

# Querying NoSQL Document Databases Using MongoDB Compass

## Lab Guide

### 0. Chapter-to-lab Mapping

*Databases Illuminated* (4e), Chapter 13.3, 14.2, 14.4

### 1. Introduction

As a database professional, you may be responsible for managing large-scale data sets with performance needs that exceed the capabilities of typical relational database management systems. In addition, rapid application development often relies on flexible data storage where a rigid, structured schema with integrity enforcement is not required. NoSQL systems were designed to address these concerns regarding efficient data distribution and access, as well as flexible and dynamic database creation. NoSQL systems, or non-relational systems, can be characterized by the type of data they store. NoSQL systems are capable of handling structured, semi-structured, and unstructured data. In this lab, you will focus on using a document database called MongoDB.

MongoDB was initially created to be the database system for a platform-as-a-service (PaaS) architecture using only open source software components. PaaS is a cloud computing service that allows users to create applications on a computing platform without being required to provision or maintain the platform. The name MongoDB is derived from the word “humongous.” The first version of MongoDB was made available in 2009, and it is estimated that it has been downloaded more than 210 million times. MongoDB is licensed as a source-available, distributed, document database and a community edition can be freely downloaded for multiple operating systems. It is intended for large-scale, database-driven application development. MongoDB seamlessly supports balancing data and query loads by redistributing documents and routing queries without a need to change application logic. Ad hoc querying and indexing are supported by MongoDB.

In a document database, the data associated with an individual record is modeled as a document. MongoDB documents are represented in JSON (JavaScript Object Notation). Documents are similar to rows in a relational database, but they provide more flexibility. Each document can share a common structure but can also vary in the types of fields they contain. This approach allows new information to be added to some documents without requiring that all documents have the same structure.

MongoDB Compass is an interface that connects to a MongoDB instance to import, add, remove, and query data. Querying can take the form of filtering, projecting, and sorting documents; it can also take the form of a sequence of complex manipulations called an aggregation pipeline. Compass is a visual interface that shows a preview of the results of each stage in the aggregation pipeline; it assists a user in interactively constructing and testing a

query but it does not fully execute an aggregation pipeline (this must be done via the MongoDB shell or embedded in a program.) Compass includes other features to profile a collection and validate data using rules. The profiling tool creates a visualization of the keys that are present, their frequency in the collection, their types, and the ranges of values for the keys.

In this lab, you will assume the role of a database professional who retrieves and manipulates documents using MongoDB Compass. Your responsibilities are to write and execute queries expressed as aggregate pipelines in Compass.

## **Lab Overview**

This lab has four parts, which should be completed in the order specified.

1. In the first part of the lab, you will set up and browse a MongoDB database using Compass.
2. In the second part of the lab, you will write and execute aggregate pipeline stages for matching, counting, projecting, and sorting documents.
3. In the third part of the lab, you will write and execute grouping and join queries using the lookup stage of the aggregate pipeline.
4. In the fourth part of the lab, you will create an aggregation pipeline to satisfy a data demand.

## **Learning Objectives**

Upon completing this lab, you will be able to

1. Define the structure of a MongoDB document.
2. Construct an aggregation pipeline in Compass for matching, counting, projecting, sorting, grouping, and joining documents, as well as creating new documents from array elements and saving collections as views.
3. Execute an aggregation pipeline over a given MongoDB database.
4. Develop a solution for a user's data demand using Compass and a MongoDB database.

## Tools and Software

The following software and/or utilities are required to complete this lab. Students are encouraged to explore the Internet to learn more about the products and tools used in this lab.

- MongoDB Compass

The following data files are provided for use with MongoDB Compass.

- restaurants.json
- nyBnB.json
- nyTheaters.json

## Deliverables

Upon completion of this lab, you are required to provide the following deliverables to your instructor:

## Hands-On Demonstration

1. Lab Report file including screen captures of the following:
  - Part 1: 2 database profiling examples using Compass.
  - Part 2: 3 aggregate pipeline stages using match, count, project, and sort operators.
  - Part 3: 4 aggregate pipeline stages using group and join operators.
  - Part 4: 3 aggregate pipeline stages to answer a data demand.

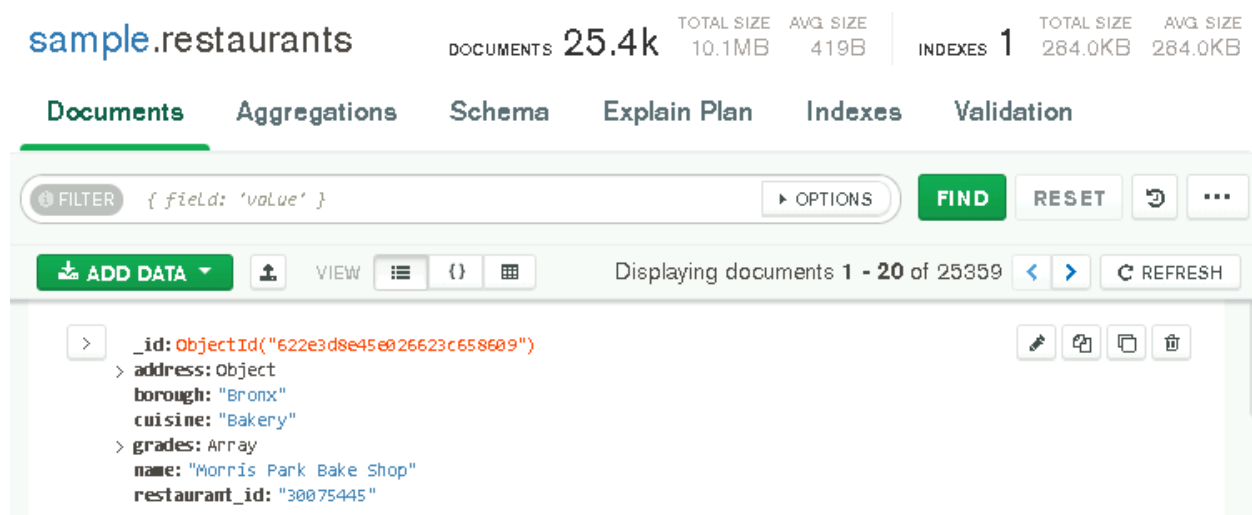
## Part 1: Create a MongoDB database

**Note:** In this part of the lab, you will become familiar with the Compass interface and use it to create and profile a MongoDB database.

Compass provides an environment where you can practice constructing aggregate pipelines and preview results of your queries. Familiarity with the aggregate pipeline stages is important for writing shell queries and MongoDB queries embedded in programs.

The lab assumes that you have already created the restaurant collection in your sample database. In the remaining steps in this part of the lab, you will use Compass features to familiarize yourself with the data format used in MongoDB as well as the specific contents of the restaurant collection. Click on restaurants to make it the active collection.

The documents in the collection can be displayed in three different ways. The default view of the data is represented by an icon that looks like a bulleted list next to the word VIEW. In this view, all of the first level keys and values are shown in a column, one key-value pair per line. The keys are shown in boldface, while the values are shown to the right of the colon (:). Each object has a unique `_id` value for the collection. Values can be strings (shown in double quotes), numerics (no quotes), objects, or arrays. If the value is an object or an array, then there is a nested object or array, respectively, inside the first level object. The keys and values in the object (or array) can be displayed by expanding the object (or array) using the arrow (>) to the left of the key.



1. Click on the > symbol next to address.
2. Click on the > symbol next to coord.

**Note:** Observe the keys that comprise the address object: building, coord, street, and zipcode. These are second level keys in the original restaurant object. Since coord is an array, its

elements are a third level of nesting in the original object. If you expand grades, you will see that the elements of the array are objects, and each object has three keys: date, grade, and score.

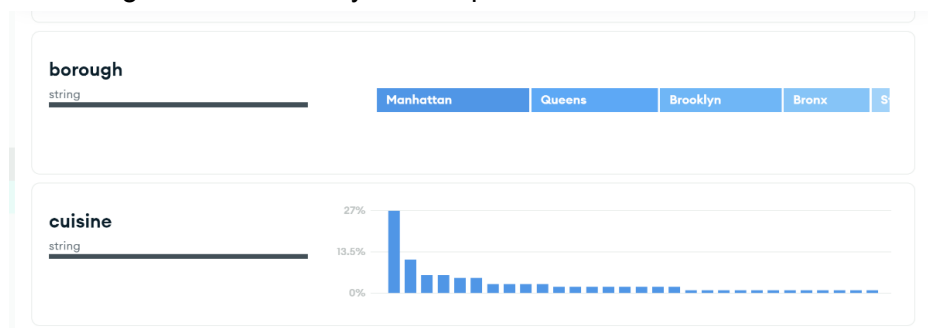
The second viewing option denoted by { } next to the VIEW label shows the documents in JSON format, while the third option that looks like a table shows the data in a tabular format. You will use the default option for viewing documents in this lab.

To find out more about the contents and values in the database, choose the Schema tab in the menu about the FILTER bar.

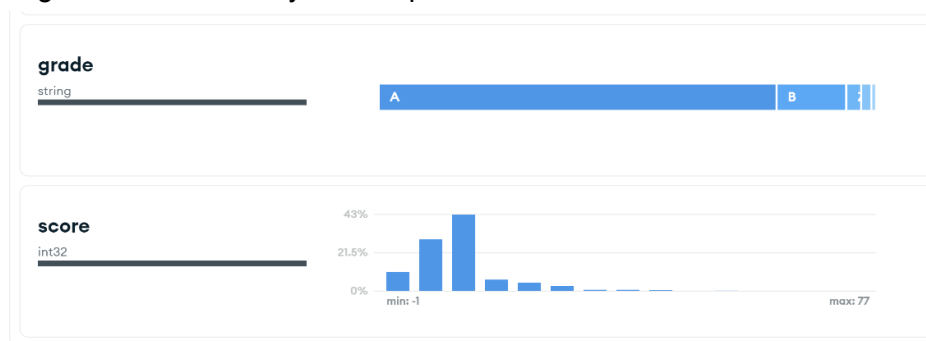
3. **Click** on **Schema** in the tool bar below sample.restaurants and above the FILTER bar.
4. **Click** on the **Analyze Schema** button in the middle of the viewing panel.

**Note:** Based on a sample of 1,000 documents, you can explore each key in the collection and the most frequent values for each in the sample. The answers may be different each time you execute the Analyze Schema command since it only computes values for a sample and the sample may be different each time.

5. Use the scroll bar on the right to **scroll down** to the **cuisine** key.
6. **Hover** your **mouse over the bar labeled string** under cuisine to see the percentage of values for cuisine that have the type string.  
String(100%)
7. To the right, **hover your mouse over the first column in the bar graph** showing the most frequent values in the sample.  
American 27%
8. **Make a screen capture** that shows the bar graphs for type and distribution of values for borough and cuisine in your sample.



9. **Click** on the **arrow** before grades to expand the array.
10. **Hover** your **mouse over the bar graph for dates** (labeled S, M, T, W, T, F, S) to see the frequencies that each value occurs in the date key.  
S: 17%  
M: 21%  
T: 21%  
W: 23%  
T: 15%  
F: 3%  
S: 0.1%
11. **Hover** your **mouse over the bar graph for grade** to see the most frequent value occurs in the grade key and its frequency.  
Most Frequent Value: A  
Frequency: 80%
12. **Make a screen capture** that shows the bar graphs for type and distribution of values for grade and score in your sample.



**Note:** In practice, profiling an unknown database is an activity undertaken frequently by developers and database users to become aware of the structure and contents of an existing database that they will be working with.

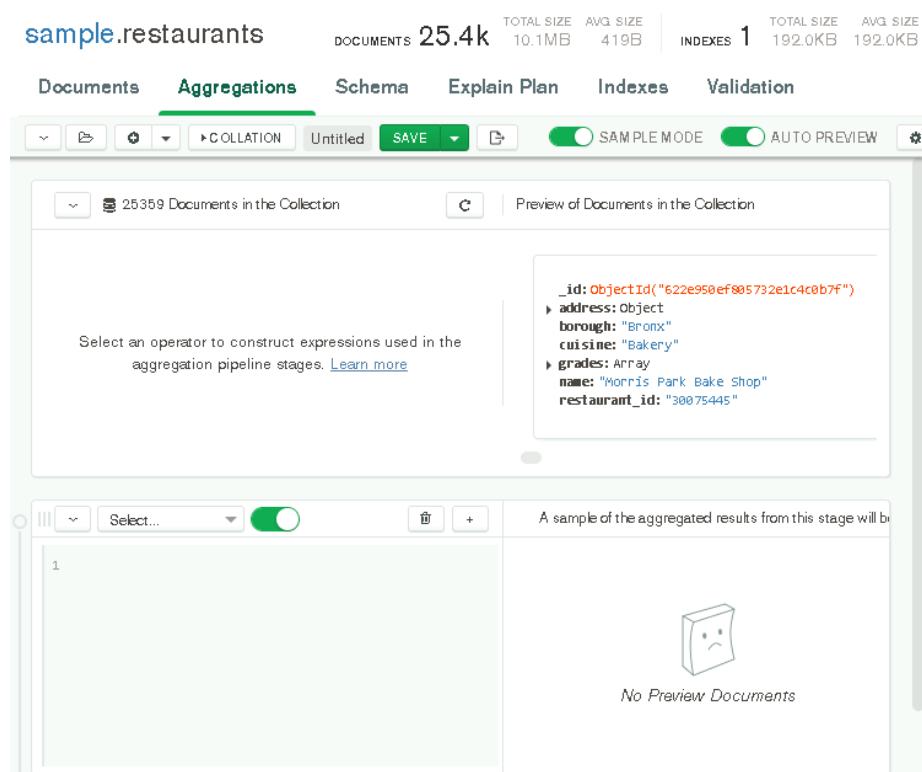
## Part 2: Create an aggregation pipeline

**Note:** In this part of the lab, you will manipulate the documents in a single collection using operators to filter documents based on conditions, project keys and values, sort documents, and group documents based on key values.

Queries in MongoDB Compass are expressed as a sequence of operations. The sequence is known as an aggregation pipeline, and each operation is a stage in the pipeline. The current collection is the input to the first stage in the pipeline, and the output from executing each stage is the input to the next stage. To create a query, choose the Aggregations tab.

1. Click on the **Aggregations** tab in the menu under sample.restaurants and above the FILTER bar.

**Note:** A preview of the entire collection is shown as the beginning of the pipeline. The first stage in the pipeline is created when an operation is selected using the down arrow next to Select... When the operation is executed, a preview of its results will appear next to it on the right. A new stage can be added in the same way below this stage.

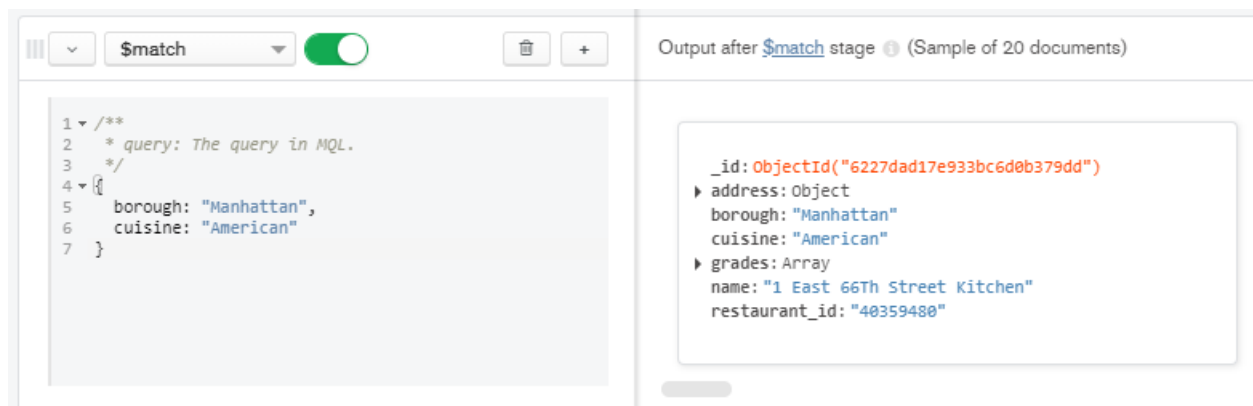


A data demand is the description of what data a user or client would like to retrieve from the database. A data demand is expressed in natural language and must be translated into executable syntax by a knowledgeable database professional. Data demands and MongoDB aggregation stages (operations) to satisfy the data demands are illustrated in the lab.

To find all the documents in the collection where the borough is Manhattan and the cuisine is American, use the \$match stage.

2. **Click** on the **down arrow** to the right of Select... to see a dropdown menu of operations.
3. **Scroll** down the menu options and **choose \$match**.
4. **Type** over the highlighted word **query** with  
**borough: "Manhattan",**  
**cuisine: "American"**

**Note:** The preview is generated as soon as a syntactically correct query is typed. The following screen capture shows the \$match stage on the left and a preview of the result on the right.



One of the tricky things about writing MongoDB queries is that it will execute any syntactically correct query. If you typed Borough instead of borough, it would still execute even though there is no key called Borough and it would return no documents. MongoDB is case sensitive, so if you are expecting results and get an empty query, check your spelling.

To see the number of documents in the result, a \$count stage can be added after the \$match stage. This is useful to know to check whether the results match your expectations, for example, but if counting the documents is not what is requested in the data demand, you can add the \$count stage for your own information and then delete it before adding additional stages to the pipeline.

5. **Click** on the **down arrow** to the right of Select... to see a dropdown menu of operations.
6. **Scroll** down the menu options and **choose \$count**.
7. **Type** over the highlighted word **string** with **tempCount** to see the result 3205.



8. Click the **trash can icon** to the right of the \$count operation to delete the \$count stage.

**Note:** The \$match query illustrated above is an exact match on two keys, where the separate conditions are evaluated using a logical AND operator. The keys are also first level keys, not nested, so they do not have to be contained in double quotes. To reference a nested key or an array element, double quotes and dot notation are used.

If a user would like to match the zipcode (a key in the address object), the following query can be used.

9. Click the **ADD STAGE** button below the \$match stage created in steps 3-4.
10. **Scroll** down the menu options and **choose \$match**.
11. **Type** over the highlighted word **query** with  
**"address.zipcode": "10021"**

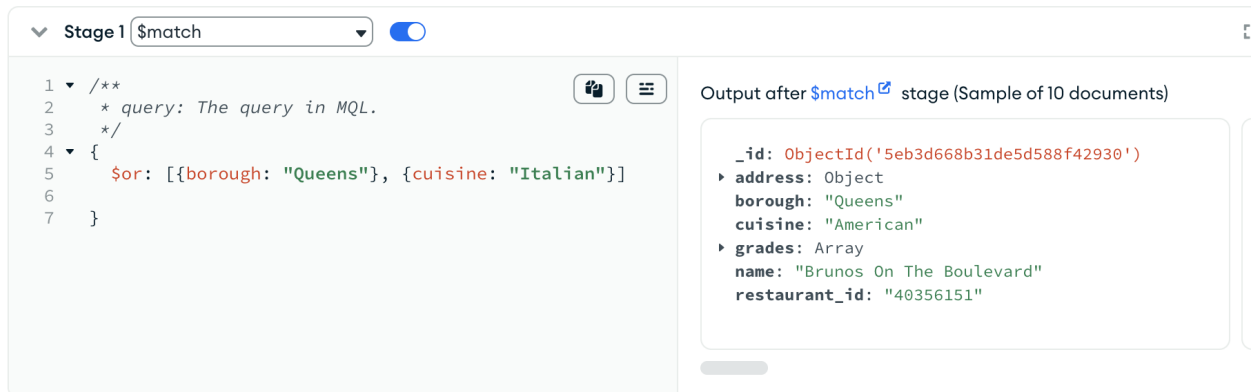
**Note:** If you use \$count to count the number of documents resulting from this \$match stage, you should see 40.

More complex matching conditions can be constructed using the logical operator \$or and the comparison operators \$gte, \$le, \$ne, and \$eq (greater than or equal to, less than or equal to, not equal to, and equal to, for example).

Reset the collection to be all of the restaurant documents by deleting any stages currently in your aggregation pipeline. Then find restaurants that are in Queens or serve Italian food.

12. **Remove any stages** in the current pipeline using the trash can icon to delete them as in step 8.
13. **Repeat steps 2-3** to add a \$match stage.
14. **Type** over the highlighted word **query** with  
**\$or: [{borough: "Queens"}, {cuisine: "Italian"}]**

15. **Make a screen capture** showing the **\$match** stage and the **first document** in the result.



**Note:** The **\$or** operator expects an array [ ] of matching conditions enclosed in { } and separated by commas. The **\$match** stage in step 14 should have 6,594 documents.

There are pattern matching functions that can be applied to strings, such as

- {name: /^abc/} returns documents where the name starts with “abc”
- {name: /abc\$/} returns documents where the name ends with “abc”
- {name: /^abc/i} returns documents that start with “ABC” or “abc” (the *i* option makes the pattern case insensitive)

To retrieve documents where the cuisine ends with “ican” and is not “American”

16. **Repeat steps 12-13** to clear the pipeline and add a **\$match** stage.

17. **Type** over the highlighted word **query** with  
**\$and: [{cuisine: {\$ne: "American"}}, {cuisine: /ican\$/}]**

**Note:** There are 822 documents for the **\$match** stage in step 17.

To search for items in arrays, use **\$all** and **\$in** operators. For example,

- {mylist: {\$all: [42, “kitty”]}} finds documents containing an array *mylist* with values of 42 and “kitty”
- {mylist: {\$in: [42, “kitty”]}} finds documents containing an array *mylist* with values of 42 or “kitty”

To find the documents where the restaurant has a score of 11 and score of 25 (there are 199 documents), create a **\$match** stage as follows.

18. **Repeat steps 12-13** to clear the pipeline and add a **\$match** stage.

19. **Type** over the highlighted word **query** with

**"grades.score": {\$all: [11, 25]}**

20. In a resulting document, **expand** the **grades** array and **expand** all of the **objects** in it to verify for yourself that one element has a score of 11 and another has a score of 25.

**Note:** If the user would like to retrieve a subset of a document's keys, you should use the **\$project** operation. To view selected keys, the **\$project** stage allows specified fields to be displayed. To omit the **\_id** key, specify **\_id: 0**. For any other key, using **: 0** eliminates only that field and displays all of the others. To explicitly include attributes, use **: 1**. Attributes that are not explicitly listed will not be included.

To display just the borough and cuisine for each document, without the object id, do the following steps.

21. **Click the trash can icon** to delete any stages in your aggregation pipeline.
22. **Click on the down arrow** to the right of Select... to see a dropdown menu of operations.
23. **Scroll down the menu options and choose \$project.**
24. **Type** over the highlighted word **specification(s)** with  
**borough: 1,**  
**cuisine: 1,**  
**\_id: 0**
25. **Make a screen capture** showing the **\$project** stage and the **first document** in the result.

Stage 1 **\$project** ☒

```
1  /**
2   * specifications: The fields to
3   * include or exclude.
4   */
5  {
6    borough: 1,
7    cuisine: 1,
8    _id: 0
9  }
10 }
```

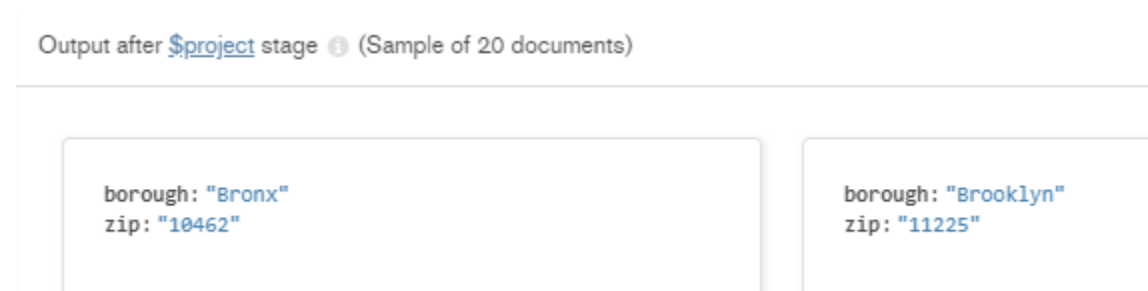
Output after **\$project** stage (Sample of 10 documents)

```
borough: "Brooklyn"
cuisine: "American"
```

**Note:** Projected attributes that are part of an object or an array will still be nested in the result unless a path expression is used. A path expression references a key or nested key by preceding the key name with **\$**. For example, the following query in a **\$project** stage does not display the **\_id** field and only displays **borough** and **address.zipcode** keys. The new key **zip** has an expression as its value that evaluates to the string **zipcode** embedded in the **address** object.

26. Click the **trash can icon** to delete any stages in your aggregation pipeline.
27. Click on the **down arrow** to the right of Select... to see a dropdown menu of operations.
28. Scroll down the menu options and **choose \$project**.
29. Type over the highlighted word **specifications** with  
**borough: 1,**  
**zip: "\$address.zipcode",**  
**\_id: 0**

**Note:** Using path expressions as the value in a key-value pair in a \$project stage is a useful way to promote an embedded key up one level. The results of step 29 are:



Some additional examples of the \$match and \$project stages are illustrated in the table below. For further examples, consult MongoDB online documentation for aggregation stages:

<https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/>

data demand	\$match and \$project examples
a. give documents where the key grades.score has a value greater than 10 for at least one element in the array grades	a. stage: \$match query: {"grades.score": {\$gt: 10}}
b. give documents where the key grades.score has a value less than 10 for at least one element in the array grades	b. stage: \$match query: {"grades.score": {\$lt: 50}}
c. give documents where there is at least one grades.score that is between 10 and 50	c. stage: \$match query: {\$and: [{"grades.score": {\$gt: 10}}, {"grades.score": {\$lt: 50}}]}
The result of (c) is 19,581 documents.	

<p>return restaurant_id, name, and cuisine for documents where the restaurant name starts with "Pizza" (the _id key will also be displayed).</p> <p>The result has 109 documents.</p>	<pre>stage: \$match query: {name: /^Pizza/}  stage: \$project query: {   restaurant_id: 1,   name: 1,   cuisine: 1}</pre>
<p>return name, borough, and cuisine for documents where the borough is Queens and the cuisine is African or Turkish (the _id key will not be displayed).</p> <p>The result has 13 documents.</p>	<pre>stage: \$match query: {borough: "Queens",   \$or: [{cuisine: "African"},         {cuisine: "Turkish"}]}  stage: \$project query: {   _id: 0,   name: 1,   borough: 1,   cuisine: 1,   zipcode: "\$address.zipcode"}</pre>

**Note:** To order documents, the \$sort stage allows keys and ascending/descending order to be specified, using 1 and -1, respectively. The following \$sort query orders the documents in ascending order by cuisine and descending order by name.

30. **Click** the **trash can icon** to delete any stages in your aggregation pipeline.
31. **Click** on the **down arrow** to the right of Select... to see a dropdown menu of operations.
32. **Scroll** down the menu options and **choose \$sort**.
33. **Type** over the words **field1: sortOrder** with  
**cuisine: 1,**  
**name: -1**
34. **Make a screen capture** showing the **\$sort** stage and the **first document** in the result.

▼ Stage1 \$sort

1 ▼ /\*\*

2   \* Provide any number of field/order pairs

3   \*/

4 ▼ {

5     cuisine: 1,

6     name: -1

7 }

8 }

Output after \$sort stage (Sample of 10 documents)

\_id: ObjectId('5eb3d669b31de5d588f48172')

▶ address: Object

borough: "Queens"

cuisine: "Afghan"

▶ grades: Array

name: "Tariq Afghan Kabab"

restaurant\_id: "50010806"

**Note:** The stages \$match, \$count, \$project, and \$sort can be combined to form an aggregation pipeline. A pipeline can contain multiple stages of the same type, as needed.

### Part 3: Write grouping and join queries

**Note:** In this part of the lab, you will learn how to group documents and performing aggregation operations over the groups, as well as how to combine two collections based on a common attribute (a join query).

You will use a different collection that contains Bed and Breakfast documents for this part of the lab.

1. **Click** on **sample** in the navigation pane on the left side of the Compass interface.
2. **Click** on the **CREATE COLLECTION** button on the top of the Collections pane.
3. In the Collection Name box, **type nyBnB**.
4. **Click** on **CREATE COLLECTION**.
5. Click on **nyBnB**.
6. **Click** on **Import Data** at the bottom of the screen.
7. In the pop-up window, **click** on **BROWSE**.
8. In the navigation pane on the left, **click** on **Desktop**.
9. In the center of the screen, **click** on **nyBnB.json**.
10. **Click** on **Open**.
11. **Click** on **IMPORT** and then **click** on **DONE** when the import is completed.
12. **Click** on **nyBnB** in the left navigation pane to make it the active collection.
13. **Click** on **Aggregations** in the right pane to go to the aggregation pipeline builder.

**Note:** 607 documents have been uploaded into the nyBnB collection.

To perform aggregation operations such as count, sum, average, maximum, and minimum, the \$group stage can create groups of documents. The general form of the \$group operator is

```
{ $group:
  { _id: <expression>,
    <field1>: { <accumulator1>: <expression1> },
    ...
  }
```

```

    <fieldN>: { <accumulatorN>: <expressionN>}
  }
}

```

If *<expression>* is *null*, the entire collection is used as the group. Each *<fieldK>* is a key that you define, and *<accumulatorK>* is a function such as \$sum, \$avg, \$max, or \$min that computes a value based on each document in the group. Each function can take a path expression or a constant as its value for *<expressionK>*. Recall that path expressions are a key value in double quotes with a \$ at the beginning of the key, for example, "\$name" where name is a key in the document.

To compute values of some aggregation functions over the entire nyBnB collection, you will use

```
_id: null
```

and some accumulator functions. The result of the accumulator function and its syntax are as follows:

- To count the number of documents in the group (in this case, the whole collection since *\_id* is null), use {\$sum: 1} as *<accumulator1>*: *<expression1>*. In this case, *<expression1>* is a constant value 1. The key *<field1>* that accumulates the value (call it groupCount), has the value 1 added to it for each document in the group.
- To compute the average number of bedrooms across the entire group, use \$avg as *<accumulator2>* and "\$bedrooms" as *<expression2>*. The key *<field2>* that accumulates the value (call it avgRooms), computes the average of the numeric field bedroom over the group.
- To compute the maximum number of persons accommodated across the entire group, use \$max as *<accumulator3>* and "\$accommodates" as *<expression3>*. The key *<field3>* that accumulates the value (call it maxAccommodates), computes the maximum of the numeric field accommodates over the group.
- To compute the total number of beds available across the entire group, use \$sum as *<accumulator4>* and "\$beds" as *<expression3>*. The key *<field3>* that accumulates the value (call it totalBeds), adds the value of the numeric field beds over the group.

14. In the first stage, **select \$group**.

15. **Replace** all of the text between { and } with

```

_id: null,
groupCount: {$sum: 1},
avgRooms: {$avg: "$bedrooms"},
maxAccommodates: {$max: "$accommodates"},
totalBeds: {$sum: "$beds"}

```

**Note:** The results of step 15 are:



```

_id: null
groupCount: 607
avgRooms: 1.1452145214521452
maxAccommodates: 13
totalBeds: 916

```

To create groups over a key field or fields, the `_id` key should have an object with the attribute or attributes as its value. In other words, the keys that are the basis for the group must be list of key-value pairs separated by commas and enclosed in `{ }`, if there is more than one. If there is only one, it is still enclosed in `{ }` but no comma is needed.

To group by `room_type` and `bed_type` and compute the same functions as in the previous `$group` stage:

16. In the current `$group` stage created in step 15, replace `null` with **`{room: "$room_type", bed: "$bed_type"}`**

17. In the result, **expand `_id`**: to show the key fields.

18. **Make a screen capture** showing the `$group` stage and the **first document** in the result.

Stage 1
\$group

```

1  ▾
2  .id: The id of the group.
3  fieldN: The first field name.
4
5  ▾
6  id: {room: "$room_type", bed: "$bed_type"},
7  groupCount: {$sum: 1},
8  avgRooms: {$avg: "$bedrooms"},
9  maxAccommodates: {$max: "$accommodates"},
10 totalBeds: {$sum: "$beds"}
11
12

```

Output after `$group` stage (Sample of 10 documents)

▾ \_id: Object
room: "Shared room"
bed: "Real Bed"
groupCount: 2
avgRooms: 1
maxAccommodates: 2
totalBeds: 3

**Note:** To compute the equivalent of a SQL GROUP BY and HAVING query, the `$group` stage can be followed by a `$match` stage to accomplish the HAVING filter on groups. To filter the groups constructed in step 16 to include only groups where the total number of beds is greater than 100:

19. **Add a `$match` stage** after the `$group` stage created in step 16 with the condition

**totalBeds: {\$gt: 100}**

20. In the result, **expand** `_id`: to show the key fields.

21. **Make a screen capture** showing the **\$match** stage and the **first document** in the result.



**Note:** If the user would like to combine two collections based on a common key value, you should write a join query. To combine collections based on a matching a condition, the \$lookup stage computes the document-oriented version of left join. The \$lookup stage can also be used for computing inner join and subtraction. To demonstrate \$lookup, two collections are needed. The collections should be in the same database.

To find restaurants that are in the same suburb (or borough) as a New York BnB, you will join two collections in the sample database: nyBnB and restaurants on a common key (address.suburb in nyBnB and borough in restaurants.)

22. **Delete the stages** in the current aggregation pipeline on nyBnB, if any.

23. **Select \$lookup** as a new stage.

**Note:** The current collection where the aggregation pipeline is active is the source (or left side of the join). The from: key is the target collection (or the right side of the join). The localField: key is the source field to match to the foreignField: key in the target collection. The as: key specifies the name of the array that will hold the joined objects.

24. **Replace string** after from: with **restaurants**.

25. **Replace string** after localField: with **address.suburb**.

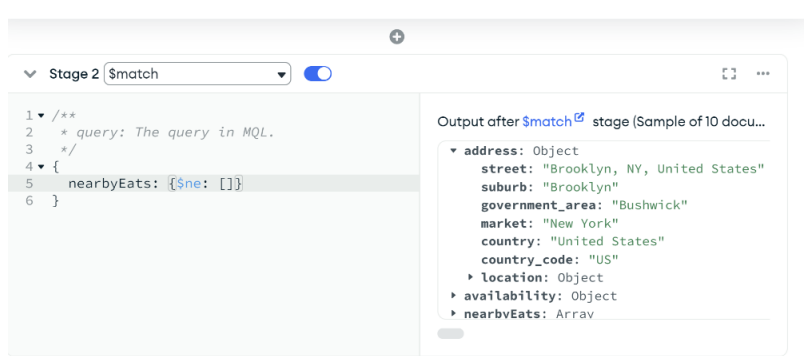
26. **Replace string** after foreignField: with **borough**.

27. **Replace string** after as: with **nearbyEats**.

**Note:** Every nyBnB document will have an array called nearbyEats. Some will have hundreds of objects in the array, one for each restaurant where the restaurant's borough is the same as the BnB's suburb. Some BnB's do not have a suburb in their address, or the value does not match any borough. In that case, nearbyEats will be an empty array [] (consecutive [ and ] with no space between.)

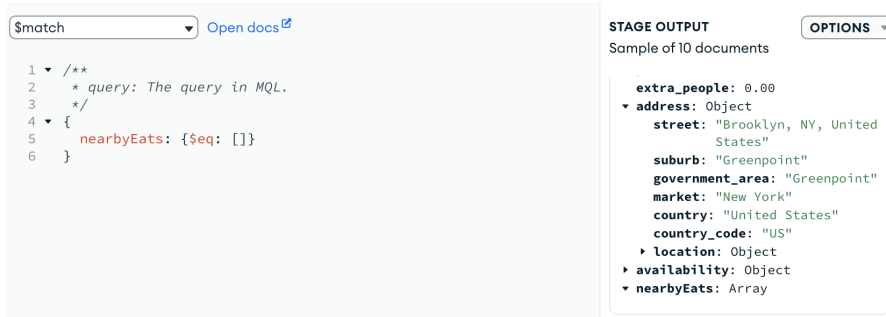
Source documents, in this case BnBs, appear in the resulting collection even if they do not have a match (essentially computing a left join). To compute an inner join, a \$match stage following a \$lookup stage with \$ne: [] eliminates all source documents that do not have a match in the target collection. There should be 306 documents.

28. Add a \$match stage after the \$lookup stage created in step 27 with the condition **nearbyEats: {\$ne: []}**
29. In the result, **expand address:** to show the key fields.
30. Make a screen capture showing the \$match stage and the **suburb** in the **first document** in the result.



**Note:** To perform subtraction, a \$match stage with \$eq: [] finds all source documents that do not have a match in the target collection. There should be 301 documents.

31. Replace the **condition** in the \$match stage of step 28 with **nearbyEats: {\$eq: []}**
32. In the result, **expand address:** and **nearbyEats:** (nearbyEats should be empty).
33. Make a screen capture showing the \$match stage and both the **suburb** and **nearbyEats** in the **first document** in the result.



## Part 4: Creating an aggregate pipeline to answer a data demand

**Note:** In this part of the lab, you will create an aggregate pipeline to retrieve the data to satisfy a user's request. The user would like to know the theaterId of theaters and names of restaurants and their streets, where the restaurants serve Turkish cuisine and are in the same zipcode as the theater, sorted by street in ascending order.

You will use a collection that contains theater documents for this part of the lab as well as the restaurants collection created in Part 1.

Two new stages are introduced in this part of the lab:

- You will learn how to create a view using the \$out stage to save intermediate results.
- You will learn how to create new documents from individual array elements using \$unwind.

1. **Click** on **sample** in the navigation pane on the left side of the Compass interface.
2. **Click** on the **CREATE COLLECTION** button on the top of the Collections pane.
3. In the Collection Name box, **type nyTheaters**.
4. **Click** on **CREATE COLLECTION**.
5. **Click** on **nyTheaters**.
6. **Click** on **Import Data** at the bottom of the screen.
7. In the pop-up window, **click** on **BROWSE**.
8. In the navigation pane on the left, **click** on **Desktop**.
9. In the center of the screen, **click** on **nyTheaters.json**.

10. Click on **Open**.

11. Click on **IMPORT** and then click on **DONE** when the import is completed.

**Note:** 81 documents have been uploaded into the nyTheaters collection.

This query will require joining nyTheaters to restaurants where the zipcode is the same. To make the join smaller, an intermediate collection of restaurants with Turkish cuisine is constructed first. The intermediate collection, or view, will be used as the target in a join later.

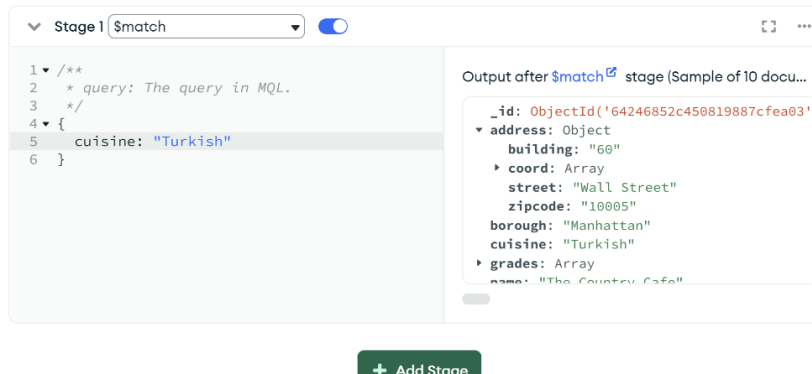
12. Click on **restaurants** in the left navigation pane to make it the active collection.

13. Click on **Aggregations** in the right pane to go to the aggregation pipeline builder.

14. Create a **\$match** stage to select restaurants where the **cuisine** is **Turkish**.

15. Expand the **address** object in the first document in the result.

16. Make a screen capture showing the **\$match** stage and both the **street** and **cuisine** in the first document in the result.



**Note:** there should be 70 documents in the \$match stage. You can determine this by adding a temporary \$count stage after the \$match stage.

Since not all the fields will be needed for the final result, you can keep the ones that are using \$project. The restaurant name and street are needed for the final result, and the zipcode is needed as a join field.

17. Below the \$match stage, create a **\$project** stage to show the restaurant **name**, **street**, and **zipcode** at the first level.

**Note:** The result should contain 70 documents with three first-level (not nested) string-valued attributes called name, street, and zipcode.

Now use the \$out stage to create a view called turkishRestaurants.

18. Below the \$project stage, **create** an **\$out** stage.
19. **Type turkishRestaurants** in place of **string** in the \$out stage.
20. In the result pane, **click** on the **SAVE DOCUMENTS** button.

**Note:** A new collection appears in your sample database called turkishRestaurants. Now you will return to the nyTheaters collection to use it as the source for the join with turkishRestaurants.

21. **Click** on **nyTheaters** in the left navigation pane to make it the active collection.
22. **Click** on **Aggregations** in the right pane to go to the aggregation pipeline builder.
23. **Create** a **\$lookup** stage.
24. **Replace string** after from: with **turkishRestaurants**.
25. **Replace string** after localField: with **location.address.zipcode**.
26. **Replace string** after foreignField: with **address.zipcode**.
27. **Replace string** after as: with **turkishEats**.
28. **Make a screen capture** showing the **\$lookup** stage and the **first document** in the result.

The screenshot shows the MongoDB Aggregation Pipeline Builder interface. On the left, the aggregation pipeline is defined with a \$lookup stage. The pipeline is as follows:

```
1  /**
2   * from: The target collection.
3   * localField: The local join field.
4   * foreignField: The target join field.
5   * as: The name for the results.
6   * pipeline: Optional pipeline to run on the matched documents.
7   * let: Optional variables to use in the pipeline.
8   */
9  {
10   from: 'turkishRestaurants',
11   localField: 'location.address.zipcode',
12   foreignField: 'address.zipcode',
13   as: 'turkishEats'
14 }
```

On the right, the output of the \$lookup stage is displayed. The output is a sample of 10 documents, showing the first document:

```
{
  "_id": ObjectId('59a47286cfa9a3a73e51e744'),
  "theaterId": 1028,
  "location": Object,
  "turkishEats": Array
}
```

**Note:** There should be 81 documents in the result since no theaters were removed. Now you will keep only the theaters that have a Turkish restaurant in the same zipcode.

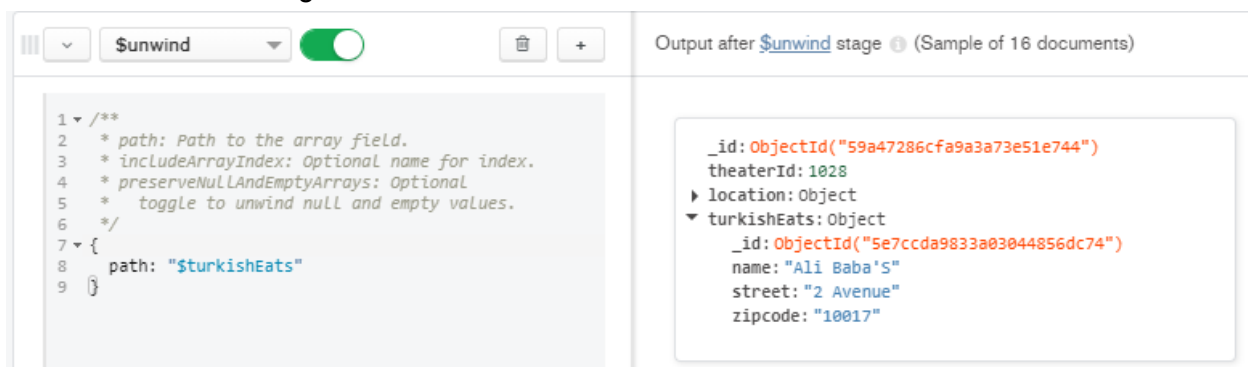
29. Add a **\$match** stage after the **\$lookup** stage with the condition **turkishEats: {\$ne: []}**

**Note:** Now you should have only 9 documents in your result. Each theater has an array of Turkish restaurants. In order to bring the array elements up to the first level (remove nesting), the **\$unwind** stage will be used. It creates new documents where the array is replaced by an array element, for each element in the array.

30. Add an **\$unwind** stage after the **\$match** stage.

31. Replace all the text between **{** and **}** with **path: "\$turkishEats"**

**Note:** The **\$unwind** stage should look as follows and include 16 documents:



The next stage is to project theaterId, restaurant name, and street. The final stage is to sort the results by street in ascending order.

32. Create a **\$project** stage that shows only **theaterId**, **name**, and **street** (not including **\_id**) as first level keys in the resulting documents.
33. Create a **\$sort** stage to sort the results in ascending order by street.
34. Make a screen capture showing the **\$project** and **\$sort** stages and the **first two documents** in the result.


1▼ /\*\*  
2 \* specifications: The fields to  
3 \* include or exclude.  
4 \*/  
5 {  
6 \_id: 0,  
7 theaterId: 1,  
8 name: "\$turkishEats.name",  
9 street: "\$turkishEats.street"  
10  
11 }

Output after [\\$project](#) stage (Sample of 10 documents)

theaterId: 1028  
name: "Ali Baba'S"  
street: "2 Avenue"

theaterId: 1028  
name: "Adana Grill"  
street: "3 Avenue"

t  
n  
s

Stage 5 [\\$sort](#) 

1▼ /\*\*  
2 \* Provide any number of field/order pairs.  
3 \*/  
4 {  
5 street: 1  
6 }

Output after [\\$sort](#) stage (Sample of 10 documents)

theaterId: 1908  
name: "Hanci Turkish Cuisine"  
street: "10 Avenue"

theaterId: 1028  
name: "Ali Baba'S"  
street: "2 Avenue"

t  
n  
s

24