HOMEWORK – 4
DATA 225-11 DB Systems for Analytics

Group 7:
Vishal Tripathi (015213224)
Harshvardhan Singh Gahlaut (017441632)
Shubhang Kalkar (016942393)
Mukul Mahajan (017443907)
Saba Tehrani (016802240)

1. Meaningful Goal

The goal of this project is to solve two problems. The first problem is solved using the graph NoSQL database Neo4j for a special NBA data analysis. We are analyzing interesting trends, player interactions, and team dynamics by modeling NBA-related elements as nodes and their relationships. In this project, a graph database provides more complex pictures of the relationships in the NBA ecosystem. The second problem is solved using the superstore dataset. We used Python to carry out CRUD operations, and MongoDB Charts were used for visualization. We are using Neo4j's containerization and orchestration, MongoDB's sharding, and cluster computing. Overall, our analysis's main objective is to offer insightful information about the NBA Dataset, enabling a deeper interpretation of the elements that are statistically significant in the dataset.

2. Workflow

The project workflow consists of multiple important phases. Data input is the first step, during which pertinent information is gathered and entered into the system. Graph modeling is the next step: building a graph database that faithfully depicts the connections between data elements. Next, Cypher queries are created to draw significant conclusions from the graph model.

The next step is resulting interpretation, which entails analyzing the collected data to make inferences and spot trends. Comprehending these results is achieved using visualization tools. Crucially, the approach is iterative, which means that improvements are continuously made to the network model and Cypher queries based on preliminary findings to improve the breadth and precision of the studies that follow.

3. Innovation

A paradigm change in sports analytics is provided by the innovative method of analyzing the complex web of relationships inside the NBA (National Basketball Association) utilizing a graph database. Graph databases are an excellent tool for understanding the intricate links that exist in the world of professional basketball since they are specially designed to capture and portray complicated interconnections.

It is feasible to obtain fresh and deeper insights into player interactions, team dynamics, and statistical correlations by utilizing a graph database. A graph database is superior at simulating the complex web of relationships between players, teams, coaches, and other entities in the NBA ecosystem compared to traditional relational databases, which frequently find it difficult to manage the complicated nature of these interactions.

This creative approach has the potential to completely change how sports fans and pundits view the game. It can reveal relationships and trends that were previously overlooked, offering new insights into player effectiveness, team tactics, and the league's overall dynamics. This method can improve team decision-making, guide strategic decisions, and ultimately improve the fan experience by providing a deeper knowledge of the NBA's complex web of linkages, thanks to the abundance of data available in professional sports.

Link to the datasets used: Group7_HW4
Neo4j Dataset: records.json (NBA Dataset)
MongoDB Dataset: Superstore.csv

4. Technical Difficulties

Solving technological issues with graph databases, like Neo4j, requires consideration of multiple important factors. First, for a thorough study of related data, complex Cypher queries must be created. This entails comprehending the linkages and data structure to create exact queries that yield insightful information.

Large dataset management in graph databases can be difficult because of the possibility of increased complexity and performance problems. Effective data management requires optimizing procedures for data retrieval and storage. Furthermore, creating an effective graph model is essential to guarantee that the database schema efficiently accommodates the intended use cases and queries.

Investigating scale-out options, such as distributed graph databases, may be a practical answer in situations when datasets grow dramatically. Because of this, the database may be made to scale horizontally, handling larger datasets while maintaining performance. It is imperative that you evaluate the requirements and limitations of your project to determine the best approaches and tactics for efficiently handling and interpreting graph data.

5. Analytics using Queries:

Neo4j Queries:
CREATING DATABASE:

Examples of a few entries that are done on cypher are given below and each of the 2 queries are attached for every type of relationship to understand the basic structure of our dataset.

a. Player details:

```
CREATE
(russell:PLAYER{name:"Russell Westbrook", age: 33, number: 0, height: 1.91, weight: 91}),
(lebron:PLAYER{name:"LeBron James", age: 36, number: 6, height: 2.06, weight: 113})
```

This query creates nodes with different players in the NBA and their biographic details .

b. Coach details:

```
(frank:COACH{name: "Frank Vogel"}),
(taylor:COACH{name: "Taylor Jenkins"})
```

This query creates the nodes for coaches that are there in teams and have their names.

c. Team details:

```
(lakers:TEAM{name:"LA Lakers"}),
(memphis:TEAM{name:"Memphis Grizzlies"})
```

This query gives names for every team that is there in the NBA and creates their nodes.

d. Teammates relationship:

```
(lebron)-[:TEAMMATES]-> (russell),
(lebron)<-[:TEAMMATES]- (russell)
```

Here, this is a relationship example where the relationship name is “TEAMMATES”, showing which player is the teammate of which other players.

e. Coaches’ relationship:

(frank)-[:COACHES]->(lebron),
(frank)-[:COACHES]->(anthony)

This relationship represents which coach node coaches which player node and the name of the relationship is “COACHES” which helps us understand the relationships between players and coaches and which coach belongs to which player.

f. Plays for relationships:
(lebron)-[:PLAYS_FOR {salary: 40000000}]-> (lakers),
(russell)-[:PLAYS_FOR {salary: 33000000}]-> (lakers)

In this relationship example, we can see that it relates - which player plays for which team and a variable salary is attached showing us the price at which that player is playing for. The name of the relationship is “PLAYS_FOR”.

g. Coaches for relationship:
(frank)-[:COACHES_FOR]->(lakers),
(taylor)-[:COACHES_FOR]->(memphis)

In the relationship given here, the name is “COACHES_FOR” and this relationship shows which coach in the dataset coaches which team specifically.

h. Played against relationship:
(lebron)-[:PLAYED_AGAINST {minutes: 38, points: 32, assists: 6, rebounds: 6, turnovers: 2}]-> (memphis),
(russell)-[:PLAYED_AGAINST {minutes: 29, points: 16, assists: 12, rebounds: 11, turnovers: 16}]-> (memphis)

In the relationship here, we create links to all the players that ever played against each other and show which player played against who, hence connecting the nodes.

COMMAND AND OUTPUT EXPLAINATION:

a. To create node for player and show team:

In the query as seen here, we are creating a player detail named Lebron that has name and height in it and continuing to show how much he plays for and what team he is in. That creates 2 nodes and 4 labels with 4 properties as seen in the output.

b. Property updating:

```
MATCH (lebron {name: "LeBron James"}) - [contract:PLAYS_FOR] → (:TEAM)
SET contract.salary = 60000000
```

Table | Code

Set 1 property, completed after 13 ms.

In the query here, we are setting the contract salary of lebron james from 40000000 to 60000000 where the relation is PLAYS_FOR, and updating the same.

c. To find highest scoring player in Laker(team):

```
// GET HIGHEST SCORING PLAYER IN THE LAKERS //
MATCH (player:PLAYER) - [:PLAYS_FOR] - (:TEAM {name: "LA Lakers"})
MATCH (player) - [gamePlayed:PLAYED_AGAINST] - (:TEAM)
RETURN player.name, AVG(gamePlayed.points) AS ppg
ORDER BY ppg DESC
LIMIT 1
```

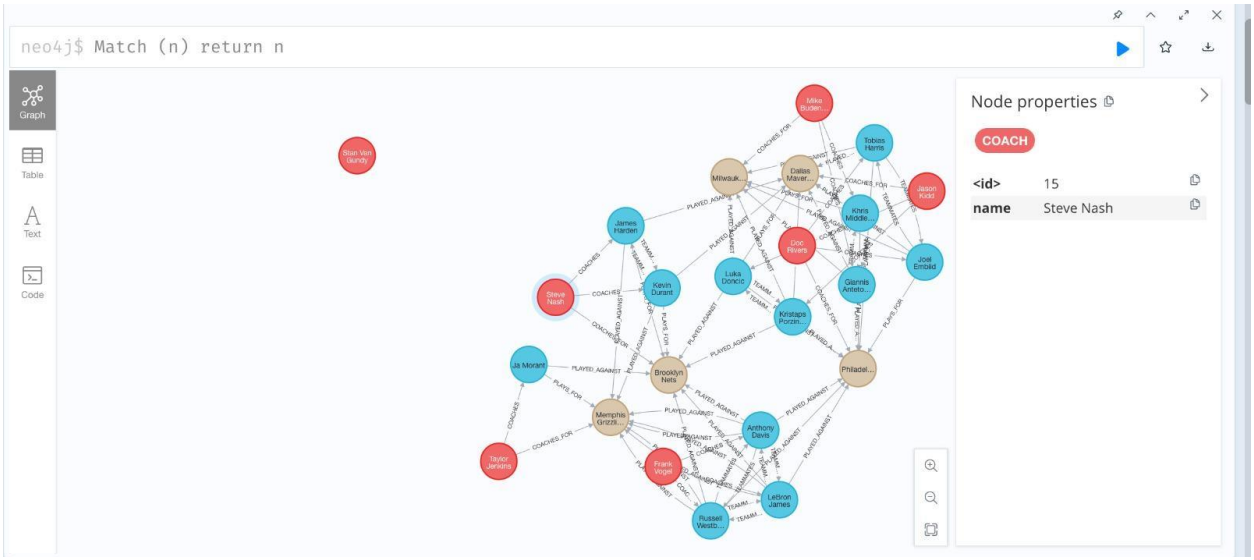
Table | Text

	player.name	ppg
1	"LeBron James"	25.5

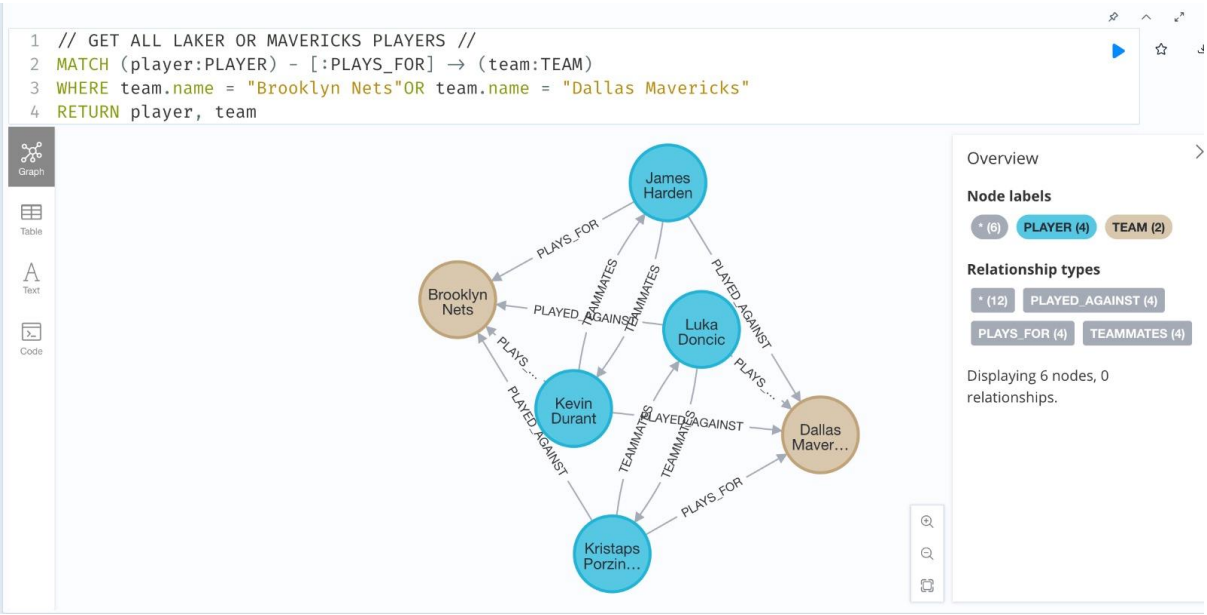
In the query here, we are trying to find who is the highest-scoring player in the LA Lakers (lakers) team and we are matching the player who plays for the lakers and who has played against all the teams on the basis of that, returning the avg points scored by each player and then finding out who has the highest score of all by placing the scores in descending order and the highest scoring player is returned who is Lebron James in this case.

d. Showing entire dataset connections in the form of graph network:

In the plot here, we can see that the query returns all the possible node connections amongst all the nodes and their respective relationships, and it gives us an overview of the dataset and database and what connections are there in order to provide us with a broader perspective so that we can infer what all details we can fetch from it. Also, there are 3 different nodes that we see which are for “Players”, “Coaches” and “Teams” which are Blue, Red, and Beige respectively.

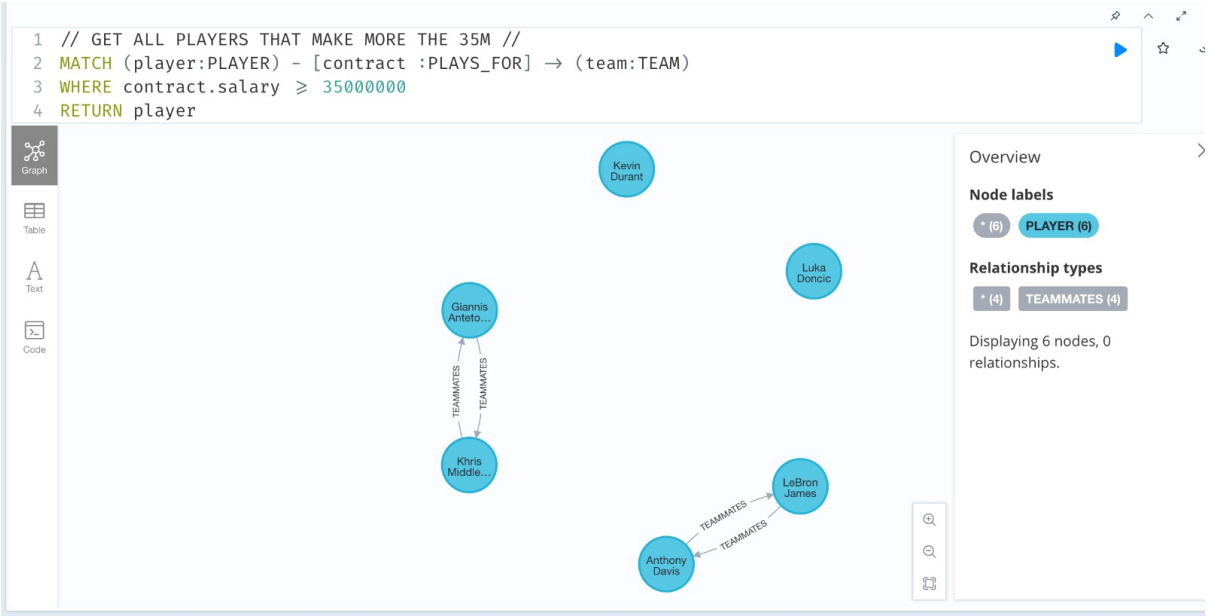


e. Query to get all lakers or mavericks players and check the network:



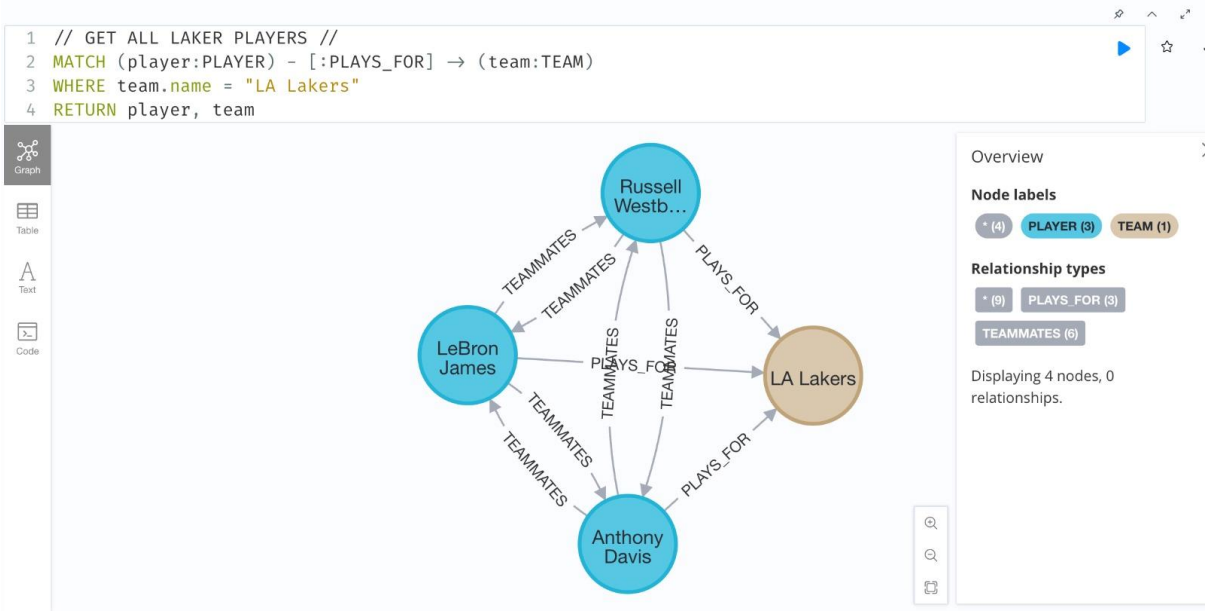
This query here matches and returns all the players in LA Lakers or Dallas Mavericks and connects them with either of the 2 teams, also shows relations among each player hence giving us a better relation understanding of who played against whom. There can be seen 3 relationships seen here namely plays_for, teammates and plays_against to clarify what each player node is to other nodes.

f. Player earnings analysis:



The query as shown helps us understand the earnings of the players in the database and here we have a constraint that only gives us the players who earn more than 35 million dollars. Not only that but we also get an analysis of whether any of the players are on the same team or not, hence giving us a better understanding of the economic stability of the teams as well since we also get the number of players from 1 team here. We can use this query to analyze the player economy and change the value accordingly to understand and predict how much that player can be sold to another team and so on.

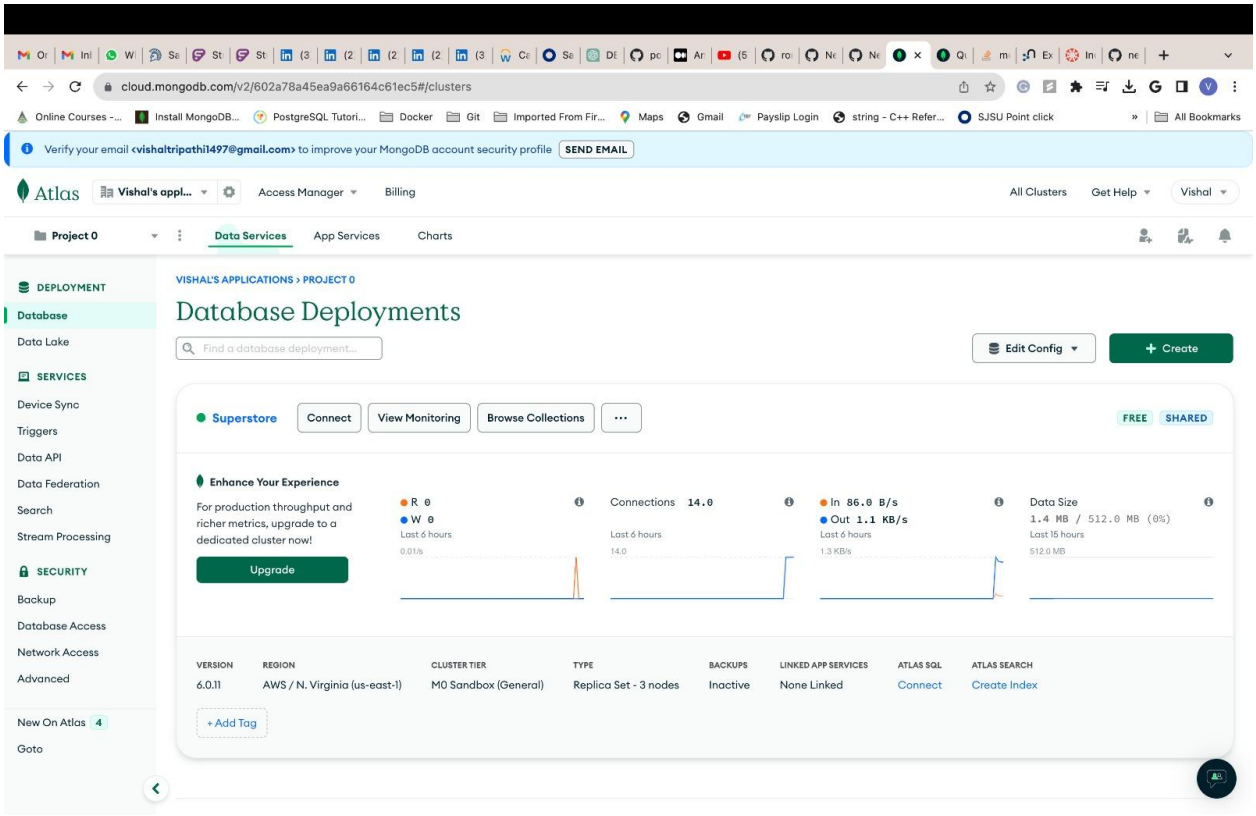
g. Team player details:



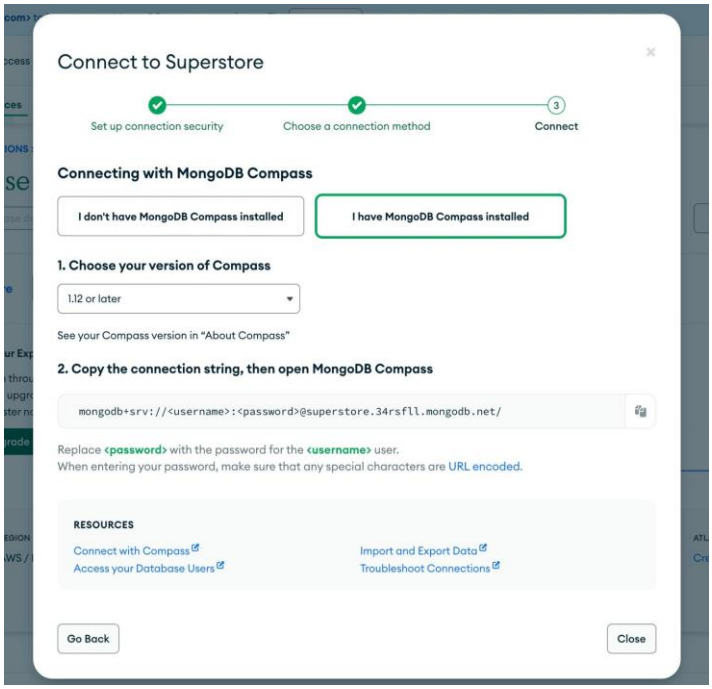
From the query here, we are printing all the players in a specific team individually to understand what players are in what teams and fetch the details pictorially rather than reading the data through. We can use this query to see the total number of players in the same team from the dataset here we have the players in Lakers there are 2 relationships one being teammate showing what players are teammates to each other and the other plays_for to show each player playing for what team. This analysis can be very useful when we wish to compare maybe that in the top 10 players list, and how many of them belong to the same team, giving us an upper edge to analyse the training strength of a team.

MongoDB Queries:
Setting up the Connection:

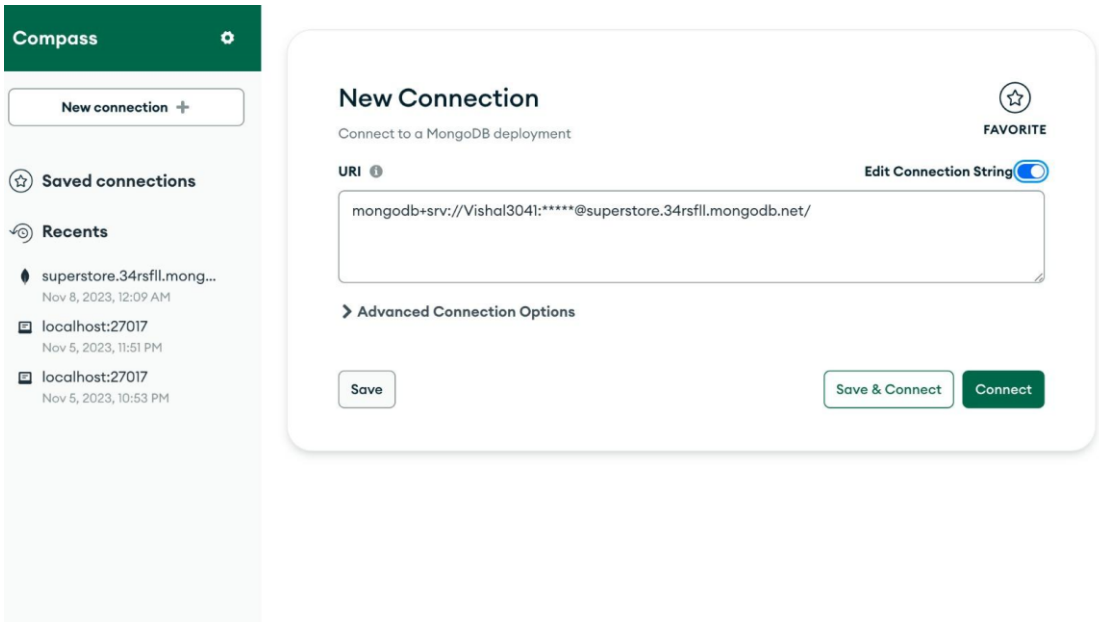
STEP-1: First go to the Atlas website and open the page. There click on “create” and the connection page will open.



STEP-2: Now on the pop-up, if you have MongoDB compass, select the option saying, “I have MongoDB compass installed” and choose the version that you have you will get a URL, copy that.



STEP-3: Now open MongoDB compass and in the new connection, it will be asking for a URL or a string to connect to. Paste the link you just copied on the portal and hit connect. The connection is established.



Aggregate and CRUD operations:

a. Python command for connection and importing CSV Data to MongoDB

The following code is executed to set up a Python Connection with MongoDB using pymongo. It reads the data from the CSV file and imports the data from the CSV file to create the documents.

```
[47]: import csv
      from pymongo import MongoClient

      # Establish a connection to MongoDB
      client = MongoClient("mongodb+srv://[username]:[password]@superstore.34rsfll.mongodb.net/")
      db = client["Superstore_database"]

      # Load data from CSV into MongoDB collections
      def import_data_into_collection(csv_filename, collection_name, relevant_columns):
          collection = db[collection_name]
          with open(csv_filename, 'r') as csvfile:
              reader = csv.DictReader(csvfile)
              for row in reader:
                  document = {key: row[key] for key in relevant_columns}
                  collection.insert_one(document)

      # Define relevant columns for each collection
      customer_columns = ['customer_id', 'customer_name', 'customer_segment', 'city', 'state', 'postal_code']
      order_columns = ['order_id', 'customer_id', 'item_id', 'order_date', 'order_quantity', 'shipping_cost',
      product_columns = ['item_id', 'item', 'category', 'department', 'unit_price']
      sales_columns = ['item_id', 'region', 'sales']

      # Import data into relevant collections
      import_data_into_collection('superstore.csv', 'customers', customer_columns)
      import_data_into_collection('superstore.csv', 'orders', order_columns)
      import_data_into_collection('superstore.csv', 'products', product_columns)
      import_data_into_collection('superstore.csv', 'sales', sales_columns)

      # # Close the MongoDB connection
      # client.close()
```

Here, we are connecting Python with MongoDB with the help of pymongo by providing the database name and the URL that we created earlier. Then we read the cleaned and refined **superstore** CSV file where we are reading the elements in the file with the help of for loops. Lastly, we are importing the CSV file in the database and creating 4 different tables namely customers, orders, products, and sales, and allotting the column names from our CSV file to input in the database for entries and creating the document.

b. Update Command

The following code updates one row in the database where the customer_id = 3035.

```
[57]: # Update a single document
      query = {"customer_id": "3035"}
      new_values = {"$set": {"customer_name": "Harshal Shah"}}
      db.customers.update_one(query, new_values)
```

```
[57]: UpdateResult({'n': 1, 'electionId': ObjectId('7fffffff0000000000000f1'), 'opTime': {'ts': Timestamp(16
      99434567, 3), 't': 241}, 'nModified': 1, 'ok': 1.0, '$clusterTime': {'clusterTime': Timestamp(169943456
      7, 3), 'signature': {'hash': b'QF\x11\xc0\x84\xb5:\x02T\x8e9\x98\xf3\x1f\xd1;g\xabsZ', 'keyId': 7233456
      183300849671}}, 'operationTime': Timestamp(1699434567, 3), 'updatedExisting': True}, acknowledged=True)
```

In the command here, we are updating a row where the customer_id is 3035 and changing the name from old to new (harshal shah).

c. Finding all the rows:

The following code reads all the rows where customer_id = 3035. The output shows 2 rows.

```
[61]: ans = db.customers.find({"customer_id": "3035"})
```

```
[63]: for x in ans :
          print(x)
```

```
{'_id': ObjectId('654b274335f7479ef5356032'), 'customer_id': '3035', 'customer_name': 'Harshal Shah',
'customer_segment': 'Home Office', 'city': 'Lombard', 'state': 'Illinois', 'postal_code': '60148'}
{'_id': ObjectId('654b274435f7479ef5356033'), 'customer_id': '3035', 'customer_name': 'Mark Bailey', 'c
ustomer_segment': 'Home Office', 'city': 'Lombard', 'state': 'Illinois', 'postal_code': '60148'}
```

In this code/ command, we are finding all the rows where customer_id is 3035 giving us all the orders by that customer and as an output, we can see that there are 2 rows returned.

d. Deleting command:

The following code deletes one row where customer_id = 3035

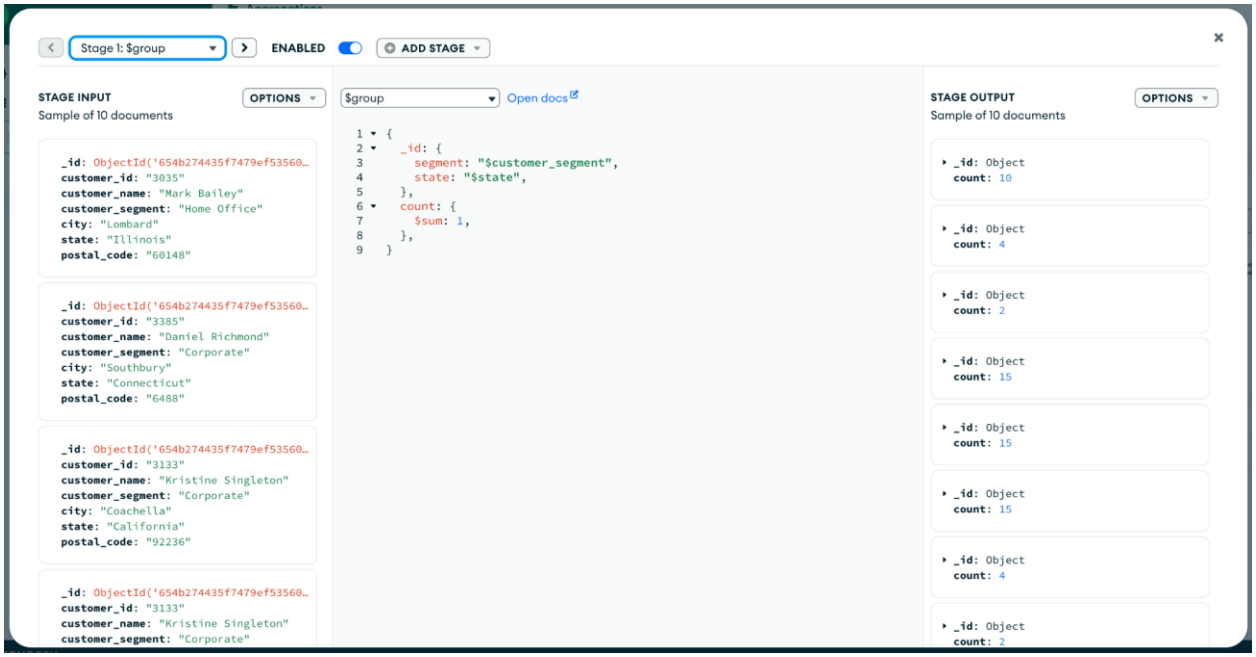
```
[66]: db.customers.delete_one({"customer_id": "3035"})

[66]: DeleteResult({'n': 1, 'electionId': ObjectId('7fffffff0000000000000f1'), 'opTime': {'ts': Timestamp(16
      99434746, 36), 't': 241}, 'ok': 1.0, '$clusterTime': {'clusterTime': Timestamp(1699434746, 36), 'signat
      ure': {'hash': b'\xac\xe6jS8\xf5F\xaf"5#S\xbf\xe9\x88\x16\\\xb0(\x02', 'keyId': 723345618330084967 1}},
      'operationTime': Timestamp(1699434746, 36)}, acknowledged=True)
```

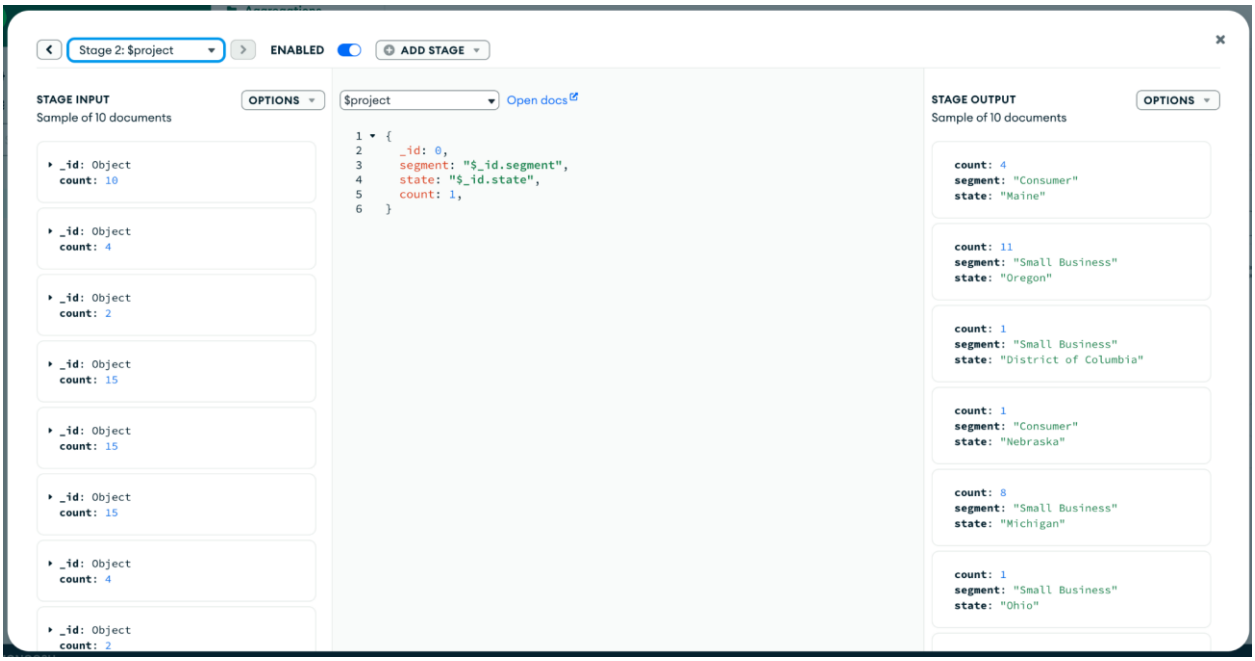
In the command here, we are deleting the row where the customer_id is 3035 and removing it from the database completely as a CRUD operation example.

Aggregation Pipelines

Query 1:

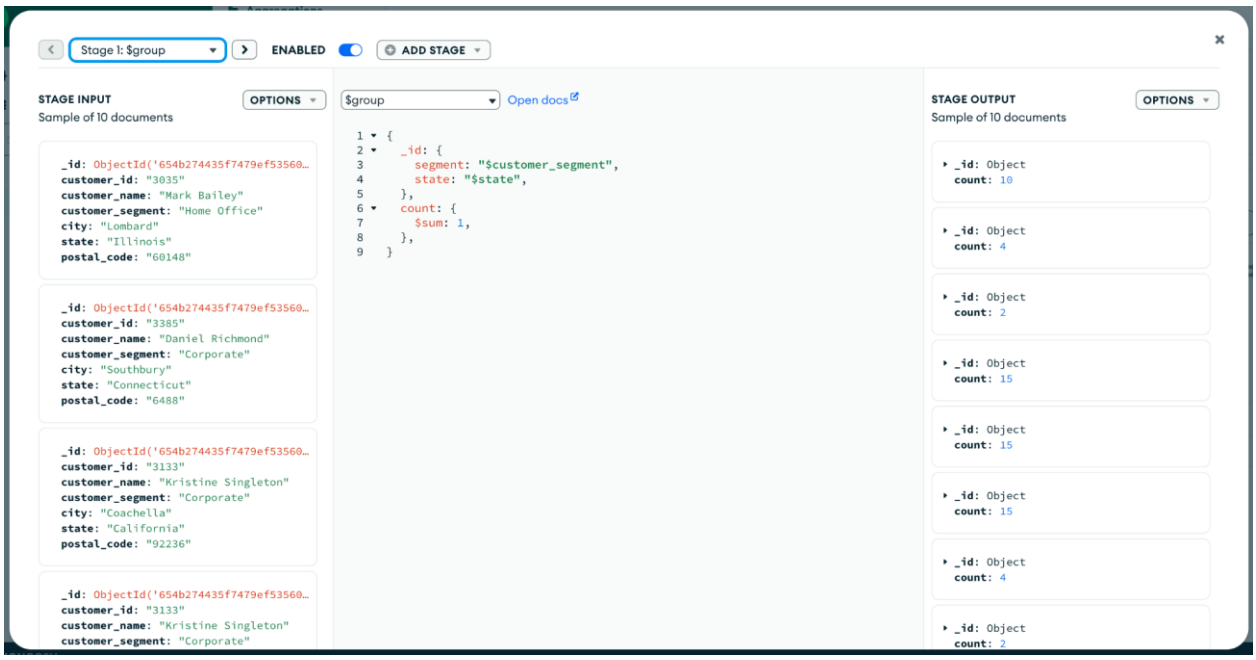
It returns the total Count of Customers, segment name, and state name who are from the same segment and reside in the same state. It consists of 2 stages, both attached.

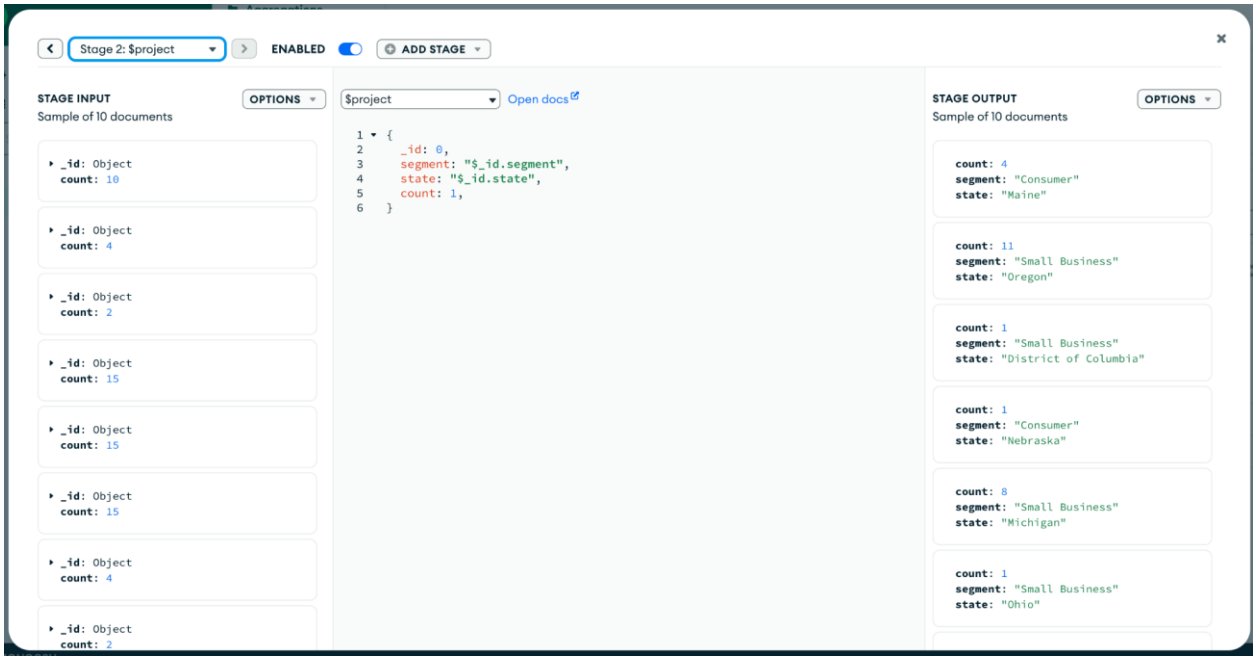




Query 2:

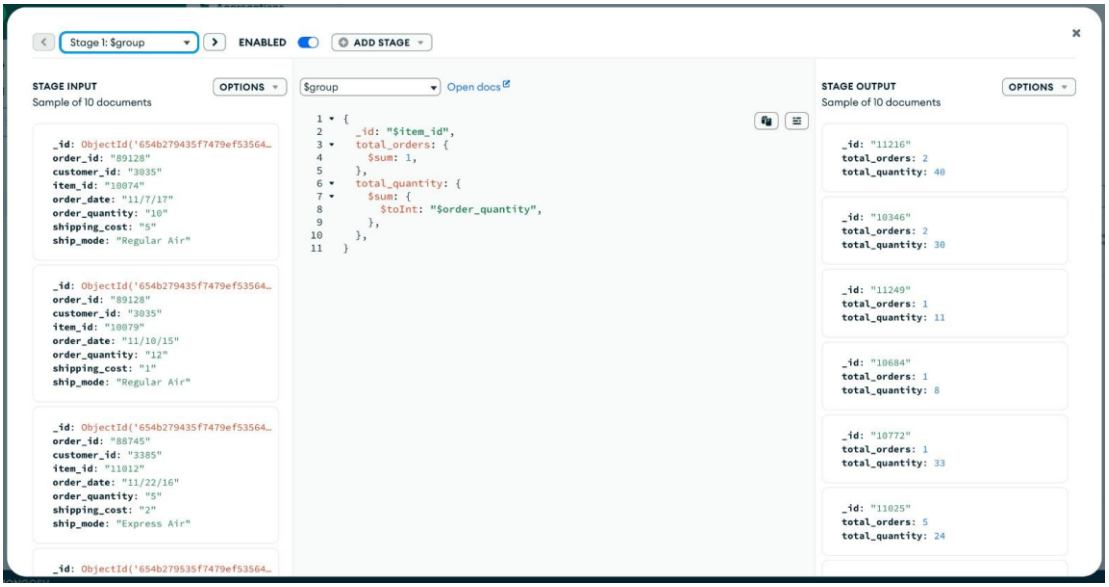
It returns the customer_id who ordered with a total of Order Quantity and a total of Shipping Cost by the Customer.

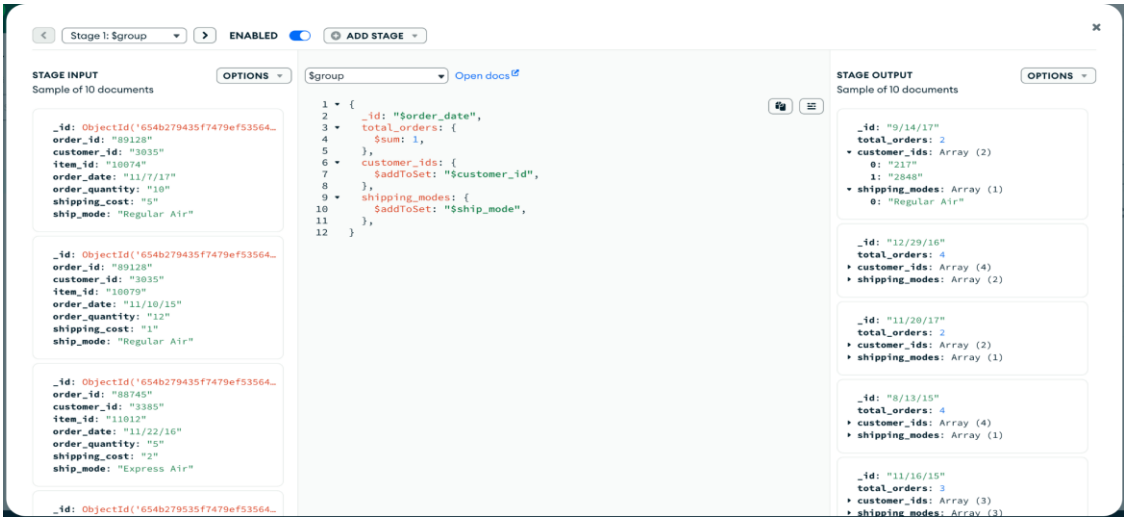




Query 3:

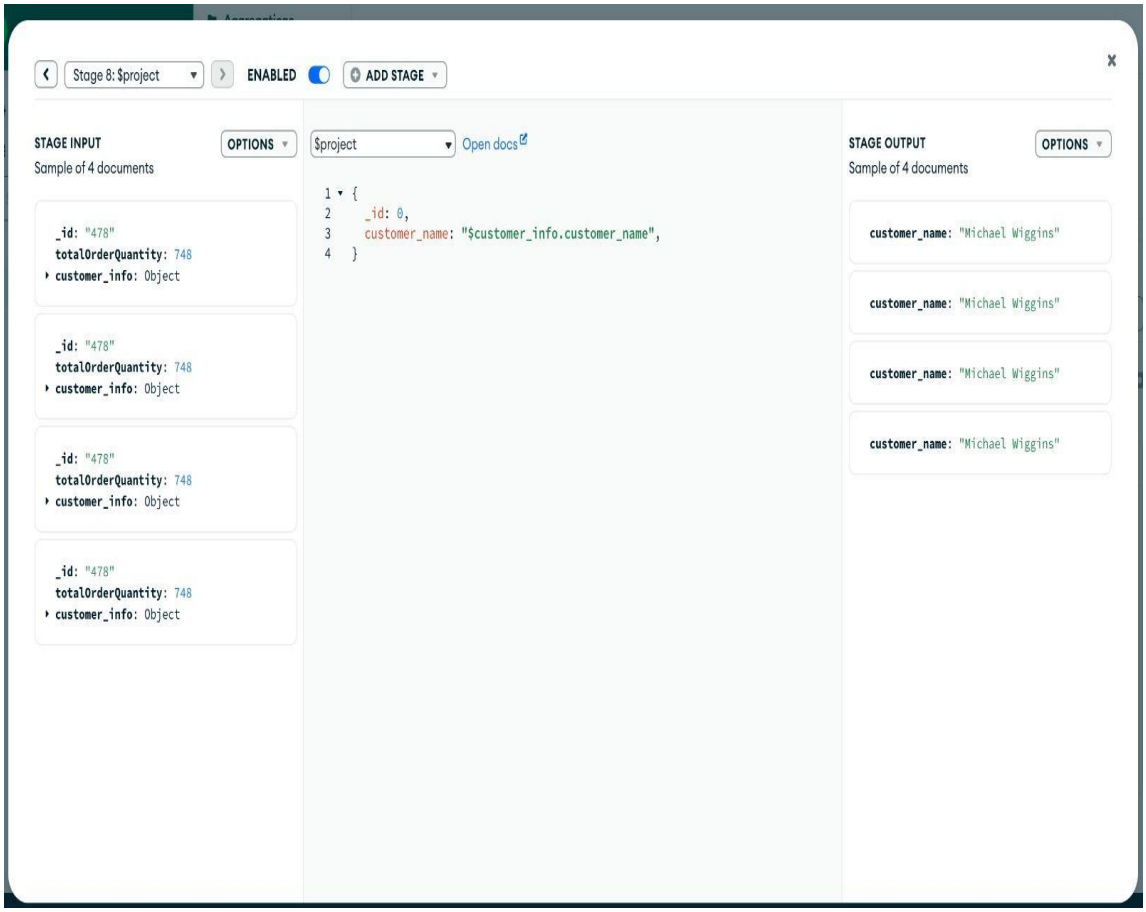
In the following 3 images, it returns the sum of all orders, order_date, all customer_ids, all shipping modes by order_date and sort by total orders in descending order returns the first 10 documents.





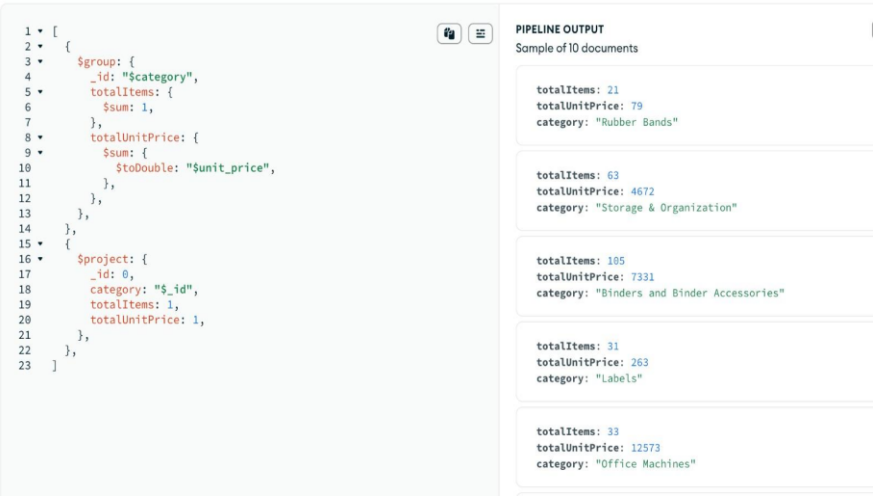
Query 4:
It returns the name of the customer who made the highest total order quantity.

```
[
 {
  $lookup: {
   from: "orders",
   localField: "customer_id",
   foreignField: "customer_id",
   as: "orders",
  },
 },
 {
  $unwind: "$orders",
 },
 {
  $group: {
   _id: "$customer_id",
   totalOrderQuantity: {
    $sum: {
     $toInt: "$orders.order_quantity",
  },
   },
  },
 },
 {
  $sort: {
   totalOrderQuantity: -1,
  },
 },
 {
  $limit: 1,
 },
 {
  $lookup: {
   from: "customers",
   localField: "_id",
   foreignField: "customer_id",
   as: "customer_info",
  },
 },
 {
  $unwind: "$customer_info",
 },
 {
  $project: {
   _id: 0,
customer_name:
    "$customer_info.customer_name",
  },
 },
]
```

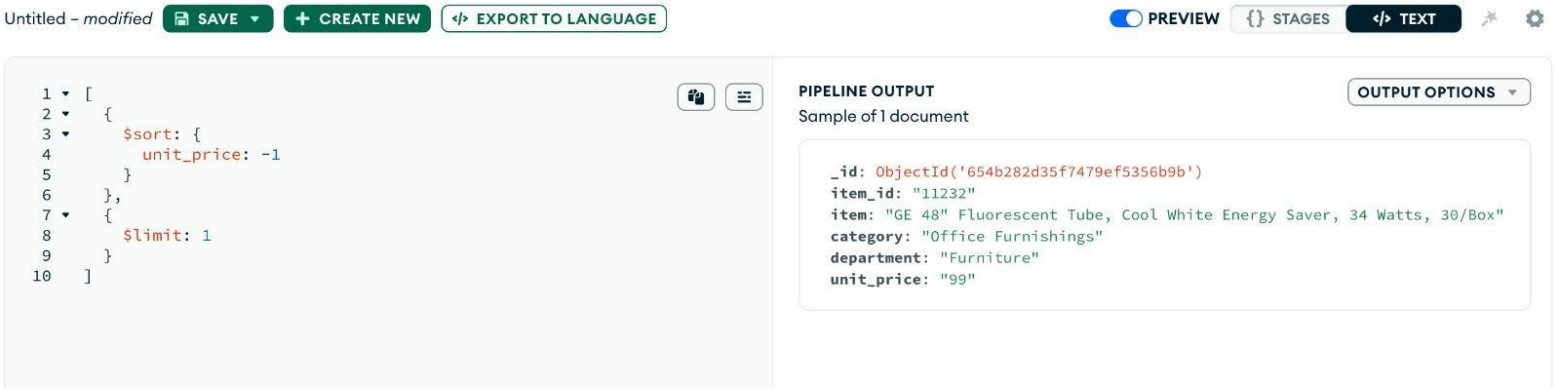


1. $lookup: performs a join between 2 collections
2. $unwind: outputs a new document for each element in specified array
3. $group: groups documents by specified expression (grouping on customer_id and doing sum)
4. $sort: reorders the documents by specified key
5. $limit: returns the number of rows mentioned (we are taking only one because we want customer with most orders quantity)
6. $lookup: based on row returned in above step it will return the record from customers collection with matching customer_id
7. $unwind: outputs a new document for each element in specified array (since there are multiple records with this customer_id)
8. $project: returns the output

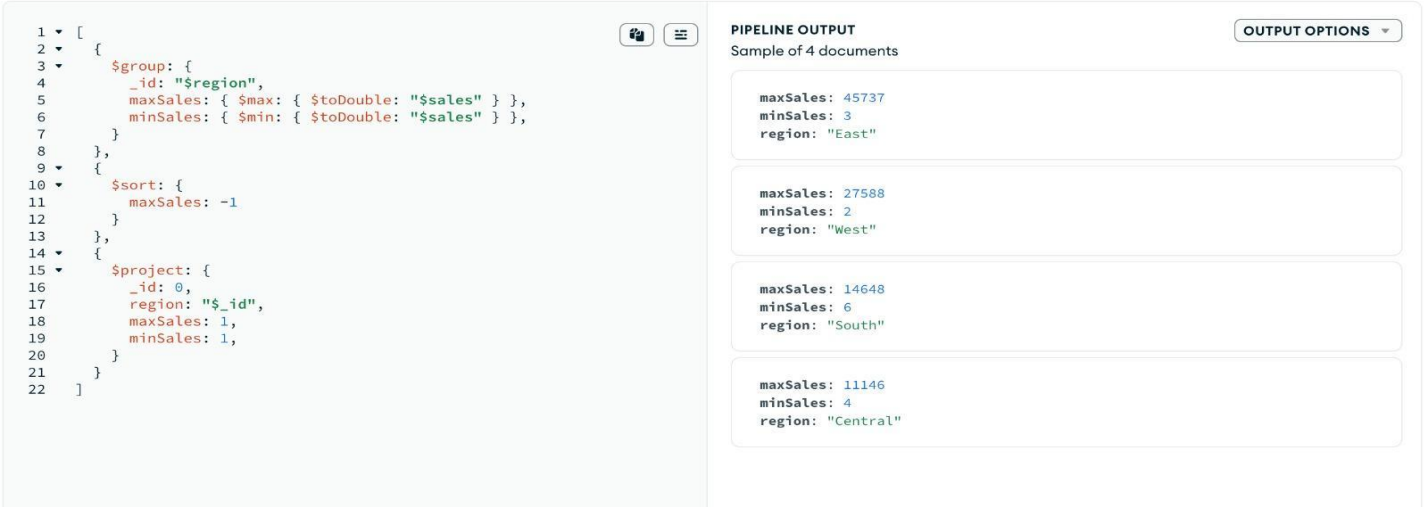
Query 5:
It returns Total Items and total Unit Price by Category



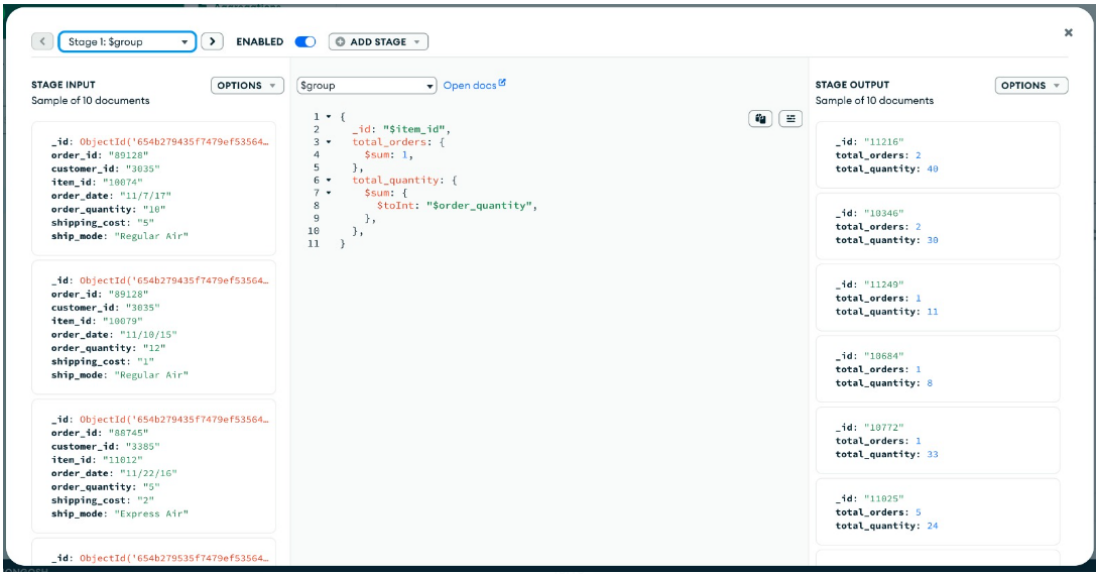
Query 6:
It returns the item details with the Highest Unit Price



Query 7:
It returns region of maximum and minimum sales by region.

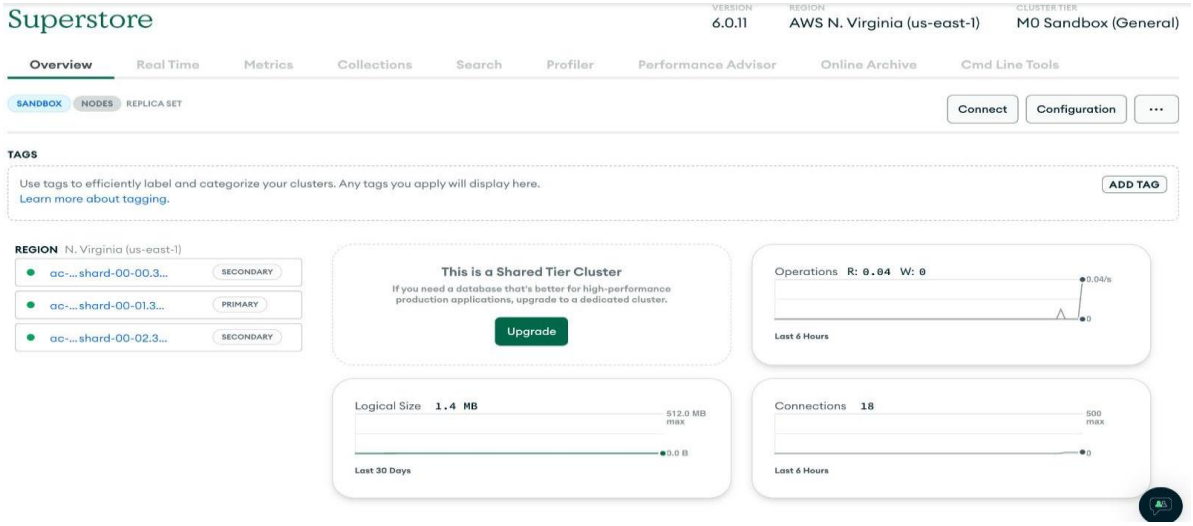


Query 8:
It returns item id, Total Orders and Order Quantity by Item.



6. Sharding and Cluster Computing:

MongoDB Sharding
MongoDB creates shards and by default it creates 3 shards where we can see that 1 is primary and 2 are secondary shards



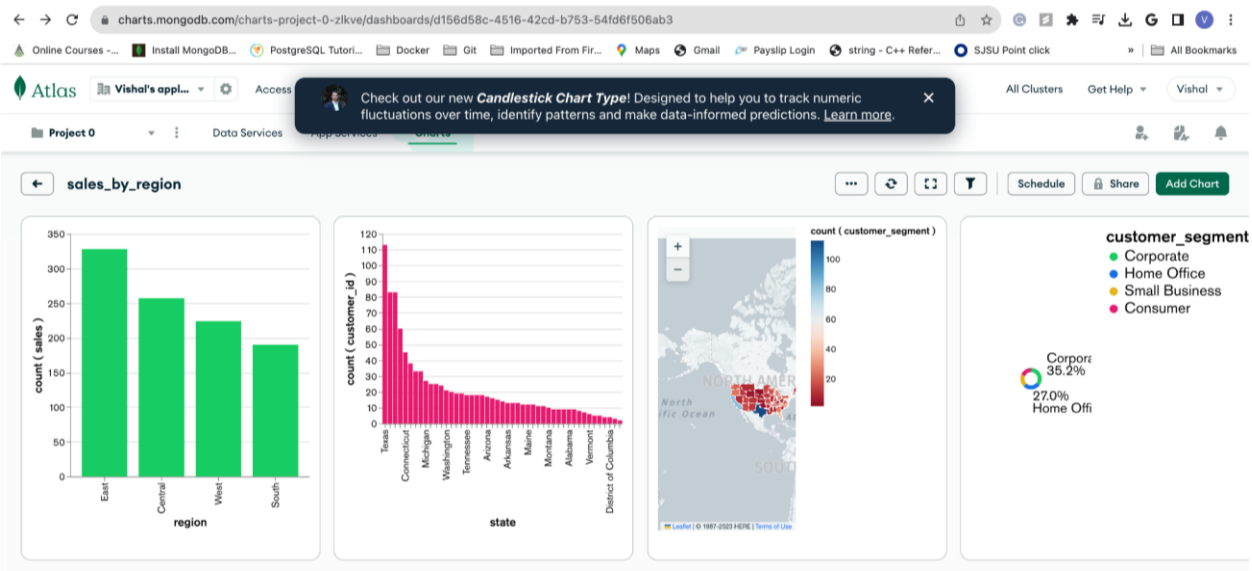
7. Visualizations

MongoDB Charts and Dashboards

1. Plot Dashboard:

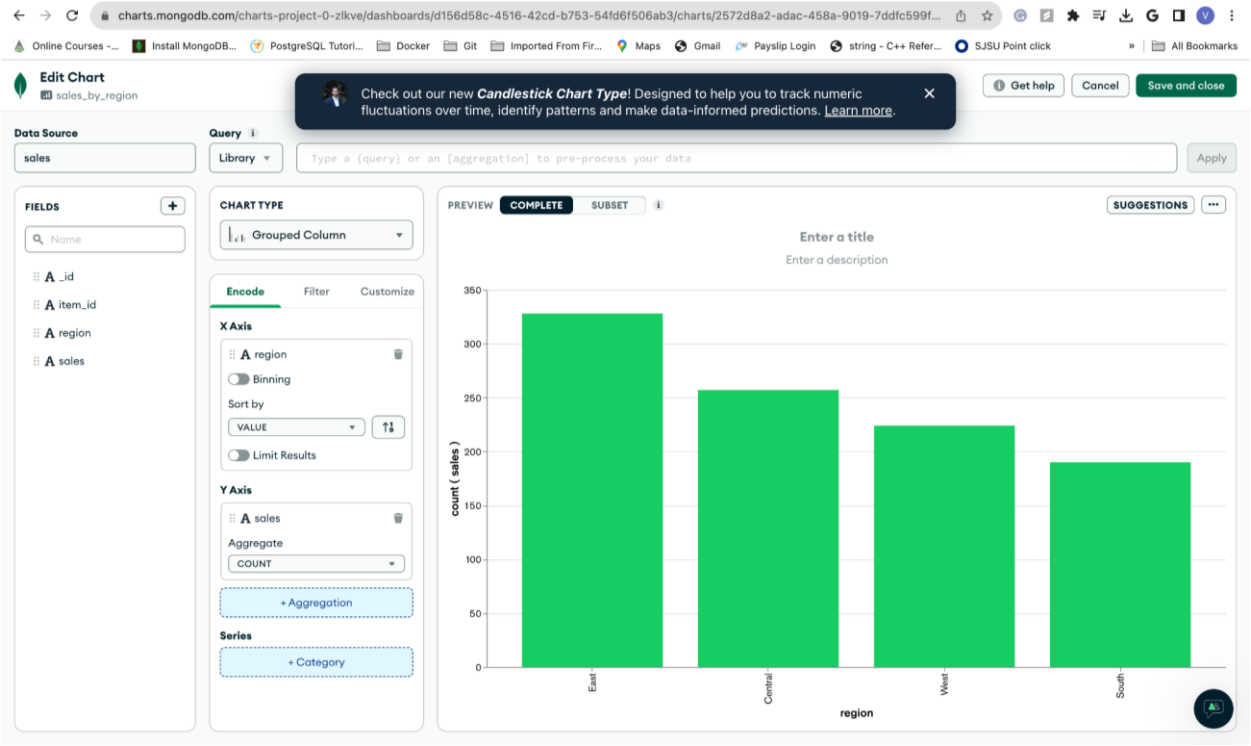
This is the dashboard created on MongoDB charts showing 4 different plots made for the sales data and they will be explained below.

Plots:



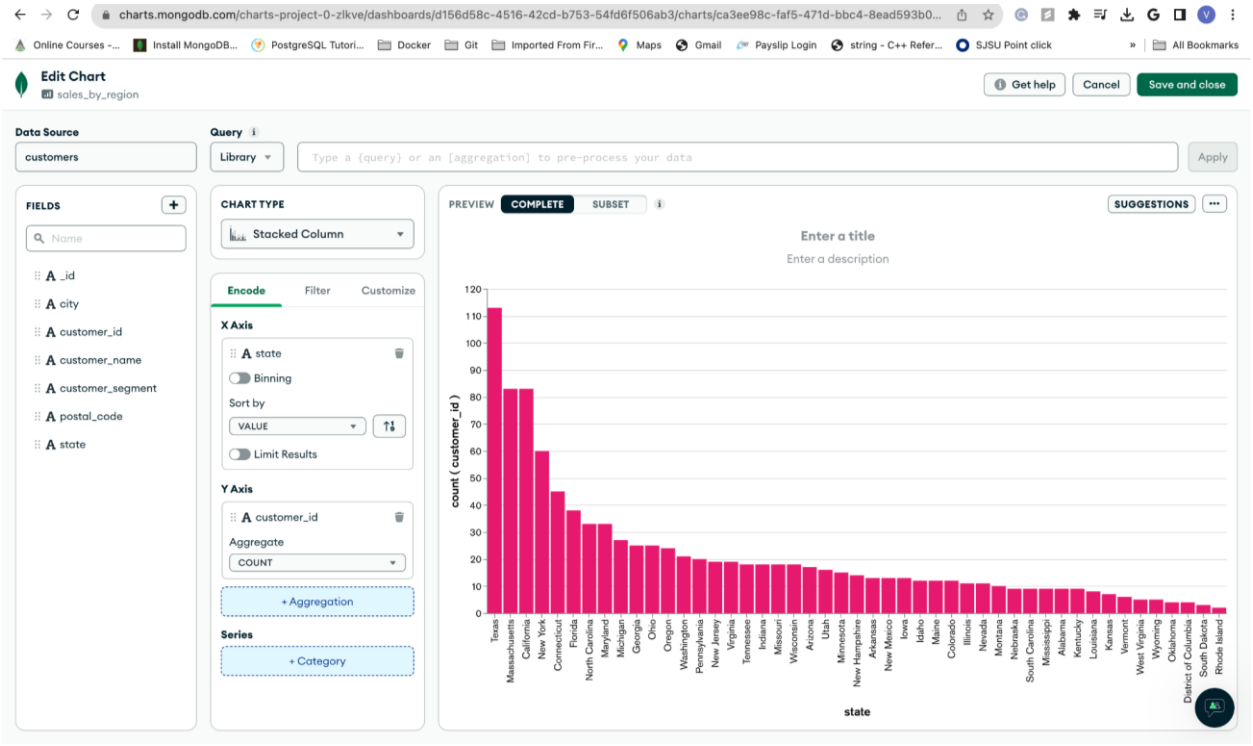
a. Sales table

From the sales table in the database, we have created this plot that shows regions on the x-axis and the count of customers on the y-axis giving us a detailed analysis of customer behavior telling the company which region uses their products the most.



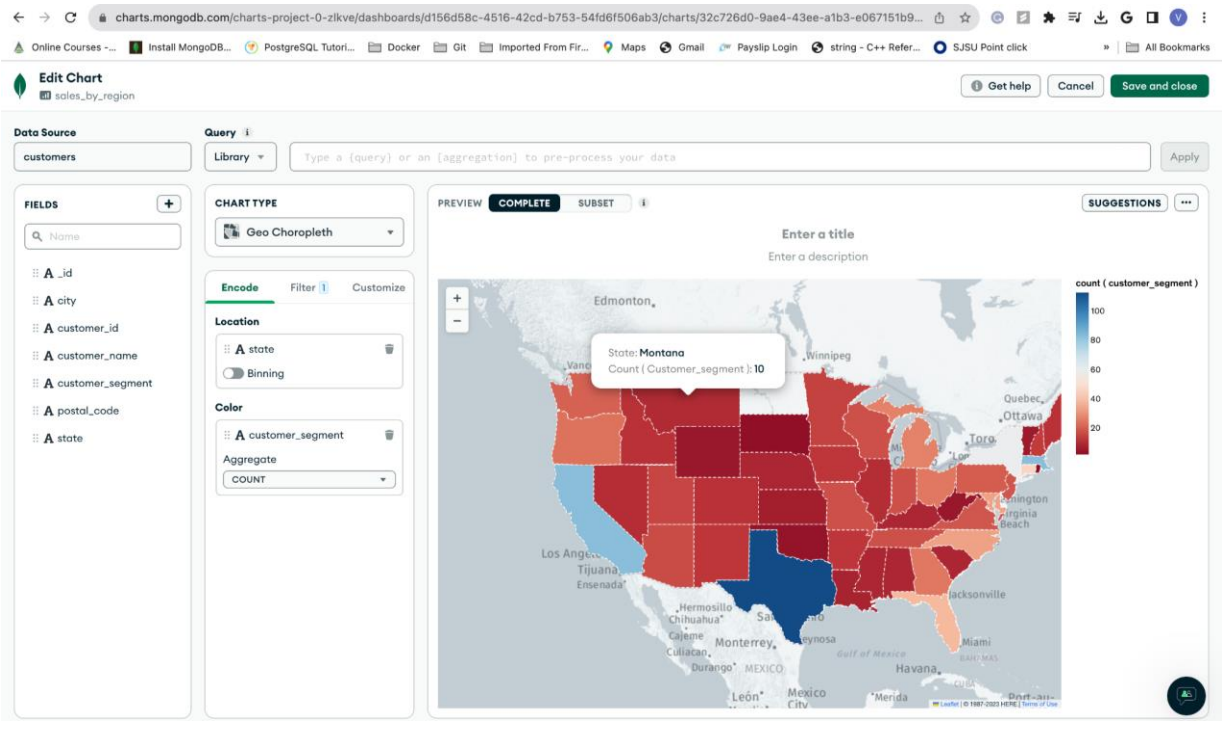
b. Customer table

This time we have chosen the customer table, and, on the plot, we have states where the customer belongs, on the x-axis, and on the y-axis, we have a count of customers and we can see the aggregates on the encoded part left of the plot. Again, this plot gives the company an analysis and inference that shows which state has most of its customers, and based on that the company can make some business decisions to maintain a well-balanced supply and demand process making the system more efficient than the existing one.



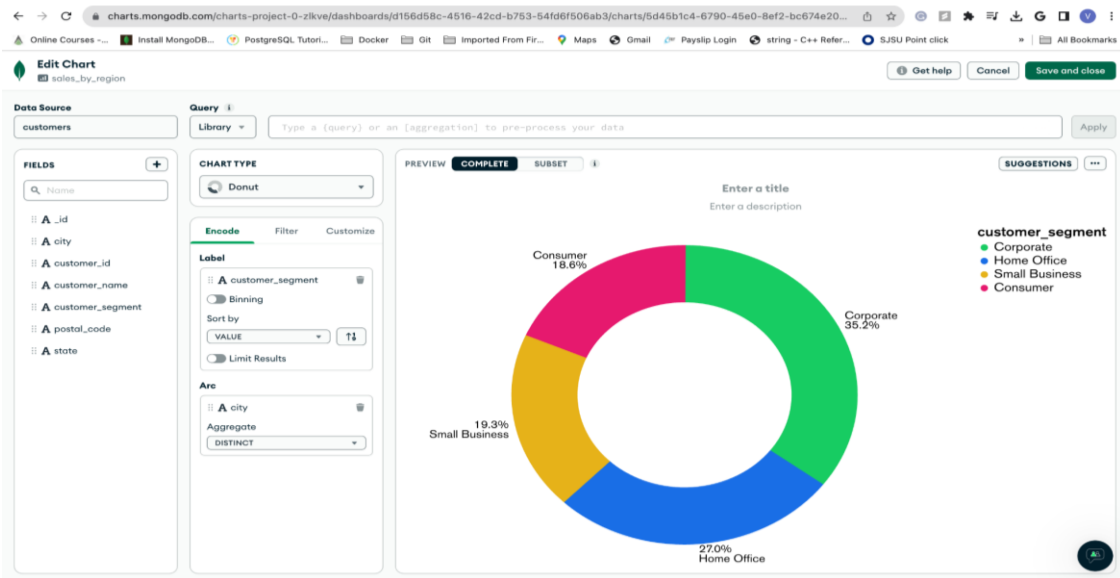
c. Customer table

Here, again, in the customer table, this time we are analyzing the customer segments (office, home office, business, home) count with the respective states so that we can analyze what states order most of what type of segment products and later the company can deploy those amounts of segments to their popularity in each state accordingly again solving the above issue. Here we have a choropleth plot where the color shows the count of segment, and the geolocations here have states in the USA giving us easier-to-understand data pictorially.



d. Customer table

For this plot, we have created a doughnut plot now providing an overall detail and count in percentages of what type of customer segments are in the dataset so far and their popularity for that specific company. Each color here represents a type of segment and their area covered is their weightage.



8. Comparing MongoDB and Neo4j (and RDBMS where applicable):

MongoDB	Neo4j
• Stores data in form of Key value pairs	• Graph Database
• Sharding Partitioning techniques are supported	• It is not compatible with partitioning techniques.
• Supports SQL through connector APIs	• Does not support SQL
• Mainly in JSON	• Has its own Cypher Query Language
• Supports MapReduce method	• Does not support MapReduce

9. Use of containerization, orchestration, microservice architecture

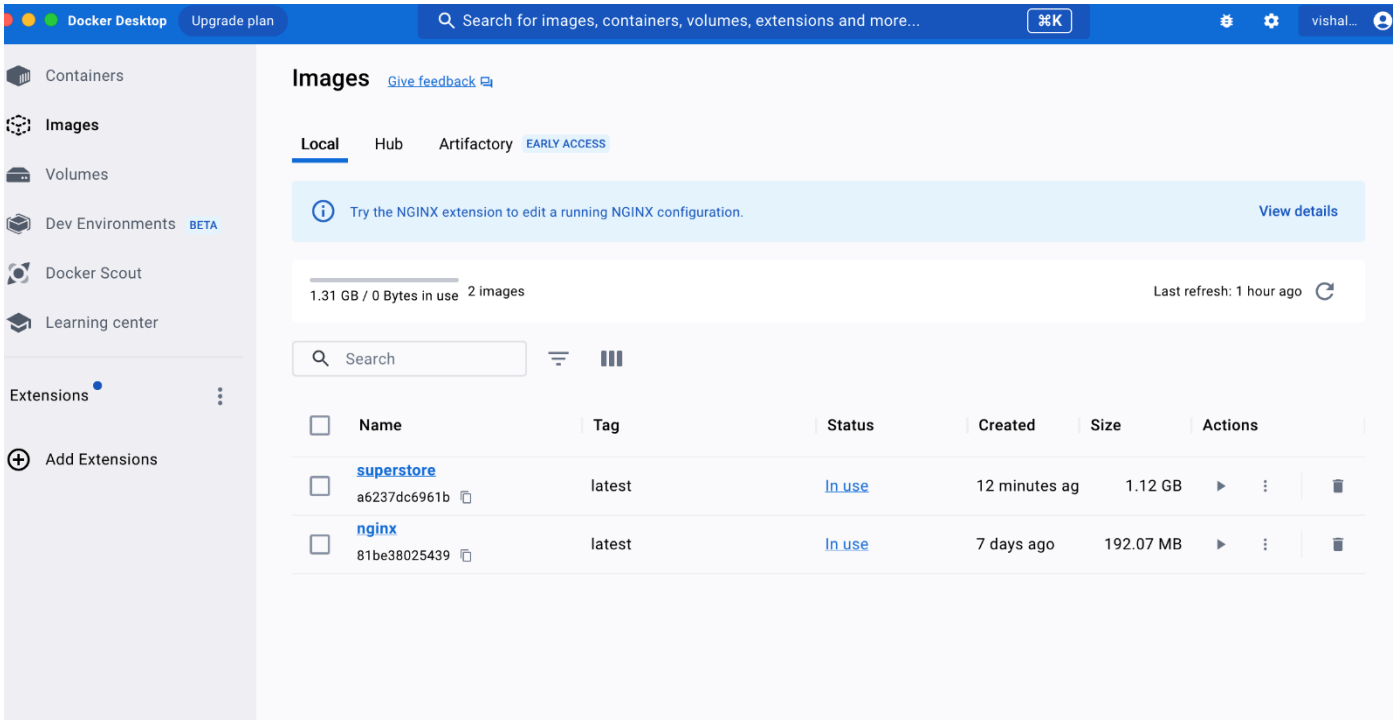
Docker image created and list of images shown

```
vishaltripathi@Vishals-MacBook-Pro Database for DA % docker build -t superstore .
[+] Building 0.8s (10/10) FINISHED                                                    docker:desktop-linu
 => [internal] load .dockerignore                                                                     0.0
 => => transferring context: 2B                                                                       0.0
 => [internal] load build definition from Dockerfile                                                  0.0
 => => transferring dockerfile: 649B                                                                  0.0
 => [internal] load metadata for docker.io/library/python:3                                           0.7
 => [auth] library/python:pull token for registry-1.docker.io                                         0.0
 => [1/4] FROM docker.io/library/python:3@sha256:7b8d65a924f596eb65306214f559253c468336bcae09fd575429774563460caf  0.0
 => [internal] load build context                                                                     0.0
 => => transferring context: 51.00kB                                                                  0.0
 => CACHED [2/4] WORKDIR /app                                                                         0.0
 => CACHED [3/4] COPY . /app                                                                          0.0
 => CACHED [4/4] RUN pip install pymongo                                                              0.0
 => exporting to image                                                                                0.0
 => => exporting layers                                                                               0.0
 => => writing image sha256:a6237dc6961b32e9c254efa60021cc65e4cf483312fce6124d2f4c9547deae23          0.0
 => => naming to docker.io/library/superstore                                                         0.0

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
vishaltripathi@Vishals-MacBook-Pro Database for DA % docker images -a
REPOSITORY   TAG       IMAGE ID       CREATED         SIZE
superstore   latest    a6237dc6961b   9 minutes ago   1.13GB
nginx        latest    81be38025439   7 days ago      192MB
```

Docker container created

```
vishaltripathi@Vishals-MacBook-Pro Database for DA % docker ps -a
CONTAINER ID   IMAGE         COMMAND                  CREATED          STATUS                      PORTS                    NAMES
da5a8bfa8e64   superstore    "python 015213224_HW…"   13 seconds ago   Exited (0) 12 seconds ago                            compassionate_heyrovsky
22f71d3501d2   superstore    "python 015213224_HW…"   10 minutes ago   Exited (0) 10 minutes ago                            fervent_ellis
21a56ad8e24f   nginx         "/docker-entrypoint.…"   52 minutes ago   Up 52 minutes               0.0.0.0:8080->80/tcp     nifty_mcnulty
```

Docker desktop showing images running in docker



10. Homework blog and video posted publicly, and other bells and whistles.

11. Use of NoSQL databases other than MongoDB and Neo4j:

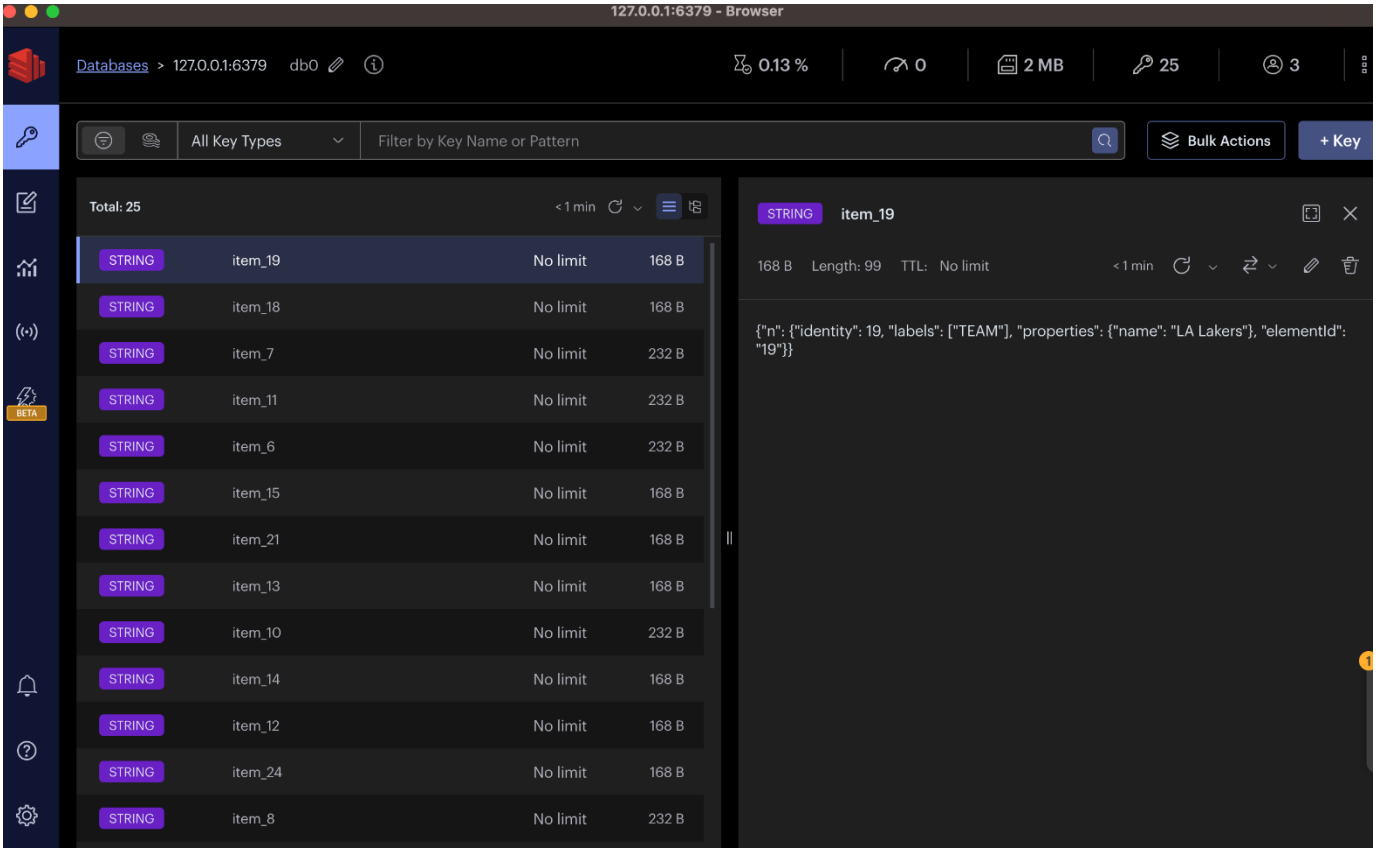
```
346:M 08 Nov 2023 16:33:02.215 * Increased maximum number of open files to 10032 (it wa
346:M 08 Nov 2023 16:33:02.215 * monotonic clock: POSIX clock_gettime
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 7.2.3 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in standalone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
      |    `-._   `._    /     _.-'    |     PID: 7846
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           https://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'

346:M 08 Nov 2023 16:33:02.216 # WARNING: The TCP backlog setting of 511 cannot be enfo
346:M 08 Nov 2023 16:33:02.216 * Server initialized
346:M 08 Nov 2023 16:33:02.217 * Loading RDB produced by version 7.2.3
346:M 08 Nov 2023 16:33:02.217 * RDB age 897 seconds
346:M 08 Nov 2023 16:33:02.217 * RDB memory usage when created 1.05 Mb
346:M 08 Nov 2023 16:33:02.217 * Done loading RDB, keys loaded: 0, keys expired: 0.
346:M 08 Nov 2023 16:33:02.217 * DB loaded from disk: 0.000 seconds
346:M 08 Nov 2023 16:33:02.217 * Ready to accept connections tcp
```

first image shows running redis server and 2nd shows the keys inserted in redis using redis-cli, here we run 2 commands

1. keys * (shows list of all the keys inserted in redis)
2. get key_name (returns result of the key given in get query)

```
vishaltripathi@Vishals-MacBook-Pro Database for DA % redis-cli
127.0.0.1:6379> keys *
 1) "item_19"
 2) "item_10"
 3) "item_15"
 4) "item_22"
 5) "item_3"
 6) "item_24"
 7) "item_8"
 8) "item_9"
 9) "item_12"
10) "item_23"
11) "item_16"
12) "item_6"
13) "item_5"
14) "item_13"
15) "item_18"
16) "item_7"
17) "item_11"
18) "item_14"
19) "item_21"
20) "item_17"
21) "item_1"
22) "item_20"
23) "item_4"
24) "item_0"
25) "item_2"
127.0.0.1:6379> get item_19
"{\"n\": {\"identity\": 19, \"labels\": [\"TEAM\"], \"properties\": {\"name\": \"LA Lakers\"}, \"elementId\": \"19\"}}"
127.0.0.1:6379>
```

Redis keys inserted in redisInsights GUI used for querying



Redis connection with python and importing NBA data into redis from json file

[24]:
```python
import json
import redis

# Connect to the Redis server
r = redis.StrictRedis(host='localhost', port=6379, db=0)

# Specify the JSON file path
json_file_path = 'records.json'

try:
    # Read the JSON file
    with open(json_file_path, 'r', encoding='utf-8-sig') as json_file:
        data = json.load(json_file)

        if isinstance(data, list):
            for index, value in enumerate(data):
                # Convert the value to JSON before storing
                json_value = json.dumps(value)
                # Use the index as the key or specify your own key logic
                key = f'item_{index}'
                r.set(key, json_value)
                print(f'Stored key: {key}')
        else:
            print('JSON data is not in the expected format (list of items).')

except Exception as e:
    print(f'Error: {e}')

# Close the Redis connection
r.connection_pool.disconnect()
```

```
Stored key: item_0
Stored key: item_1
Stored key: item_2
Stored key: item_3
```