**OBJECTIVE**

The objective of this lab is to introduce MongoDB, using the MongoDB cloud environment and use MongoDB Compass. Together, we will learn how to:

* Understand document stores
* Set up the MongoDB Compass
* Create documents within MongoDB environment
* Run queries against the documents

**PREREQUISITES**

Before attempting this lab, it is best to read the textbook and lecture material covering the objectives listed above. While this lab shows you how to create and use these constructs, the lab does not fully explain in full the theory behind the constructs, as does the lecture and textbook.

**REQUIRED SOFTWARE**

The examples in this lab will execute in the MongoDB cloud environment as well as Mongo Compass GUI. The screenshots in this lab display execution using Mongo Compass. Alternatively, you are also provided shell scripts.

**LAB COMPLETION**

Use the submission template provided in the assignment inbox to complete this lab.

# Part 1: Introduction to MongoDB

MonoDB is a schemaless database system relying on dynamic schema based on the document structure instead of a statically defined schema such as in relational database designs. We have to understand the structure of the documents before inserting the data into the database to improve the read performance and avoid unnecessary join operations in the application logic.

## **Designing MongoDB document stores**

* The design decisions for the structure of the documents should predominantly be application driven and should be focused towards an access pattern. In some cases, this would compromise the consistency of the data, meaning we are lowering benefits of normalization. However, document stores do allow for linking within the designs, which creates a structure in order to avoid consistency issues.
* A good rule of thumb is that if you find yourself designing the document structure in a manner similar to the relational design, you are probably not making the best use of document stores.
* MongoDB supports relationships such as one to one and one to many through embedding and linking which will be covered in the next section.
* Look to Leverage the nested nature of the BSON (Binary JSON) documents: Focus on embedded designs by organizing information (logically) in a single document. This will avoid **JOIN** operations. JOINs are one of the primary reasons which affect the performance negatively of certain queries within MongdoDB so we want to limit them even in the application logic. This is a trade-off between performance and data consistency and in MongoDB the focus here is on performance.

## **Understanding Relationships**

Let’s have a look at how we would model the different types of relations that we are familiar from the relational model. The two popular techniques for representing any relation are Linking and Embedding.

As an example, let’s model the relationship between a Book and its Author entities. Below A book is represented with the following attributes: ID, Name, Authors and Language.

***Linking Representation***

**Linking** is similar to using primary and foreign keys, note how Authors is a list of IDs which point to the ID of the authors. In this design pattern we represent Books having one or many Authors, one to many relationship. In this design we assume that we would often query books and link authors too them. Observe possible anomalies within the language highlighted in red. As with any design, there is a trade-off between performance and consistency which should be potentially mitigated by the application logic.

**Book**

{ \_id: “10”,

name: “Database Systems ”,

authors: [1,2,3],

language: “ENG”

}

{ \_id: “20”,

name: “Time-Constrained Transaction Management”,

authors: [4,2,1],

language: “english”

}

**Authors**

{ \_id: 1, name: “Avi Silberschatz”}

{ \_id: 2, name: “Henry F. Korth”}

{ \_id: 3, name: “S. Sudarshan”}

{ \_id: 4, name: “N. Soparkar”}

***Embedding* *Representation***

Let’s organize our data in a different way, from a perspective of Authors and **embed** a **collection** of Books. In this example the assumption is that we often query our data by Authors. Observe possible anomalies within both the names of the book and the language highlighted in red.

**Authors**

{ \_id: 1,

name: “Avi Silberschatz”,

books:[ { name: “Database **System** Concepts”, language: “**english**”},

{name: “Operating **System** Concepts”, language: “**english**”}]

}

{ \_id: 2,

name: “Henry F. Korth”,

books: [ { name: “Database **Systems** Concepts”, language: “**ENG**”},

{name: “Time-Constrained Transaction Management”, language: “**ENG**”}]

}

{ \_id: 3,

name: “S. Sudarshan”,

books: [{ name: “Database **System** Concepts”, language: “**english**”}]

}

{ \_id: 4,

Name: “Abraham Silberschatz”,

books: [{ name: “Database **System** Concepts”, language: “**German**”}]

}

In the next section we will discuss best practices around document modelling.

## **Modelling relationships**

**1:Many Relationship**

* Let’s say we want to model the relationship between a College at a University and a Student, both would be tracked through separate documents. Since the cardinality of students within College is high, it is best to use linking. Linking works in a similar way **foreign keys** from the relational design. Either the College document would have a key corresponding to each Student document as a part of its structure or vice-versa, we would not duplicate the relationship. This is similar to how Books and Authors were represented in the Linking example introduced earlier.
* Now let’s try and model the relationship between a Facebook Post and the comments for that post. Observe that a post is not going to have many comments. This type of relationship is **1:Few** where there is low cardinality. For such relationships we embed the entity representing the **Few** as a part of the other **Collection**.

e.g. **Post**

{ \_id: “9876543210”,

name: “Post Name”,

comments: [{\_id: “1”, content: “Great Post!”, user\_id: 1}, {\_id: “2”, content: “Thanks for sharing !”, user\_id: 2}]

}

* As a general guideline use **linking** in case of **true 1:Many** **relationships** and **embedding** in case of **1:Few relationships.** Inspect the cardinality as illustrated above to make the final decision.

**Question 1:**

Within the movie rental business scenario, you are now familiar with, examine the relationships and think about the cardinalities:

**Question 1a**) What would you implement as a 1:Many documents through linking? List two document names and what is the link between them. (short answer).

**Question 1b**) What would you implement as a 1:Few document through embedding List the document name and what would be embedded within the document. (short answer).

**Many:Many Relationship**

* In case of a true **Many:Many** relationship use **linking**. In this example below we will model Books having Many publishers.

**Book**

{ \_id: “10”,

name: “Database System Concepts”,

authors: [1,2,3],

language: “english”

}

**Publisher**

{ \_id: “10”

name: “McGraw Hill”

}

**Published\_Book**

{ book\_id: “10”,

Publisher\_id: “10”

year: “2011”

}

Observe how we have modelled the Published Book relationship between the Book and Publisher collection in the Published Book document. We are linking them by their **\_ids** just like we would do in a bridge table of a relational design.

* Cardinality drives the document’s relationship design. Observe the relationship between Book and Author introduced earlier. Such type of relationships come under the category of **Few: Few**. To model this type of relationship use embedding but preferably only in one direction based on your access pattern, in our example the access pattern is based on authors.

**1:1 Relationship**

* 1:1 relationship can be modelled the same way as few: few with embedding the preferred option depending on the access pattern, with a linking pattern used for bidirectional access.

**Modelling best practices**

* For performance, model in a way that entities accessed together are a part of the same document (this is the embedding example we explored earlier)
* If documents are growing in size, then use linking instead of embedding since the maximum size of each document should not be more than 16 MB.

**Question 1c):** What would you implement as a 1:1 relationship through embedding within the movie rental database? (short answer)

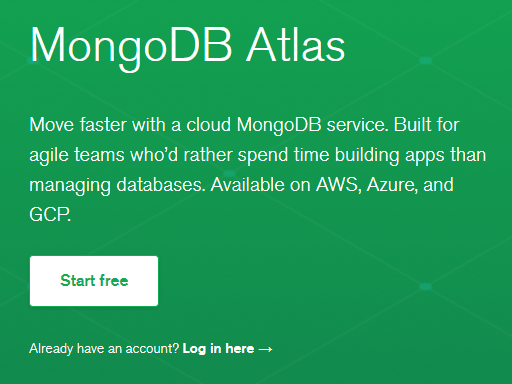
# Part 2: Setting up a MongoDB cloud instance

**Overview:**

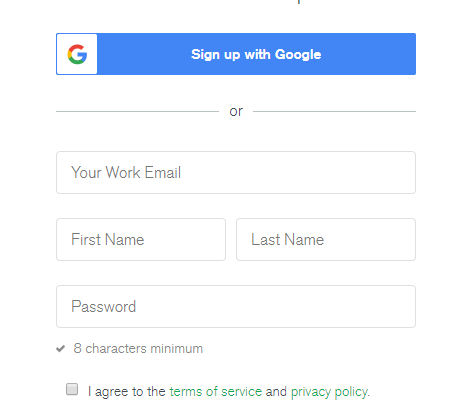
1. Creating a MongoDB cloud instance
2. Installing MongoDB Compass
3. Connecting MongoDB Compass to your cloud instance

**Creating a MongoDB cloud instance**

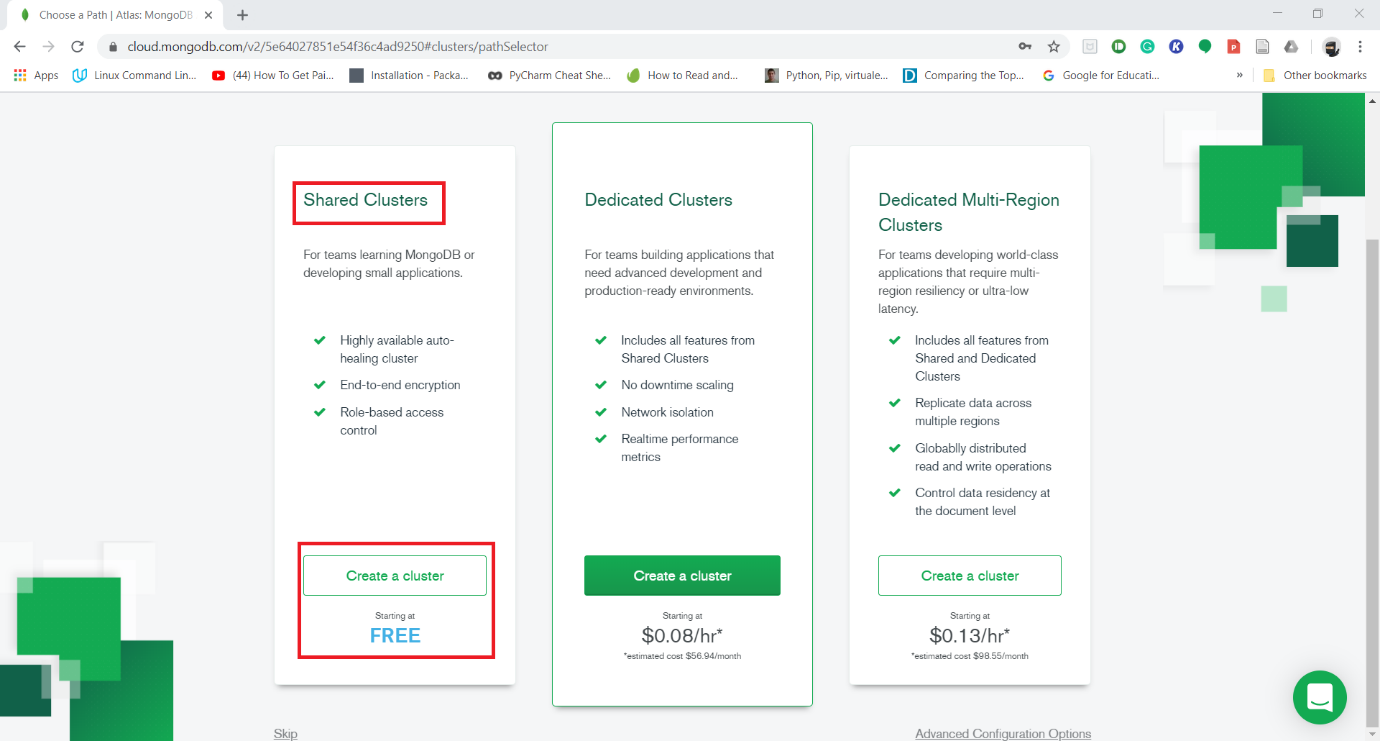
1. Go to <https://www.mongodb.com/cloud/atlas>
2. Select Start Free



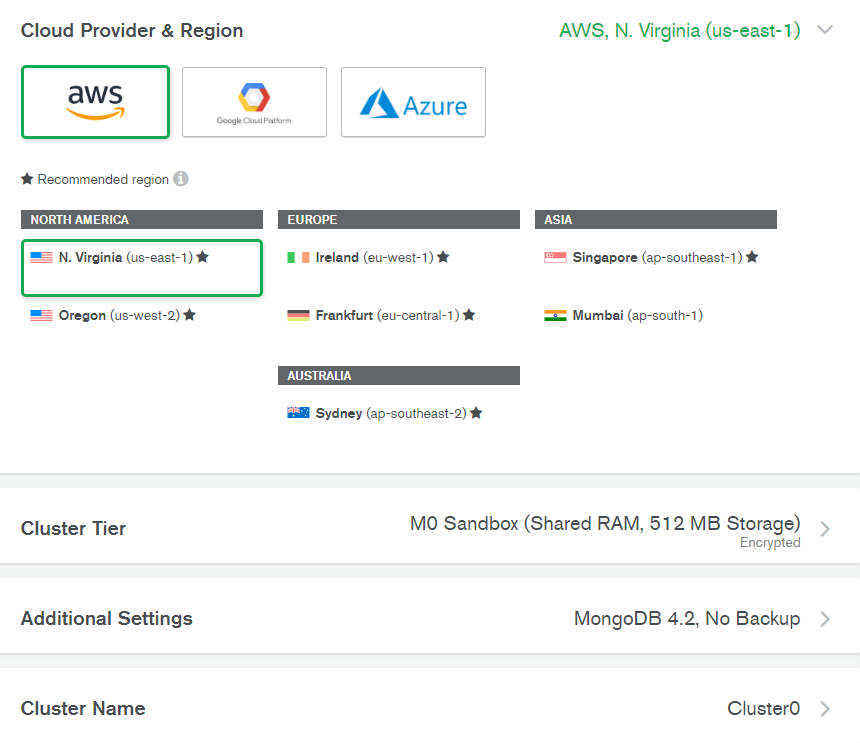
1. Register for an account. Alternatively sign in with your Google Apps BU Gmail account



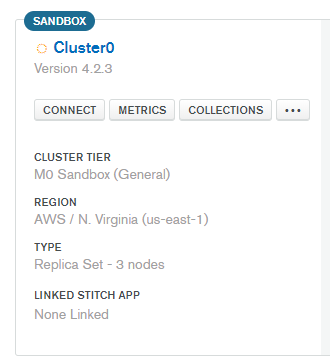
1. Select **Shared Clusters** and click on **Create a cluster**



1. For the purpose of this assignment we are going to deploy our cluster on AWS which is closest to your location. For example, students living on the **East Coast** would select **North Virginia** data center. Default settings are recommended.



1. You will be redirected to your Atlas dashboard as your cluster is deployed which takes about 3 minutes.
2. Once your cluster is ready within your dashboard Click on Connect within Cluster0

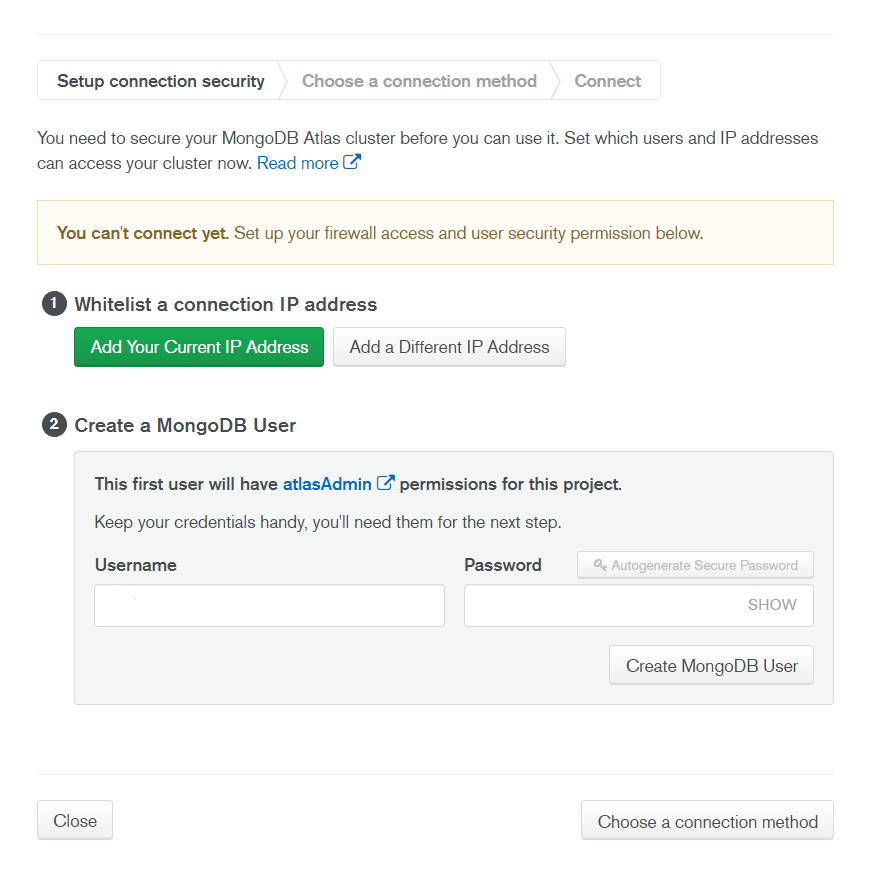


1. Click on **Add Your Current IP Address**

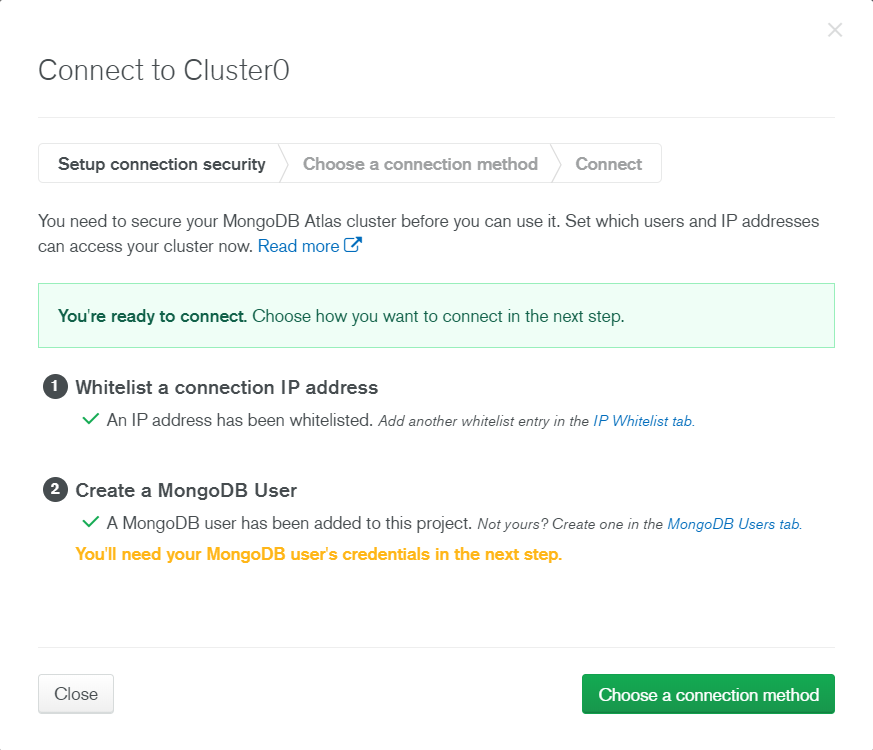
**(If you are going to use a separate machine to connect to this cluster using MongoDB Compass then include the IP address of that machine)**

1. Create a Username and Password for the database user

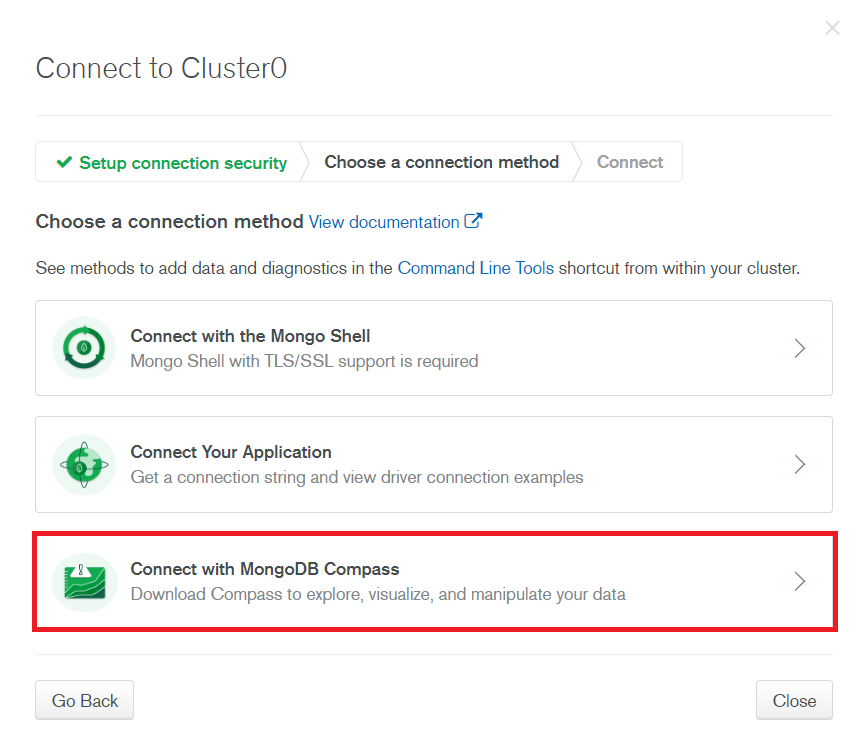
(This username and password corresponds to the database user and is different from your Atlas account)



1. Click on Choose a connection method

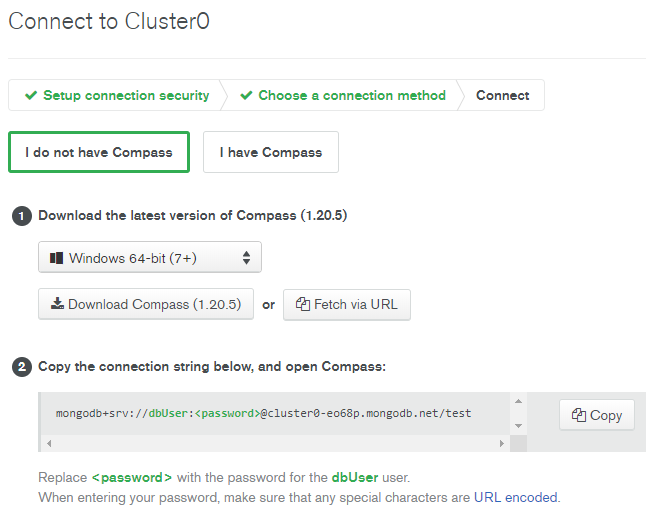


1. For the purpose of this assignment we would be connecting to our MongoDB instance via MongoDB Compass



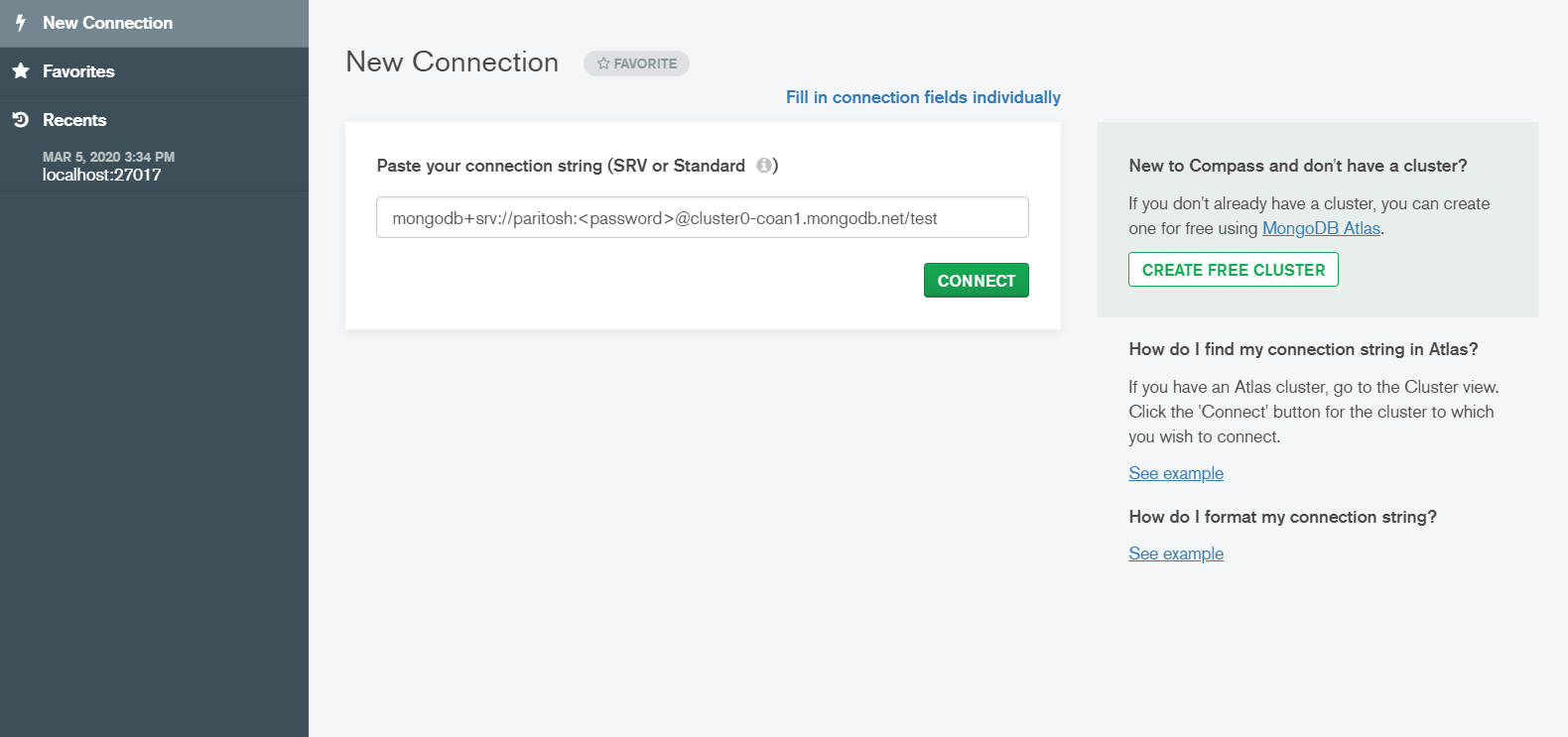
1. Please select your operating system and download the latest version of MongoDB Compass.

Follow the onscreen instructions to install MongoDB Compass.

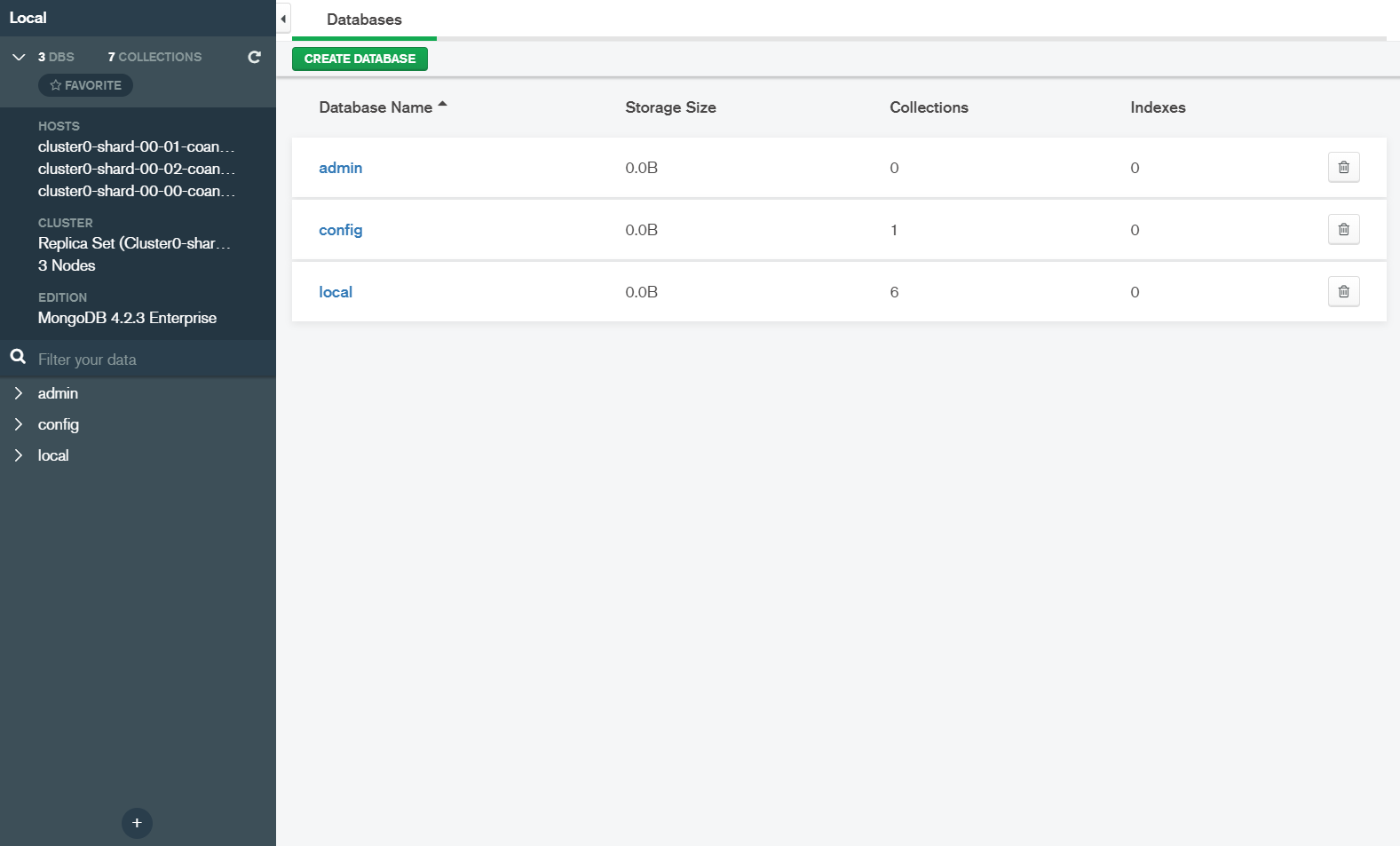


1. After Compass has been successfully installed. Launch Compass and copy-paste the string highlighted in part 2 in the New Connection section of Compass.

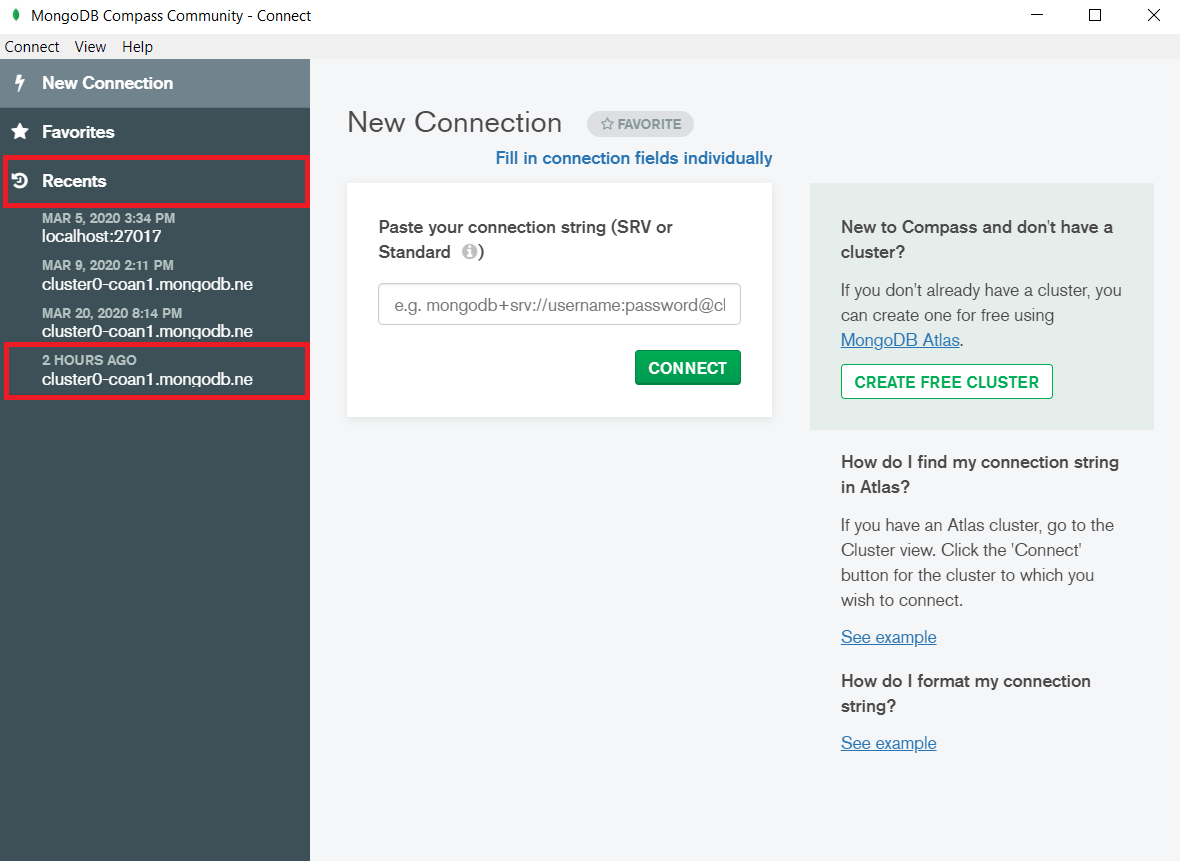
**Important:** Replace <password> with the password which you created for the database user.



1. After a successful connection your Compass window should look like this



1. If you exit Compass and come back at a later time, to reconnect to your MongoDB cluster from Compass you should select the latest connection from the **Recents** section



# Part 3: Getting Started with MongoDB

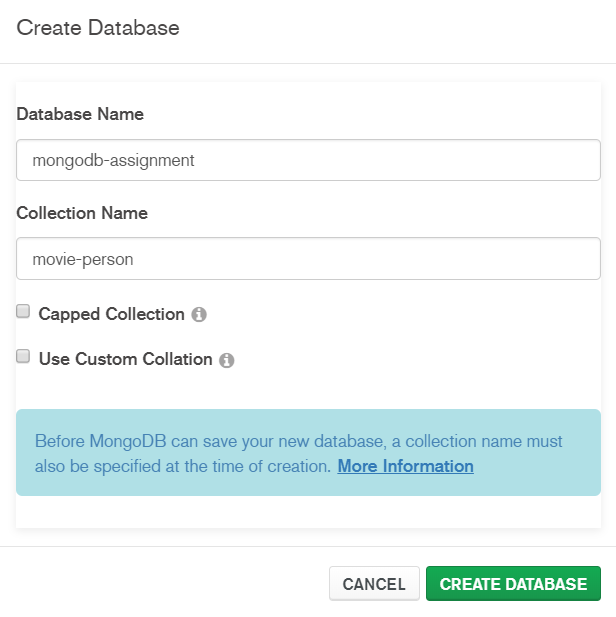
In this section you will find directions through Compass GUI as well as alternative shell commands if you prefer to work with MongoDB through shell.

## **Creating a Database**

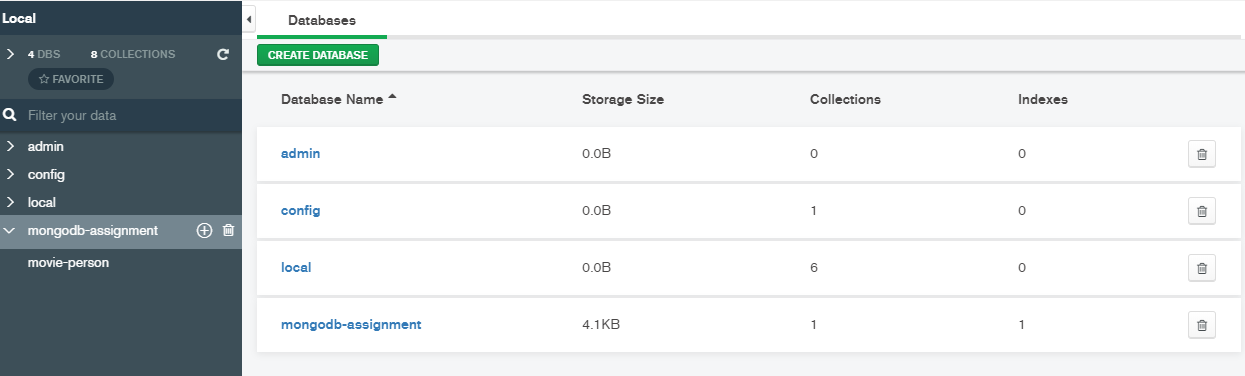
Use the **Create Database** button to create a new database for this assignment.

Name it **mongodb-assignment**

Compass will not allow you to create an empty database so initialize it with the **movie-person** collection.



You will see 3 system databases along with the database that you created.

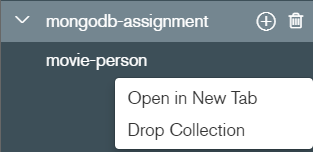
****

**Mongo Shell Alternative to create database**

use mongodb\_assignment

## **Dropping a Database/Collection**

You don’t need to try the following step unless you plan to re-create the database: Within the Atlas GUI on the left hand side, click the bin icon to delete a database. If you want to delete a collection click on the down carrot icon next to the collection’s name and select Drop Collection from the three dot menu.



**Mongo Shell Alternative to drop database or collection**

First Make sure you are using the correct database that needs to be deleted

use <database\_name>

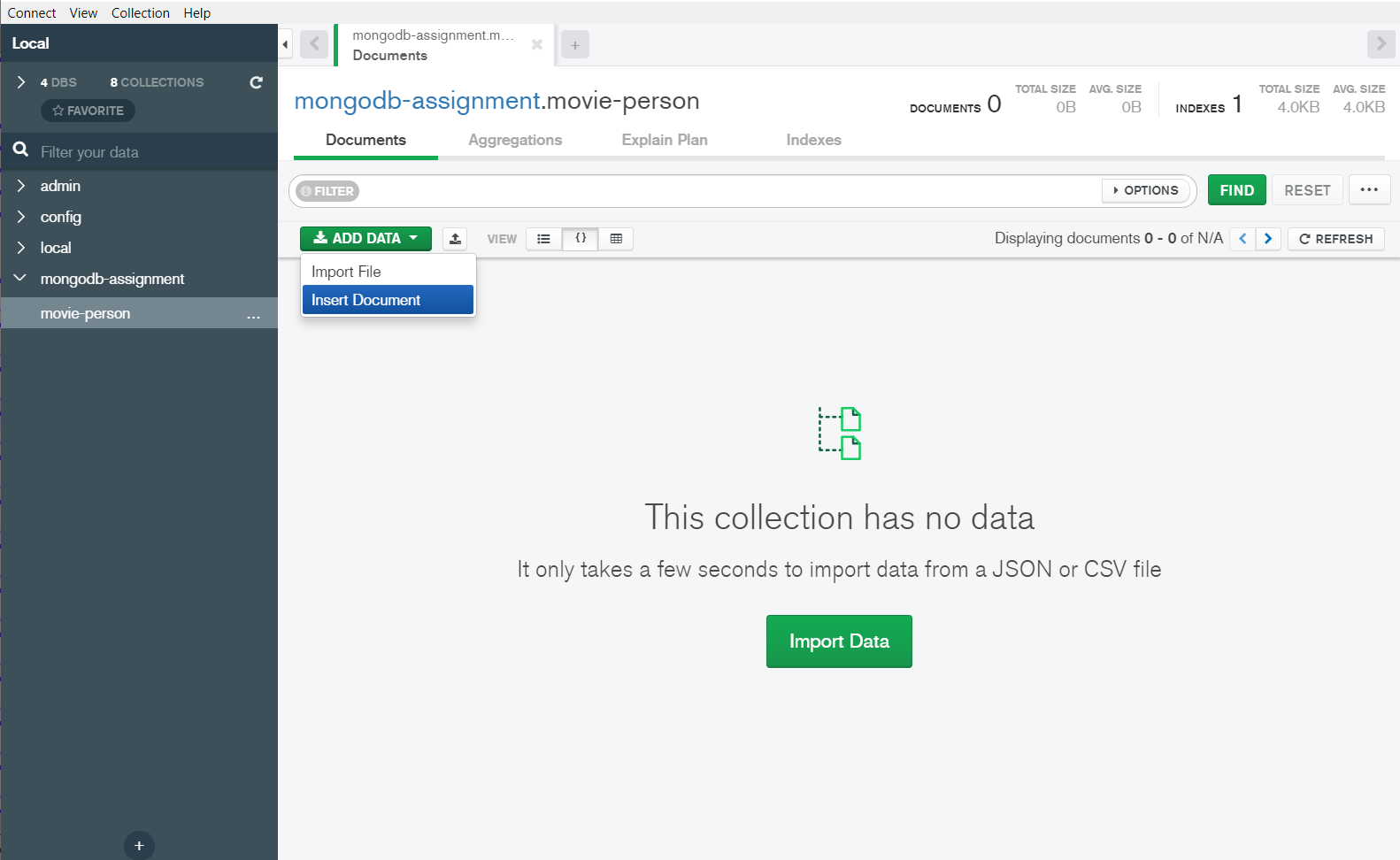
db.dropDatabase()

*To drop a collection from a database:*

db.<collection\_name>.drop()

## **Adding data to a Collection/Inserting documents in a Collection**

Select the **movie-person** collection in which data is to be added. Click **Add Data** and select **Insert Document**.



Position your curses on line 4 and Copy and paste the following JSON document in the dialog box and click **Insert**. Note that alternatively you can import from a JSON or a CSV file.

{

"first\_name": "Bill",

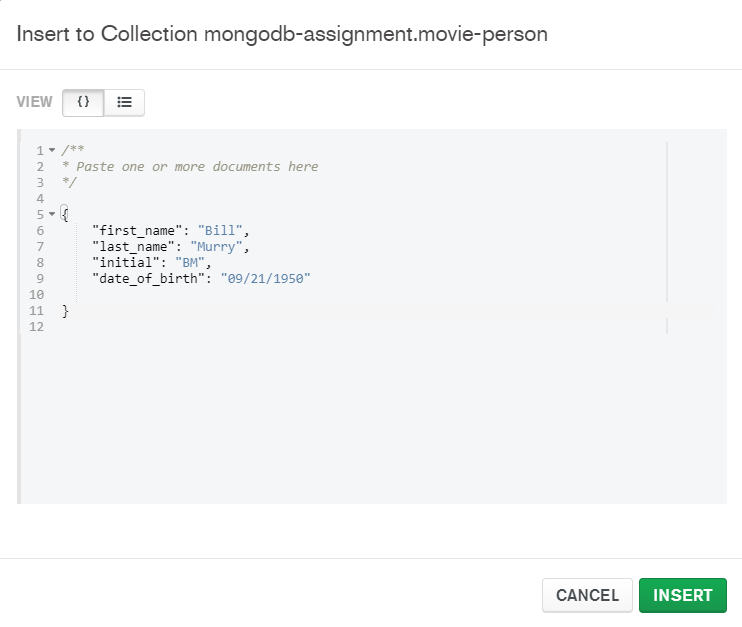
"last\_name": "Murry",

"initial": "BM",

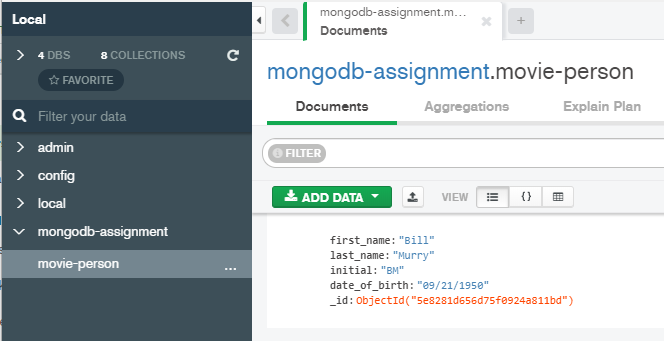
"date\_of\_birth": "09/21/1950"

}

This is what your document should look like before you insert it



You should see a record for Bill Murry as shown below. Notice how an ObjectID was generated for the primary key.



**Mongo Shell Alternative:**

*To insert one Document:*

db.<collection\_name>.insert({JSON})

*To insert multiple Documents:*

db.<collection\_name>.insert ([Array of JSON objects])

**Question 3a):**

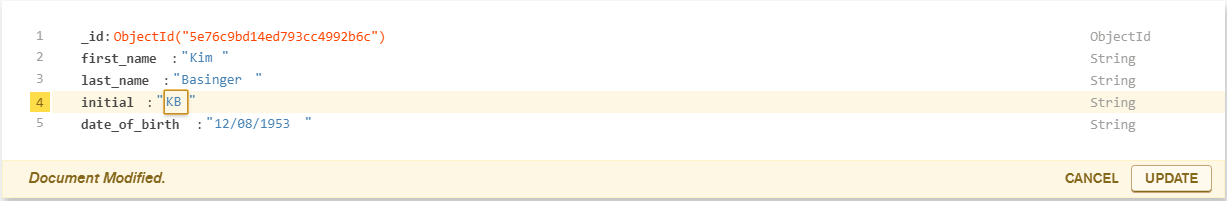
Import the rest of the movie people by using either the copy paste method or importing the JSON document.

Paste screenshot showing both copy and paste or importing the JSON document and the inserted data result. The screenshot of the result should show how many total documents are in the movie-person collection.

## **Updating a Document in a Collection**

MongoDB Compass GUI allows you to graphically update documents. In order to do so through a command line like SQL, you have to use Shell alternative. Below we will show you how to update a document through the GUI. Click the pencil icon next to a document that needs to be updated. Change the initials for Kim Basinger from SC to KB and click update. (Kim Basinger will be the last document)





**Mongo Shell Alternative:**

*To update a single Document based on a condition:*

db.<collection\_name>.update(

{<condn>}, {$set: {key:value}}

)

***Update command changes only the first document that satisfies the condition***

*To update multiple Documents that satisfy the condition:*

db.<collection\_name>.update(

{<condn>}, {$set: {key:value}, {multi: true}}

)

*Other alternative to update a Document*

db.<collection\_name>.save({new JSON with document id})

If document is found, mongo updates it. Otherwise, mongo creates it.

## **Delete a Document**

MongoDB Compass GUI allows you to graphically delete documents. In order to do so through a command line like SQL, you have to use Shell alternative. Below we will show you how to delete a document through the GUI. You won’t need to delete a document here, but to try it, click the trash icon next to a document that needs to be updated. Note if you delete a document from movie-person, you will need to add it back.

*To delete all the documents from the collection:*

db.<collection\_name>.remove()

*To delete all the documents from the collection that satisfies a condition:*

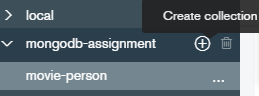
db. <collection\_name>.remove({<condn>})

*To delete the first document from the collection that satisfies a condition*

db. <collection\_name>.remove({<condn>}, 1) – will remove the first document that satisfies the condition <condn>

## **Populating the schema**

**Next** lets populate the **dvd** and **member** collections using the files **dvd-insert.json** and **member-insert.json** respectively. To add new collections, next to the database name click on the plus sign to create a new collection.



You will need to create a dvd and member documents and populate them.

**Question 6a):** Provide screen-shots of both dvd and member collections loaded with the documents from the provided JSON files, make sure to show the record counts.

**Question 6b):** Examine the DVD document and specify how the Genre and Rating relationship was implemented, was it through embed or linking? What is the cardinality of these relationships and does it correctly support the design choice? (short answer)

**Question 6c**): How are Member, DVD and rental document related? (through embdding or linking, explain your answer brifly)

We will now need to create a rental document and populate it. Find the **rental-insert.json** file. You will notice the document has the following structure.

{

"member\_id": {

"$oid": ""

},

"dvd\_id": {

"$oid": ""

},

"rental\_request\_date": "",

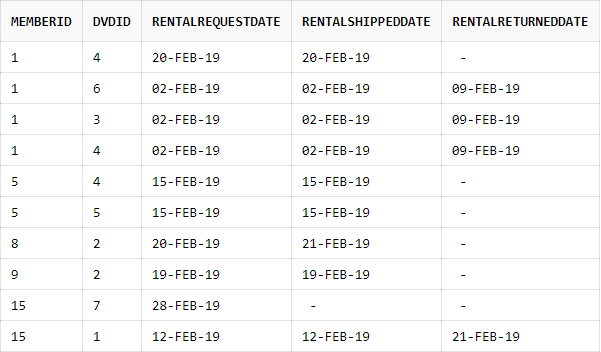
"rental\_shipped\_date": "",

"rental\_returned\_date": ""

}

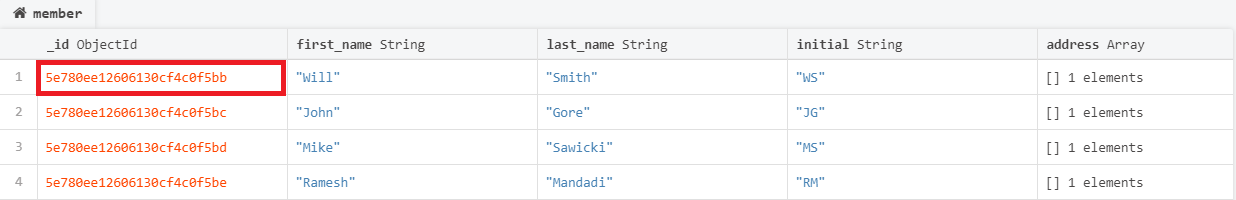
**Note:** $oid is used to indicate that this field has a data type of ObjectId

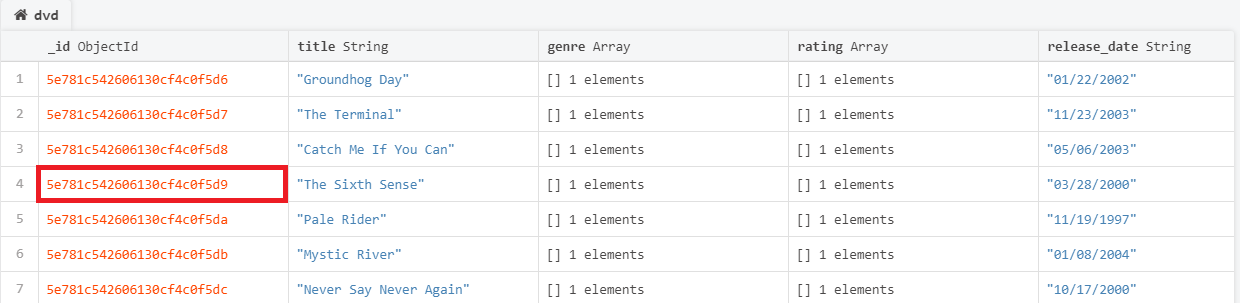
Notice that $oid in MemberID and DVDID is not provided in the file. Below screen-shot shows you how the foreign keys map, you will need to substitute the $oids in the rental-insert with the appropriate $oids from your instance.



For the **member\_id** and **dvd\_id** fields you have to use the **\_id** field from the **member** and **dvd** collection respectively.

e.g. To insert the first record you would need the **\_id** field corresponding to the 1st member and 4th dvd from the member and dvd collection respectively. **Hint**: use the **grid view** to explore the collections, it will be easier to copy and paste the oids.





After using the \_id’s from the respective collections and filling in the appropriate dates the first document for the rental collection would look similar to the following

**Question 6d):** Provide a screenshot of the populated rental collection

**Question 6e):** Examine the dvd collection and movie-person collection. We have not established a relationship between them. Briefly describe how would you establish the relationship i.e. via linking or embedding and what kind of a relationship is it.

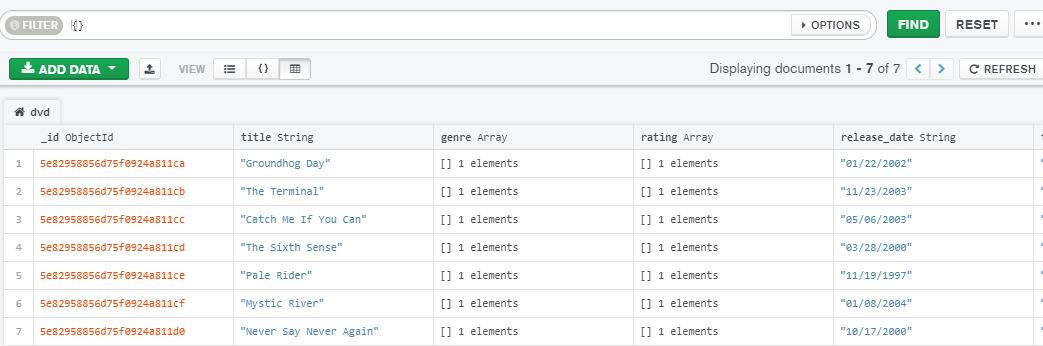
**Question 6f - extra credit):** Create a linked collection for DVD and Movie-Person called movie-person-dvd and populate it.

## **Querying and filtering**

The Query Bar specifies the various operations that can be performed on a collection.

Let’s take a look at **Filter**

To display all the data that is present in the collection enter {} in the **Filter** section of the query bar and click Find.*Note: This is similar to SELECT \* FROM table-name*

****

**Mongo Shell Alternative:**

use databasename

db.<collection\_name>.find()

*To display the data in a presentable/readable format:*

db.<collection\_name>.find().pretty()

*To fetch only the first result:*

db.<collection\_name>.findOne()

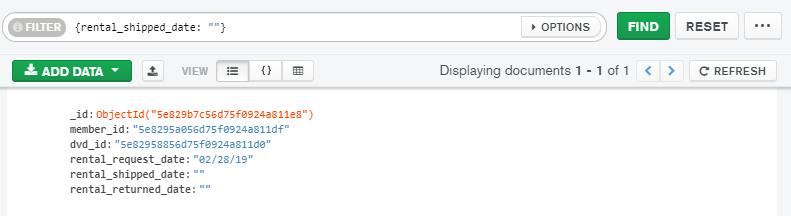
**Querying the data present in the collections based on a condition**

To filter the documents present in the collection enter {key: value} in the **Filter** section of the query bar and click Find.

We want the rental documents for the DVD’s that have been requested but have not yet been shipped.

Go to the Rental collection and Enter {rental\_shipped\_date: ""} in the query bar and click Find.

You should get the following result:



**Question 7a):** Find all the rental documents for DVDs that have not been returned. Take a screenshot of the result.

**Projection operation**

Projection operation is used to display only the necessary fields instead of all the fields from the documents of a collection.

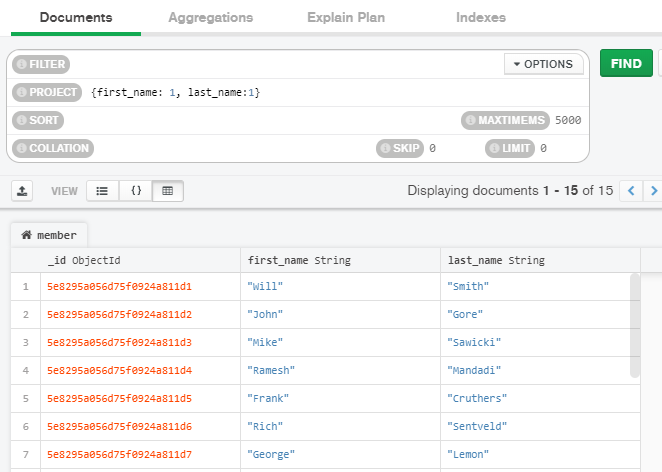
We want only the first and last name field from all the documents of the member collection.

Go to the member collection and in the filter bar click on **Options**

Enter {first\_name: 1, last\_name:1} in the **Project** section of the query bar and click Find.

1 - True, 0 - False

**Note:** \_id field will appear by default unless specified otherwise



**Mongo Shell Alternative:**

db.<collection\_name>.find({}, {key:1})

1 - True, 0 - False

**Note:** \_id field will appear by default unless specified otherwise

**Question 7b):** Find the first name, last name and email of all the members. Please make sure that the result does not have the \_id field. Take a screenshot of project command the result.

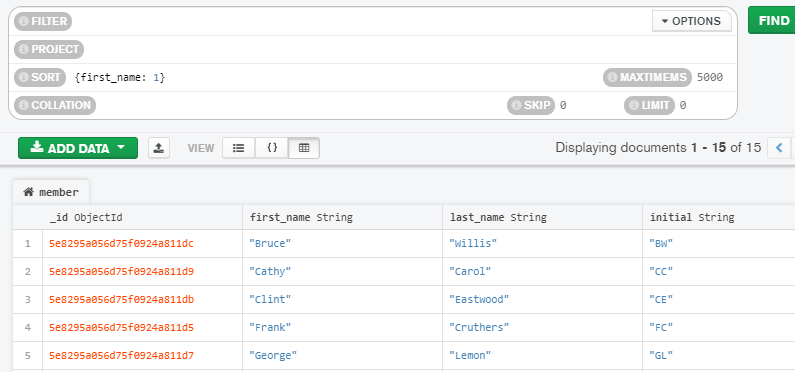
**Sorting the result**

We can sort the documents based on the values for a specific attribute.

To sort all the documents from the member collection on the basis of the first name in ascending order.

Enter {first\_name: 1} in the **Sort** section of the query bar and click Find.

-1: for sorting in descending order, 1: for sorting in ascending order

****

**Mongo Shell Alternative:**

db.<collection\_name>.find().sort({<key\_for\_sorting>: 1})

-1: for sorting in descending order, 1: for sorting in ascending order

**Question 7c):** Sort the documents in the member collection based on the last name of the members in descending order. Take a screenshot of the sort command and the result.

**Limit the result**

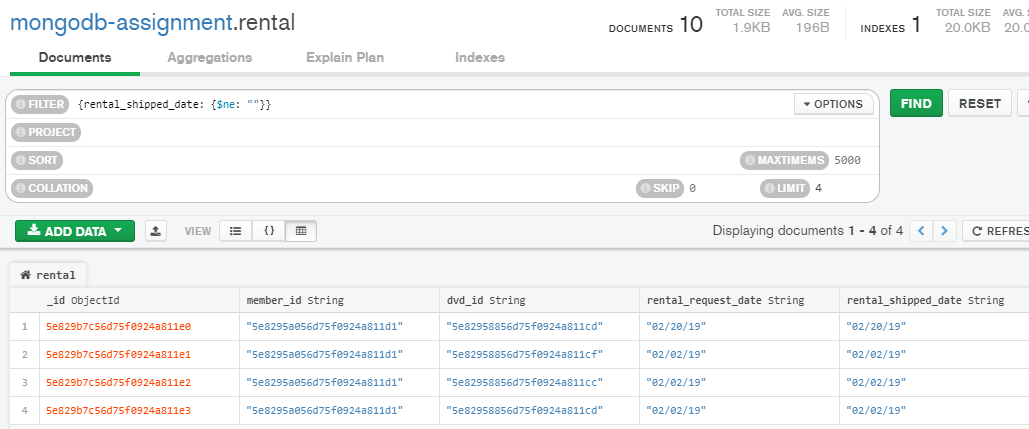
We can limit the number of documents in the result.

To find 4 documents of the rental collection where the DVD has been shipped.

Enter {rental\_shipped\_date: {$ne: ""}} and 4 in the **Filter** and **Limit** section of the query bar respectively and click Find.

**Note:** $ne is the not equal operator.

You can read more about operators here: <https://docs.mongodb.com/manual/reference/operator/query/>



**Mongo Shell Alternative:**

db.<collection\_name>.find().limit(<number>)

**Question 7d)** Find 4 documents of the rental collection where the DVD is not returned by the member. Take a screenshot of the result and attach it to the final submission.

## **Joining Collections**

**Concept of Pipelining**

MongoDB supports the pipelining of commands in the aggregation framework. Pipelining enables us to execute a set of commands in a sequential fashion where the output of one command serves as the input for the next command. There are various stages associated with the pipeline, each of which is capable of performing specific operations on the set of **Documents** before moving onto the next stage of the pipeline.

For more information on Aggregation Pipeline Stages, please refer <https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/>

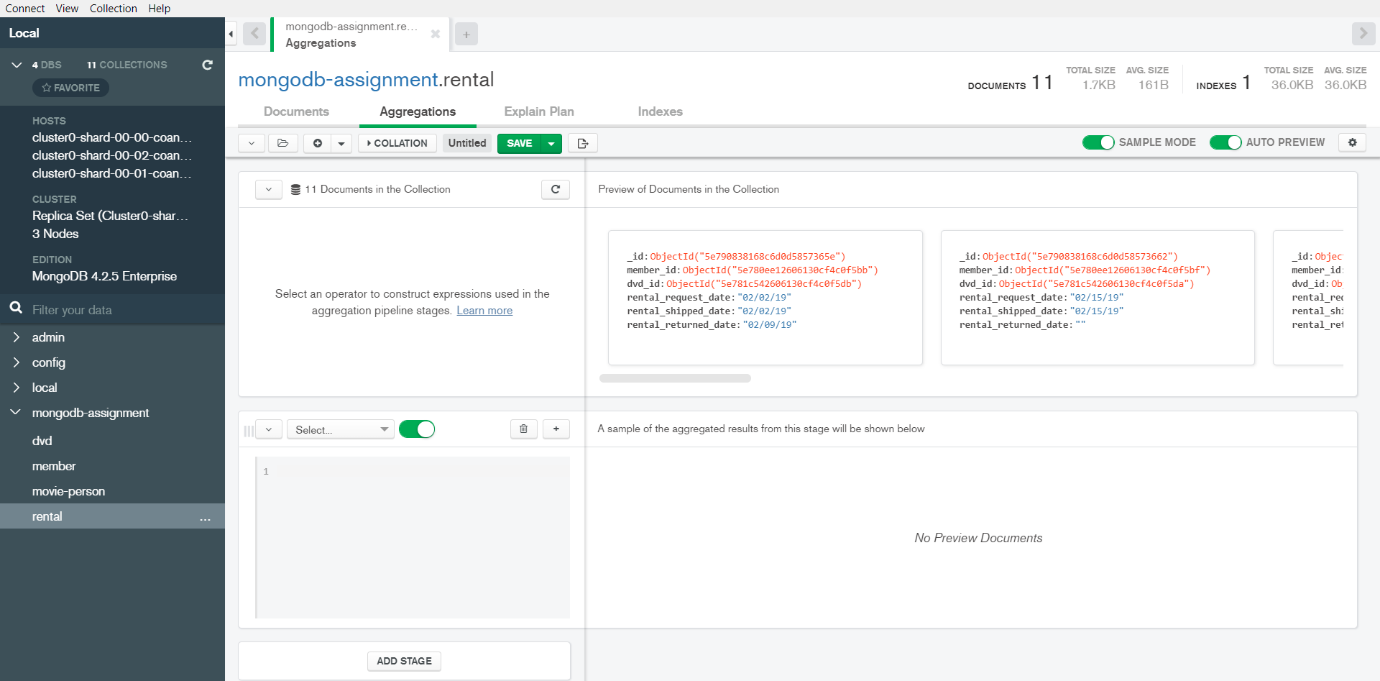
The **$lookup** stage is an important one since we can perform a **left outer join** between two collections of the same database. Both of these collections have to be **unsharded**.

Finally, to persist/write the documents returned by the aggregation pipeline we use the **$out** operator. It should always the last stage of the pipeline.

Below we will demonstrate how to perform a left outer join between the **rental** and **dvd** collection on the basis of the key **dvd\_id**

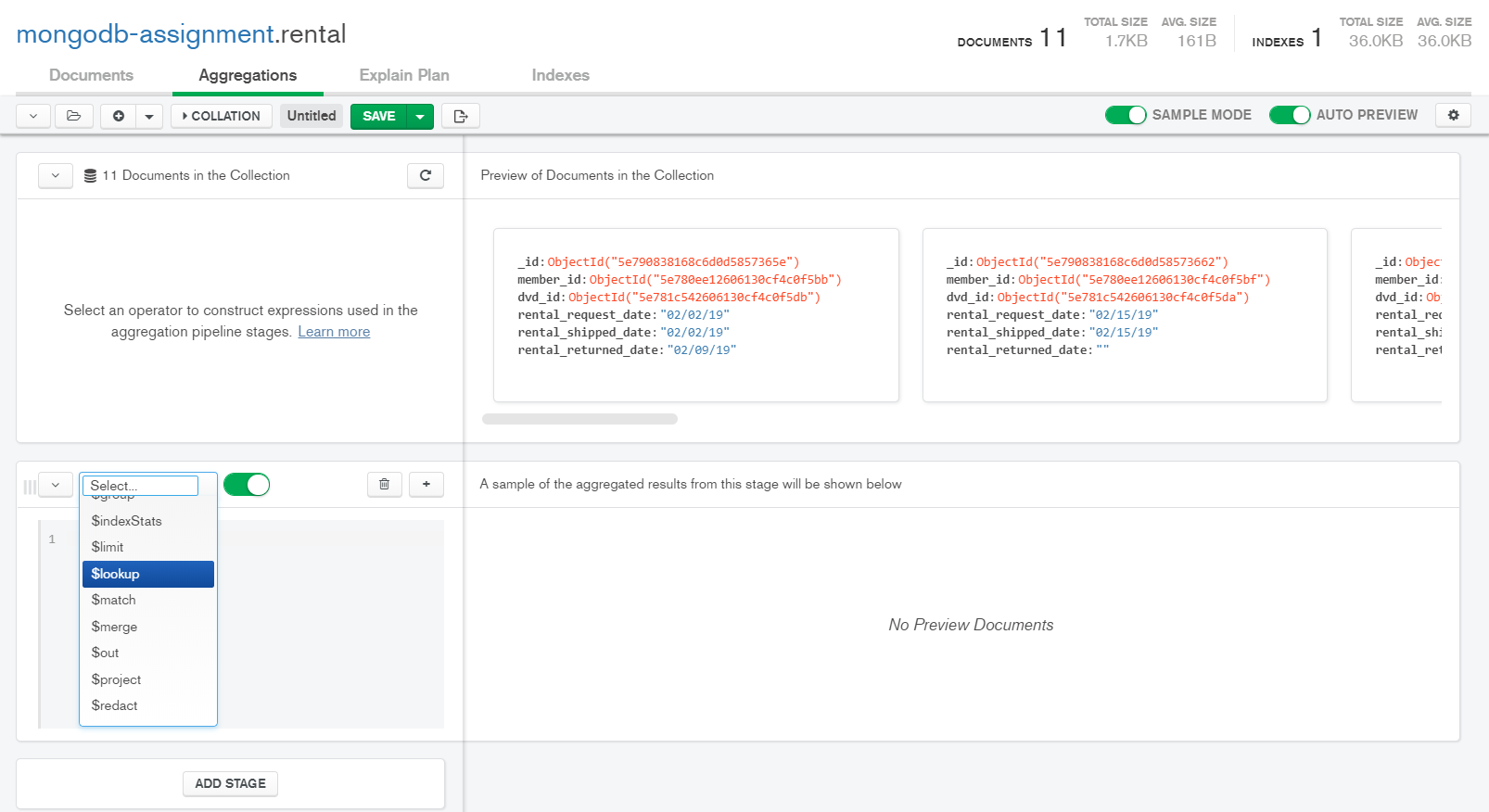
**Creating an Aggregation Pipeline**

1. Select the **rental** collection and head over to the **Aggregation Pane** at the top.

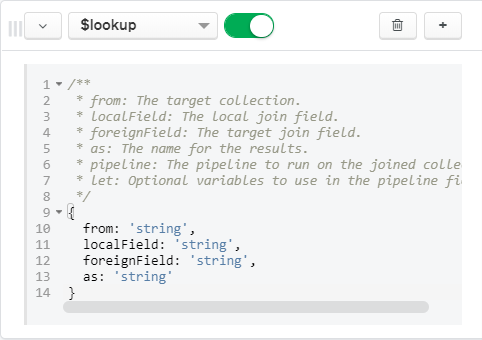


1. Add an aggregation pipeline stage

Select **$lookup** from the drop down in the bottom-left view of the pane. This will ensure that the first stage in the pipeline is $lookup.



The pipeline stage should auto-populate like the following



**from:** Name of the collection from the **same** database to perform the join with.

**localField:** The field from the documents input to the $lookup stage.

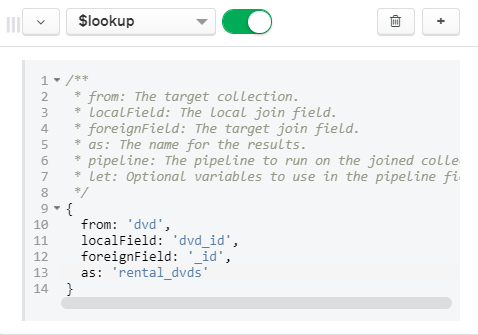
$lookup performs an equality match on the localField to the foreignField from the documents of the from collection. If an input document does not contain the localField, the $lookup treats the field as having a value of null for matching purposes.

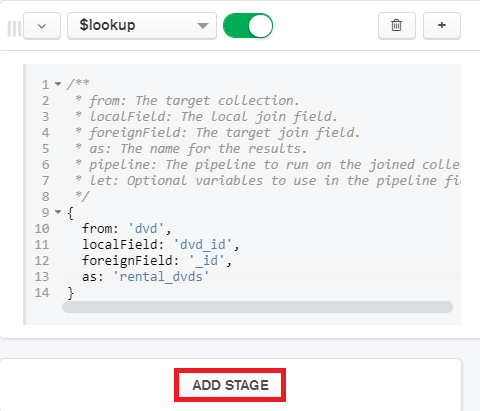
**foreignField:** The field from the documents in the **from** collection.

$lookup performs an equality match on the foreignField to the localField from the input documents. If a document in the **from** collection does not contain the foreignField, the $lookup treats the value as null for matching purposes.

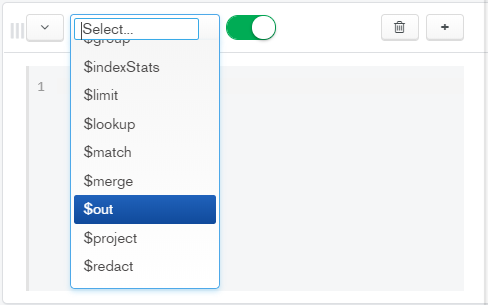
**as:** Specifies the name of the new array field to add to the input documents. The new array field contains the matching documents from the **from** collection. If the specified name already exists in the input document, the existing field is overwritten.

**Reference:** <https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/>

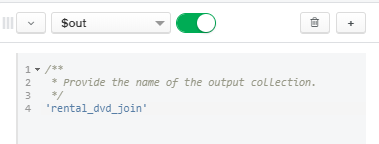
1. Fill in the pipeline stage as follows
2. Add the **$out** stage to persist the result in a separate Collection

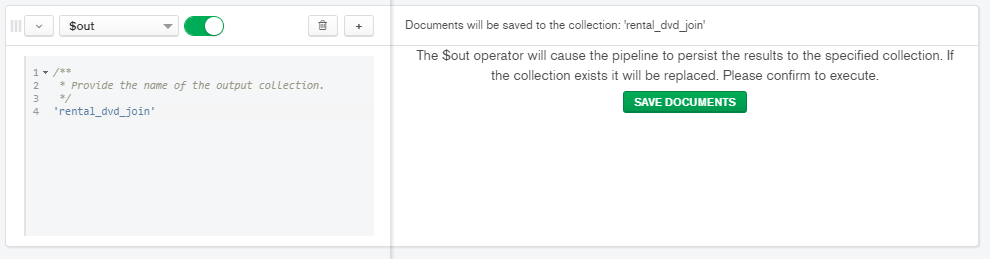
Click on Add Stage in the bottom-left view of the pane.

Select **$out** from the drop down

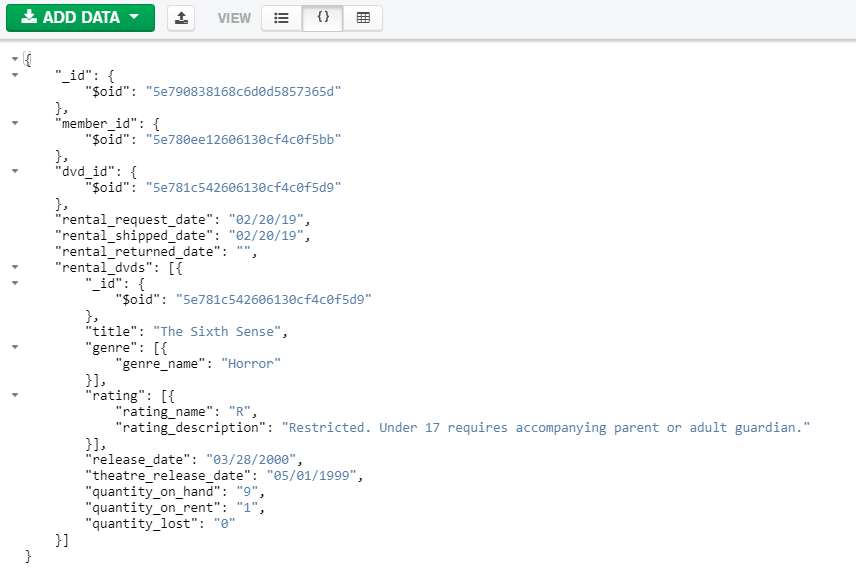


Name the collection as **rental\_dvd\_join** and click **SAVE DOCUMENTS**

****

****

Click on the newly formed **rental\_dvd\_join** collection and examine the JSON structure of its first document.



Observe the title, genre and rating of the dvd are now embedded. We are able to access these fields since we had stored a link to a dvd in each document of the rental collection.

**Question 8a):** Perform a left outer join between the rental and member collection and name the resulting collection as **rental\_member\_join**. Take a screenshot of the JSON structure of the last document (i.e. 11th document) from the rental\_member\_join collection. Please expand all the fields so that the values are visible clearly.

**Extra credit questions:**

**Question 8b):** We want to access the dvd and member data together for each rental document. Which collections would you join to achieve this? Perform the join and take a screenshot of the resulting collection.

**Question 8c):** Examine the resulting collection from Question 9. Briefly describe the problem with this collection.