS COLLECTION – create separate database and import this //collection there**

1. Navigate to **RESTAURANTS** collection. Get all fields for restaurant\_id, name, borough and cuisine for all the documents in this collection.

db.restaurants.find**({},{**"restaurant\_id" **:** 1**,**"name"**:**1**,**"borough"**:**1**,**"cuisine" **:**1**})**.pretty**();**

1. Get all fields for restaurant\_id, name, borough and cuisine for all the documents in this collection, but do not include \_id for the documents:

db.restaurants.find**({},{**"restaurant\_id" **:** 1**,**"name"**:**1**,**"borough"**:**1**,**"cuisine" **:**1**,**"\_id"**:**0**});**

1. Get all the restaurants in Bronx

db.restaurants.find**({**"borough"**:** "Bronx"**})**.pretty**();**

1. Display first five restaurants in Bronx in this collection

db.restaurants.find**({**"borough"**:** "Bronx"**})**.limit**(**5**);**

1. Display next five restaurants after skipping first five that are in Bronx

db.restaurants.find**({**"borough"**:** "Bronx"**})**.skip**(**5**)**.limit**(**5**)**.pretty**();**

1. Display restaurants with score less than 90

db.restaurants.find**({**grades **:** **{** $elemMatch**:{**"score"**:{**$lt **:** 90**}}}})**.pretty**();**

1. Display restaurant\_id, name and grades for those restaurants where the 2nd element of grades array contains a grade of “A” and score 9 on an ISODate of "2014-08-11T00:00:00Z"

db.restaurants.find**(**

**{** "grades.1.date"**:** ISODate**(**"2014-08-11T00:00:00Z"**),**

"grades.1.grade"**:**"A" **,**

"grades.1.score" **:** 9

**},**

**{**"restaurant\_id" **:** 1**,**"name"**:**1**,**"grades"**:**1**}**

**)**.pretty**();**

1. Display restaurant\_id, name address and georgraphical location for those restaurants who’s 2nd element of coord array contains a value which is > 42 and up to 52

db.restaurants.find**(**

**{**

"address.coord.1"**:** **{**$gt **:** 42**,** $lte **:** 52**}**

**},**

**{**"restaurant\_id" **:** 1**,**"name"**:**1**,**"address"**:**1**,**"coord"**:**1**}**

**)**.pretty**()**

1. Display the name of restaurants in descending order

db.restaurants.find**()**.sort**(**

**{**"name"**:**-1**}**

**)**.pretty**();**

**//Persons collection (note: JSON does not support comments, JavaScript does)**. //Create separate database Person, import this collection there

//Persons collection has 5000 documents, if your queries start running slow, cancel out and re-run.

1. Find all documents in **persons** collection where first name is not victor

db.persons.find**({** "name.first"**:** **{** $ne**:** "victor" **}})**.pretty**()**

1. Navigate to **persons** collection, get all documents with gender = female using aggregation:

db.getCollection**(**"persons"**)**.find**({})**.pretty**()**

db.getCollection**(**"persons"**)**.aggregate**([{**$match**:** **{**gender**:** "female"**}}])**.pretty**()**

1. Use **persons** collection and get all males living in each of the states. Use **$group** stage of aggregation framework to group data in **persons** collection for all males by **state** field in **location** object. We are grouping by location/state**:**

****

db.persons.aggregate**([**

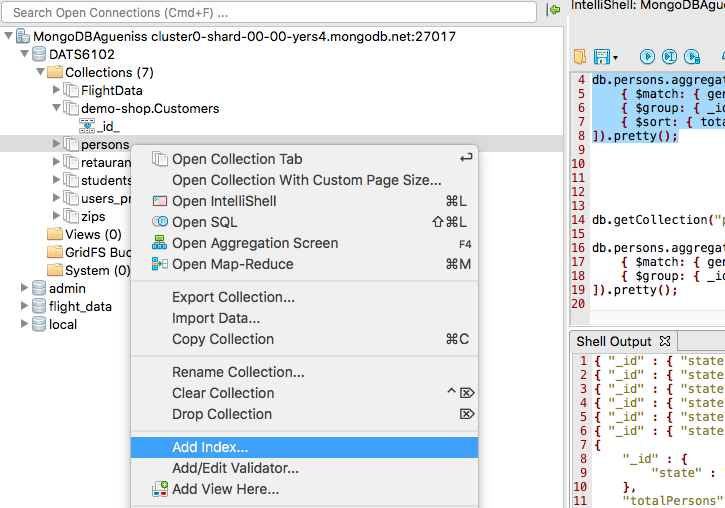
**{** $match**:** **{** gender**:** 'male' **}** **},**

**{** $group**:** **{** \_id**:** **{** state**:** "$location.state" **},** totalPersons**:** **{** $sum**:** 1 **}** **}** **}**

**])**.pretty**();**

**Note the following:**

1. Parameters to $group: \_id defines fields by which we are grouping
2. We are using a document as a value for \_id {state: “$location.state”} 🡪 will allow multiple fields by which you want to group
3. totalPersons – new key added to each document
4. we are using $sum aggregation function
5. $sum: 1 🡪 if we have 5 people in location/state sum will be incremented by 1 times 5
6. In a result we used group to merge our documents into new documents to get \_id, location/state and totalPersons
7. Let’s sort the above aggregation by totalPersons field by using **$sort** stage. Let’s first create an index on location.state (we are in a free tier, things are not optimized and slow):

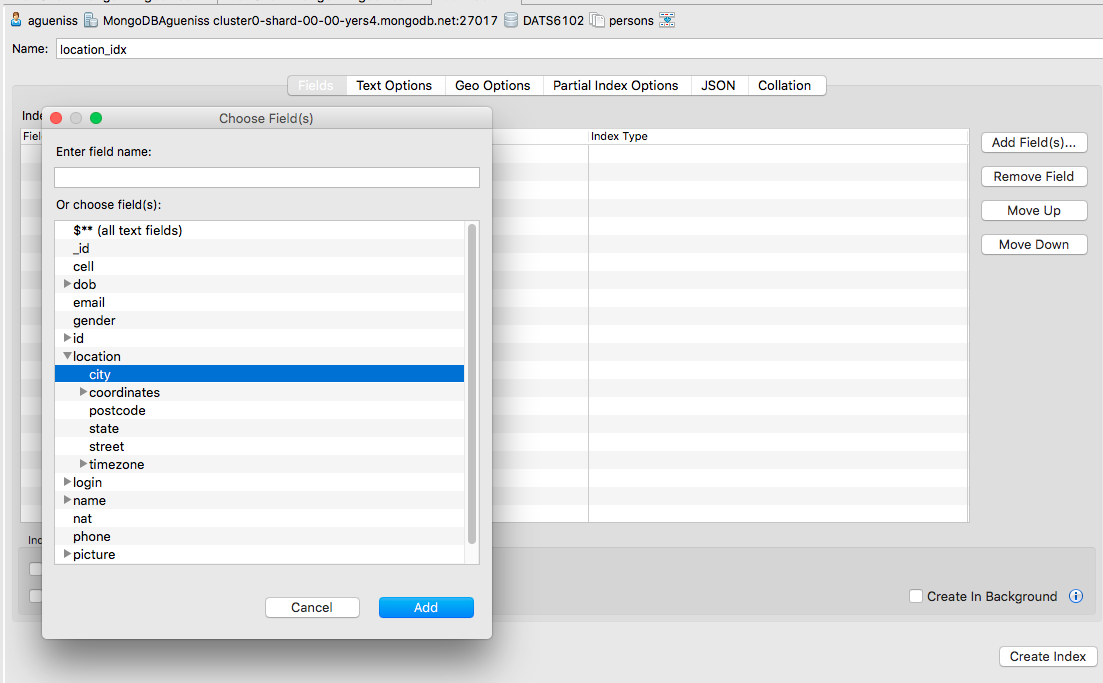
****

**Name: city\_idx**

**Add Field 🡪 location 🡪 city**

**Add**

**Create Index**

****

db.persons.aggregate**([**

**{** $match**:** **{** gender**:** 'female' **}** **},**

**{** $group**:** **{** \_id**:** **{** state**:** "$location.state" **},** totalPersons**:** **{** $sum**:** 1 **}** **}** **},**

**{** $sort**:** **{** totalPersons**:** -1 **}** **}**

**])**.pretty**();**

1. Use **$project** aggregate stage to not show \_id, only displaying gender and name (display it in one field using **$concat).** Use $substrCP and $strLenCP to capitalize first and last name.

db.persons.aggregate**([**

**{**

$project**:** **{**

\_id**:** 0**,**

gender**:** 1**,**

fullName**:** **{**

$concat**:** **[**

**{** $toUpper**:** **{** $substrCP**:** **[**'$name.first'**,** 0**,** 1**]** **}** **},**

**{**

$substrCP**:** **[**

'$name.first'**,**

1**,**

**{** $subtract**:** **[{** $strLenCP**:** '$name.first' **},** 1**]** **}**

**]**

**},**

' '**,**

**{** $toUpper**:** **{** $substrCP**:** **[**'$name.last'**,** 0**,** 1**]** **}** **},**

**{**

$substrCP**:** **[**

'$name.last'**,**

1**,**

**{** $subtract**:** **[{** $strLenCP**:** '$name.last' **},** 1**]** **}**

**]**

**}**

**]**

**}**

**}**

**}**

**])**.pretty**();**

**//DEMO-SHOP.CUSTOMERS COLLECTION (you should have it from Lab 1)**

1. Navigate to **demo-shop.Customers** collection. Find customers whose name is not “Susan” nor “Whitney”

db.getCollection**(**"demo-shop.Customers"**)**.find**({** name**:** **{** $nin**:** **[**"Susan"**,** "Whitney"**]** **}})**

1. Find persons that are either have last name Grey or first name Cheryl

db.getCollection**(**"demo-shop.Customers"**)**.find**({** $or**:** **[{** last**:** "Grey" **},** **{** first**:** "Cheryl" **}]})**

**Part II – Working with Schema Validator**

1. Create a new collection and add schema validation to it as follows:
2. create new collection called **posts**
3. use **validator** key and pass a sub-document to it, where you define the schema
4. use **$jsonSchema** key to hold the actual schema:
   * set bsonType to “object” – everything that will be added to this collection will be a valid document/object
   * set **required** key, which is an array – defines names/fields in the document that will be part of the collection that are required. Make **text, title, comments and creator** fields required
   * set **properties** key (how every document that gets added to this collection should look like)
     1. Define **title** must be a required bsonType string
     2. Define **text** – must be a required bsonType string
     3. Define **creator –** must be a required bsonType objectid
     4. Define **comments** – must have a required bsonType of array. Must have **items** key (describes how items should look like)

db.createCollection**(**'posts'**,** **{**

validator**:** **{**

$jsonSchema**:** **{**

bsonType**:** 'object'**,**

required**:** **[**'title'**,** 'text'**,** 'creator'**,** 'comments'**],**

properties**:** **{**

title**:** **{**

bsonType**:** 'string'**,**

description**:** 'must be a string and is required'

**},**

text**:** **{**

bsonType**:** 'string'**,**

description**:** 'must be a string and is required'

**},**

creator**:** **{**

bsonType**:** 'objectId'**,**

description**:** 'must be an objectid and is required'

**},**

comments**:** **{**

bsonType**:** 'array'**,**

description**:** 'must be an array and is required'**,**

items**:** **{**

bsonType**:** 'object'**,**

required**:** **[**'text'**,** 'author'**],**

properties**:** **{**

text**:** **{**

bsonType**:** 'string'**,**

description**:** 'must be a string and is required'

**},**

author**:** **{**

bsonType**:** 'objectId'**,**

description**:** 'must be an objectid and is required'

**}**

**}**

**}**

**}**

**}**

**}**

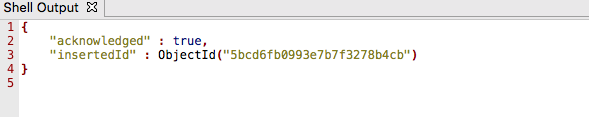
**}**

**});**

1. Add a document to posts collection and observe that it was added to the collection:

db.posts.insertOne**({**title**:** "My first post!"**,** text**:** "This is my first post!"**,** tags**:** **[**"new"**,** "tech"**],** creator**:** ObjectId**(**"507f1f77bcf86cd799439011"**),**

comments**:[{**text**:** "I like this post!"**,** author**:** ObjectId**(**"507f191e810c19729de860ea"**)}]})**

****

db.posts.findOne**()**

1. Try to insert something that would violate validator schema. Instead of ObjectId try to insert random number. Check to make sure that the post was not added after it failed validation:

db.posts.insertOne**({**title**:** "My first post!"**,** text**:** "This is my first post!"**,** tags**:** **[**"new"**,** "tech"**],** creator**:** ObjectId**(**"507f1f77bcf86cd799439011"**),**

comments**:[{**text**:** "I like this post!"**,** author**:** 12334**}]})**

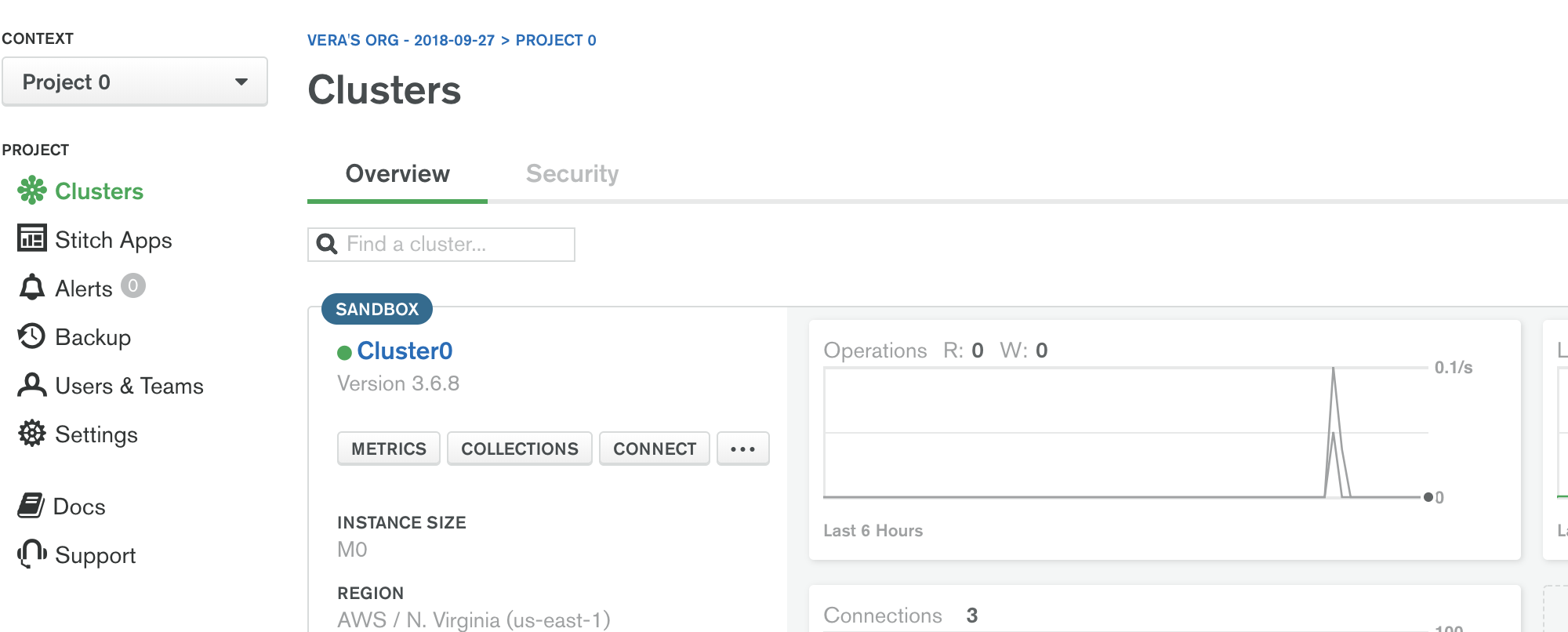
**You will get an error “Document Failed Validation”**

****

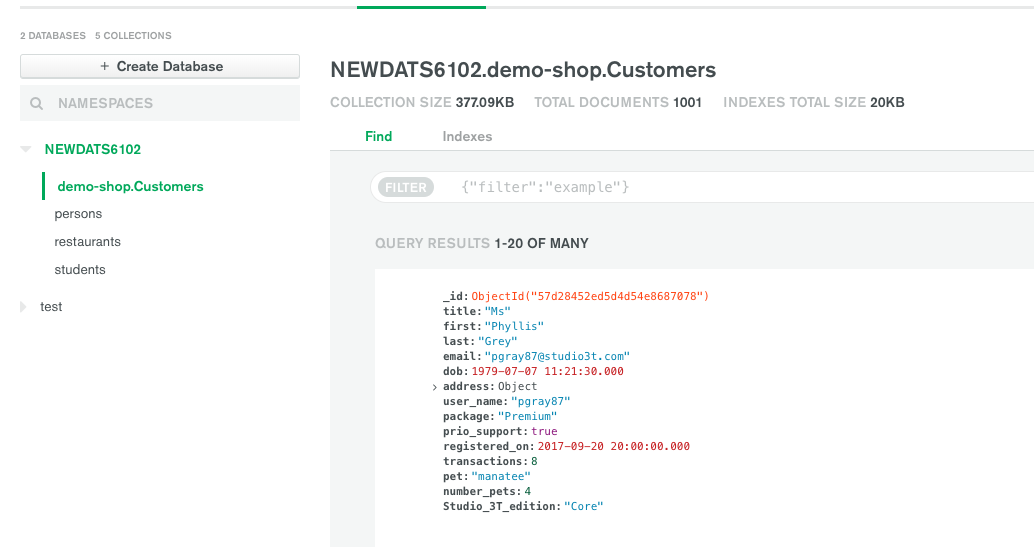
db.posts.findOne**()**

**Part III– Working with Data Explorer Within Atlas**

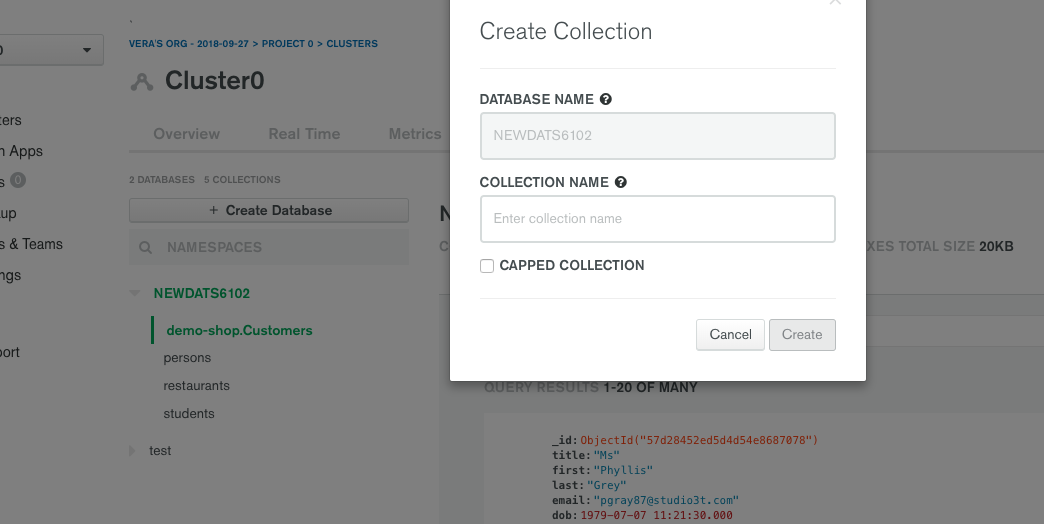
1. Navigate to your MongoDB Atlas account

****

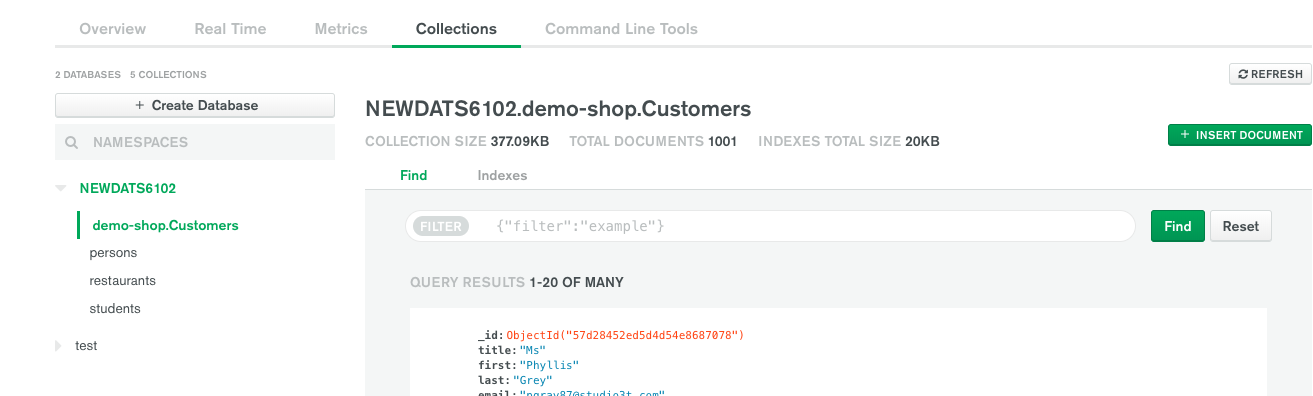
1. On the left observe all of the databases and their respective collections

****

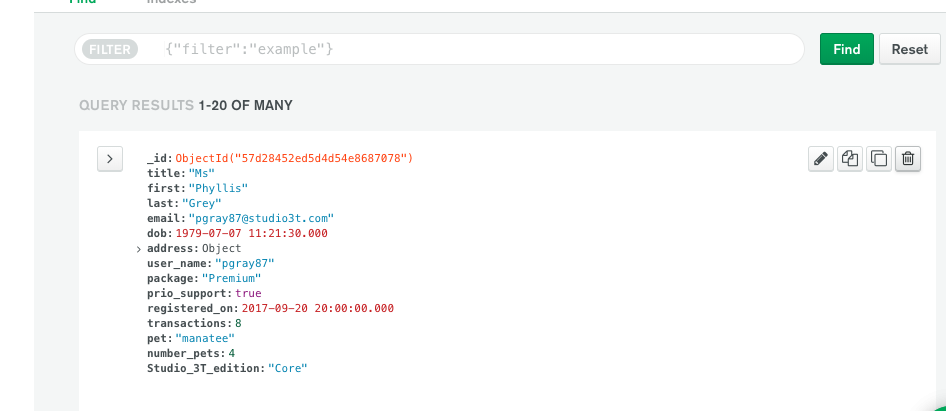
1. Observe that we can create a database and add/drop a collection here:

****

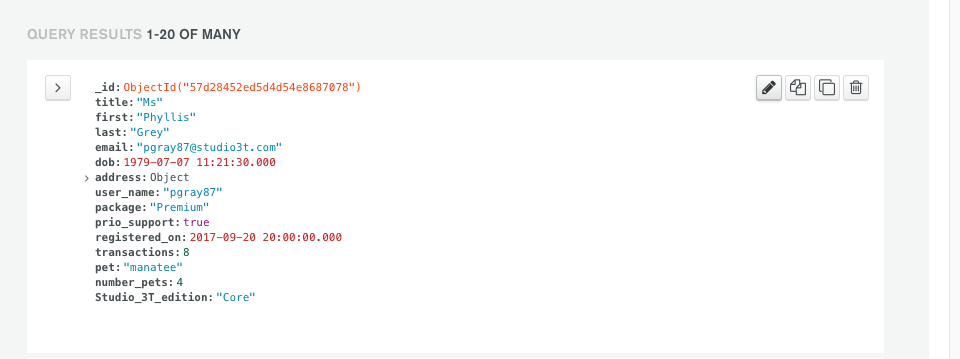
1. Observe that we can view all of the documents, add and filter (demo-shop.Customers)

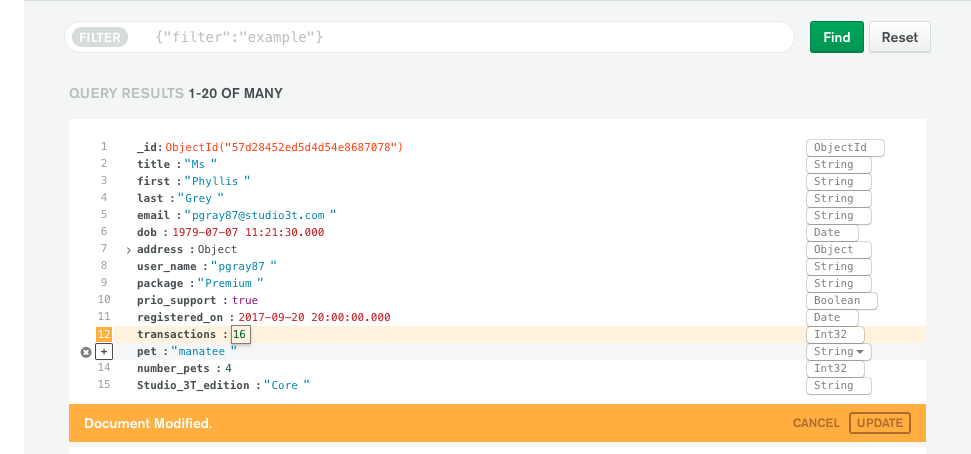
****

1. Drop first document from the collection:

****

1. Update number of transactions in a document:

****

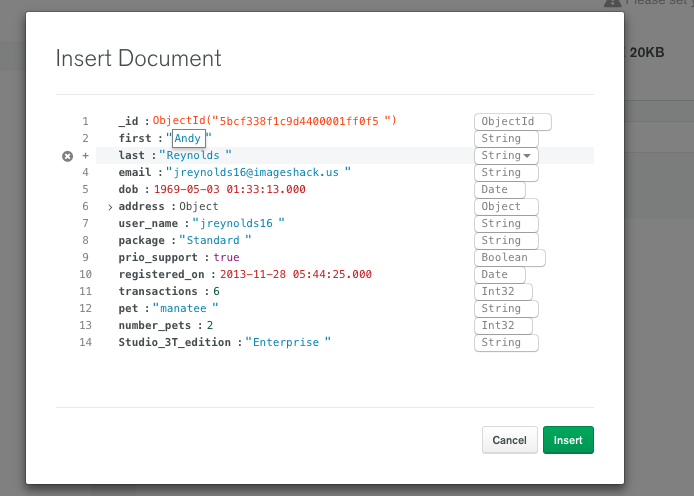
****

1. Insert New document into a collection:

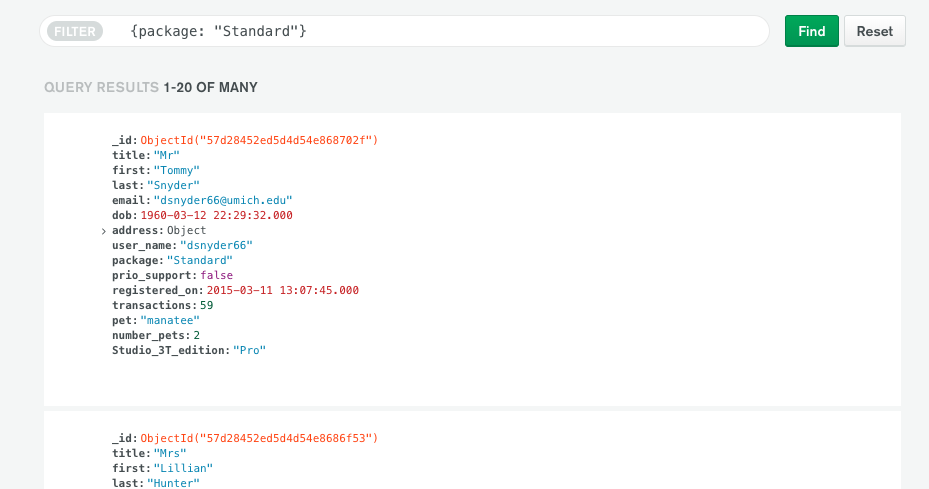
****

1. Duplicate existing document and modify it creating a new document:

****

****

1. Search for all documents with **Standard package:**

****