proj4

December 19, 2023

[1]:
```
# Initialize Otter
import otter
grader = otter.Notebook("proj4.ipynb")
```

1 Project 4: Mongo

1.1 Due Date: Thursday 11/30, 5:00 PM

In this project, we will be investigating how different database systems handle semi-structured JSON data. In particular, we will be placing emphasis on the use of MongoDB: a database system that stores data in a construct known as documents. These documents are very similar to the JSON objects we've explored in lecture, with a few differences in representation and indexing that we will explore in the following questions.

In this project, we will be working with the **Yelp Academic Dataset** which contains a dataset of `businesses`, `reviews`, and `users`. Due to the limitations of JupyterHub and the Mongo instances we are working with, `reviews` and `users` are truncated to 7500 reviews and 1000 users. We will be using the full `businesses` dataset, however.

Throughout the course of this project, you should understand what Mongo can (and cannot) do with regards to its documents as a NoSQL datastore and compare and contrast this to other data representation formats such as the relational model.

1.2 Logistics & Scoring Breakdown

Please read the submission instructions carefully and double check that your submission is not throwing any errors. Please ensure that public tests pass upon submission. It is your responsibility to wait until the autograder finishes running. We will not be accepting regrade requests for submission issues.

Each coding question has **both public tests and hidden tests**. Roughly 50% of your coding grade will be made up of your score on the public tests released to you, while the remaining 50% will be made up of unreleased hidden tests. **Free-response questions (marked 'm' in the table below) are manually graded.**

This is an **individual project**. However, you're welcome to collaborate with any other student in the class as long as it's within the academic honesty guidelines.

Question	Points
1a	1
1b	1
1c	2
1d	1
1e	2
1f	1
2a	m: 2
2b	1
2c	1
2d	m: 2
3a	m: 1
3b	1
3c	1
3d	1
3e	1
3f	3
4a	1
4b	2
4c	2
4d	1
Total	28

Grand Total: 28 points (autograded: 23, manual: 5)

1.3 Loading Up Mongo

We will be using Pymongo, a Python wrapper for MongoDB, for this project. Every student should have access to their own MongoDB instance, running on the localhost of your Datahub server. After running the following cell, for the rest of the project, you can use the Python variables business, review, and user to access the corresponding collection.

To prevent bracket mismatches while creating your queries, it is recommended to turn on “Auto Close Brackets” via Settings in JupyterHub. Furthermore, since we are using Python dictionaries as our query filter, make sure to wrap all keys and values inside quotes.

[2]:

```python
import pickle
import pandas as pd
import pymongo
from pymongo import TEXT
import numpy as np

myclient = pymongo.MongoClient("mongodb://localhost")
mydb = myclient["yelp"]
business = mydb["business"]
review = mydb["review"]
user = mydb["user"]
```

1.4 Troubleshooting

PLEASE READ: Please avoid printing too much debugging query output—it may crash your Jupyter Hub if your file size becomes too large! It's recommended to use the `limit()` method and delete any debugging query cells if no longer needed as you go through the project.

You might run into issues on the project where you are certain your code works but the output is incorrect. This may be because your collections have been corrupted. Run the following cell and uncomment the specific collections you would like to drop if you would like to remake your collections from scratch. **Be sure to re-run the Load Datasets cells below if you drop your collections so you aren't working with empty collections!**

[3]:
```
# UNCOMMENT AND RUN THIS CELL IF YOU WOULD LIKE TO REMAKE YOUR COLLECTIONS FROM␣
 ↪SCRATCH.
# IF YOU DROP ANY COLLECTIONS, RE-RUN THE NEXT TWO CELLS TO LOAD IN THE DATA.

# review.drop()
# business.drop()
# user.drop()
```

1.5 Load Datasets

The following 2 cells will load the JSON datasets into the appropriate Mongo collections. You will only need to run them once unless you drop the collections above. The second cell **may take a couple of minutes to run** if you are running it for the first time or are running it after you dropped the collections.

[4]:
```
import zipfile
import os.path

if not os.path.isfile('data/yelp_academic_dataset_review.json'):
    with zipfile.ZipFile('data/yelp_academic_dataset_review.json.zip', 'r') as␣
 ↪zip_ref:
        zip_ref.extractall('data')

if not os.path.isfile('data/yelp_academic_dataset_user.json'):
    with zipfile.ZipFile('data/yelp_academic_dataset_user.json.zip', 'r') as␣
 ↪zip_ref:
        zip_ref.extractall('data')

if not os.path.isfile('data/yelp_academic_dataset_business.json'):
    with zipfile.ZipFile('data/yelp_academic_dataset_business.json.zip', 'r')␣
 ↪as zip_ref:
        zip_ref.extractall('data')
```

[5]:
```
# THIS CELL MAY TAKE AT MOST 5 MINUTES. BUT HOPEFULLY YOU WILL ONLY NEED TO RUN␣
 ↪IT ONCE.
import json
```

```
if business.count_documents({}) == 0:
    print("Loading business collection...")
    with open('data/yelp_academic_dataset_business.json', encoding='utf-8') as␣
 ↪f:
        for line in f:
            business.insert_one(json.loads(line))

if review.count_documents({}) == 0:
    print("Loading review collection...")
    with open('data/yelp_academic_dataset_review.json', encoding='utf-8') as f:
        for line in f:
            review.insert_one(json.loads(line))

if user.count_documents({}) == 0:
    print("Loading user collection...")
    with open('data/yelp_academic_dataset_user.json', encoding='utf-8') as f:
        for line in f:
            user.insert_one(json.loads(line))
```

Let's take a quick look at our collections. For the command below, replace `user` with `review` or `business` to count the number of documents in each collection.

[6]:

```
user.count_documents({})
```

[6]:

```
1000
```

Now let's inspect our collections. Replace `business` with `review` and `user` to see the first document in each collection.

[7]:

```
business.find_one()
```

[7]:

```
{'_id': ObjectId('656657f8ca20f68f10dacfc5'),
 'business_id': '6iYb2HFDywm3zjuRg0shjw',
 'name': 'Oskar Blues Taproom',
 'address': '921 Pearl St',
 'city': 'Boulder',
 'state': 'CO',
 'postal_code': '80302',
 'latitude': 40.0175444,
 'longitude': -105.2833481,
 'stars': 4.0,
 'review_count': 86,
 'is_open': 1,
 'attributes': {'RestaurantsTableService': 'True',
  'WiFi': "u'free'",
  'BikeParking': 'True',
  'BusinessParking': "{'garage': False, 'street': True, 'validated': False,
'lot': False, 'valet': False}",
  'BusinessAcceptsCreditCards': 'True',
  'RestaurantsReservations': 'False',
  'WheelchairAccessible': 'True',
  'Caters': 'True',
  'OutdoorSeating': 'True',
  'RestaurantsGoodForGroups': 'True',
  'HappyHour': 'True',
  'BusinessAcceptsBitcoin': 'False',
  'RestaurantsPriceRange2': '2',
  'Ambience': "{'touristy': False, 'hipster': False, 'romantic': False, 'divey':
False, 'intimate': False, 'trendy': False, 'upscale': False, 'classy': False,
'casual': True}",
  'HasTV': 'True',
  'Alcohol': "'beer_and_wine'",
  'GoodForMeal': "{'dessert': False, 'latenight': False, 'lunch': False,
'dinner': False, 'brunch': False, 'breakfast': False}",
  'DogsAllowed': 'False',
  'RestaurantsTakeOut': 'True',
  'NoiseLevel': "u'average'",
  'RestaurantsAttire': "'casual'",
  'RestaurantsDelivery': 'None'},
 'categories': 'Gastropubs, Food, Beer Gardens, Restaurants, Bars, American
(Traditional), Beer Bar, Nightlife, Breweries',
 'hours': {'Monday': '11:0-23:0',
  'Tuesday': '11:0-23:0',
  'Wednesday': '11:0-23:0',
  'Thursday': '11:0-23:0',
  'Friday': '11:0-23:0',
  'Saturday': '11:0-23:0',
  'Sunday': '11:0-23:0'}}
```

If you see a document containing a business named `Oskar Blues Taproom` when you run the command above, it means that our JSON data has successfully been imported into the collection! Now we can get started with exploring Mongo in a bit more detail.

1.6 Connect to the grader

Run the following cell for grading purposes.

[8]:
```python
# Just run the following cell, no further action is needed.
from data101_utils import GradingUtil
grading_util = GradingUtil("proj4")
grading_util.prepare_autograder()
```

[9]:
```python
# Do not delete/edit this cell
import pickle
import pandas as pd
```

1.7 Question 1: Basic MQL

1.7.1 Question 1a

In lecture, we discussed how one could find specific attributes from a JSON object using dot (.) notation.

- While you can still use the dot notation in queries, PyMongo represents documents returned from Mongo queries using Python dictionaries, making it convenient to manipulate JSON using a mix of Mongo queries and array indexing. Specifically, given the result of a retrieval `find` query, you can look up the third document by indexing with `[2]`. Note, since we are using Python dictionaries, we will be using 0-based indexing. Then, given this document, you can look up the field `'amount'` by appending `['amount']` etc., adding multiple square brackets as needed to "walk down" the JSON tree representation via `collection.find(...)[2]['amount']`. This will return the 'amount' field from the 3rd document returned from the query. This combination of query and indexing will be useful in obtaining the necessary information you need for this question.
- In order to get a visual output of the query results, you will need to wrap `collection.find(...)` inside `list()`, e.g. `list(collection.find(...))`. This is because `collection.find(...)` returns a **Cursor** object, which is an iterator. **An important consequence** is that if we set `result = collection.find(...)`, then calling `list(result)` for the first time will get you the expected list of documents in the query result, but calling `list(result)` for a second time will give you an empty list! So wrapping `collection.find(...)` directly inside `list()` would avoid this issue. With that in mind, you may not *always* need to obtain a visual output of the results.
- Be aware of the distinction of when you are querying with Mongo versus Python-based array indexing into your Mongo query results (i.e. you are wrapping your query inside `list()` and *then* indexing into that list.)
- **As a reminder, since we are using Python dictionaries as our query filter, make sure to wrap all keys and values inside quotes.**

As a warmup to get you familiarized with PyMongo syntax, find the **Tuesday hours** for the restaurant named **Legal Sea Foods** at **100 Huntington Ave** in **Boston**. Be careful—there are many Legal Sea Foods in Boston!

```
[10]: result_1a = business.find({"name" : "Legal Sea Foods", "address" : "100␣
        ↪Huntington Ave", "city" : "Boston"})[0]["hours"]["Tuesday"]
      result_1a
```

```
[10]: '12:0-21:0'
```

```
[11]: # Do not delete/edit this cell!
      # You must run this cell before running the autograder.
      grading_util.save_results("result_1a", result_1a);
```

```
[12]: grader.check("q1a")
```

```
[12]: q1a results: All test cases passed!
```

1.7.2 Question 1b

Now let's get some practice with aggregation and filtering. Our goal is to write a query that computes the average star rating for all businesses in Colorado with 30 reviews or greater. However, this won't be as easy as setting the state to CO! If we inspect this dataset more closely, we will notice that some cities are not matched up with the right states. As an example, run the query below.

[13]:
```
list(business.find({"state": "CA"}).limit(3))
```

[13]:
```
[{'_id': ObjectId('656657f8ca20f68f10dad6e0'),
  'business_id': 'SNCRnaSy6E5fHgQuoCmmbQ',
  'name': 'Katia Photography',
  'address': '',
  'city': 'Portland',
  'state': 'CA',
  'postal_code': '97007',
  'latitude': 45.4501529,
  'longitude': -122.8849111,
  'stars': 5.0,
  'review_count': 11,
  'is_open': 1,
  'attributes': {'BusinessAcceptsBitcoin': 'False',
   'BusinessAcceptsCreditCards': 'True',
   'WiFi': "u'no'"},
  'categories': 'Shopping, Clothing Rental, Event Planning & Services, Fashion,
Event Photography, Photographers, Session Photography',
  'hours': {'Monday': '8:0-22:0',
   'Tuesday': '8:0-22:0',
   'Wednesday': '8:0-22:0',
   'Thursday': '8:0-22:0',
   'Friday': '8:0-22:0',
   'Saturday': '8:0-22:0',
   'Sunday': '8:0-22:0'}},
 {'_id': ObjectId('65665801ca20f68f10db474c'),
  'business_id': 'cjwnQMQOGOYgB5uNmiYWLA',
  'name': 'Verizon Authorized Retailer - GoWireless',
  'address': '4655 SW Griffith Dr, Ste 125',
  'city': 'Beaverton',
  'state': 'CA',
  'postal_code': '97005',
  'latitude': 45.486312,
  'longitude': -122.796487,
  'stars': 2.5,
  'review_count': 9,
  'is_open': 0,
  'attributes': {'BusinessAcceptsCreditCards': 'True',
   'BusinessParking': 'None',
   'ByAppointmentOnly': 'False',
   'BikeParking': 'True'},
  'categories': 'Mobile Phones, Telecommunications, Shopping, Home Services, IT
Services & Computer Repair, Internet Service Providers, Local Services,
Professional Services, Mobile Phone Accessories, Electronics',
  'hours': {'Monday': '0:0-0:0',
   'Tuesday': '10:0-18:0',
   'Wednesday': '10:0-18:0',
   'Thursday': '10:0-18:0',
   'Friday': '10:0-18:0',
   'Saturday': '10:0-18:0',
   'Sunday': '11:0-18:0'}},
 {'_id': ObjectId('65665806ca20f68f10db8a33'),
  'business_id': 'l92RJMHpxZgJl7FQuqpW6w',
  'name': 'Here We Grow',
  'address': '',
  'city': 'Atlanta',
  'state': 'CA',
  'postal_code': '30345',
  'latitude': 33.8484195,
  'longitude': -84.2858121,
  'stars': 5.0,
  'review_count': 6,
  'is_open': 1,
  'attributes': {'GoodForKids': 'True'},
  'categories': 'Childbirth Education, Specialty Schools, Preschools, Adult
Education, Parenting Classes, Education, Active Life, Kids Activities',
  'hours': {'Tuesday': '9:0-17:0',
   'Thursday': '9:0-17:0',
   'Saturday': '9:0-13:0',
   'Sunday': '11:0-15:0'}}]
```

Notice how cities like Portland and Atlanta, and Orlando are classified as California cities! However, the latitude and longitude is generally correct. The latitude of Colorado is between 37 and 41 **inclusive** and the longitude is between -109 and -102 **inclusive**. Now, use this to **find the average star rating** of all businesses in this range with **30 or more reviews**.

Recall that in SQL, we would use a GROUP BY with the AVG aggregation function. In Mongo, we use an aggregation pipeline (documentation here), comprised of multiple stages (e.g., `$match` followed by `$group`). Each stage transforms the documents in some way. Pipeline stages do not need to produce one output document for every input document. For example, some stages may generate new documents or filter out documents.

Hints: - As in the previous question, you may find it helpful to use the PyMongo array notation to extract the pertinent information/document once you have composed the right Mongo aggregation query. You are required to wrap `collection.aggregate(...)` inside `list()`, e.g. `list(collection.aggregate(...))` before indexing / visualizing the output. Similar to `collection.find(...)`, `collection.aggregate(...)` also returns a **Cursor** object (which is an iterator).

- You can set multiple conditions for a given field within the same object, e.g. `{"$gte": 0, "$lte": 10}`. This is the recommended approach, or else you may need to worry about the ordering between the conditions.

[14]:
```
result_1b = list(business.aggregate([
    {
        "$match": {
            "latitude": {"$gte": 37, "$lte": 41},
            "longitude": {"$gte": -109, "$lte": -102},
            "review_count": {"$gte": 30}
        }
    },
    {
        "$group": {
            "_id": None,
            "average_stars": {"$avg": "$stars"}
        }
    }
]))[0]['average_stars']
result_1b
```

[14]:
```
3.6735412474849096
```

[15]:
```
# Do not delete/edit this cell!
# You must run this cell before running the autograder.
grading_util.save_results("result_1b", result_1b);
```

[16]:
```
grader.check("q1b")
```

[16]:
```
q1b results: All test cases passed!
```

1.7.3 Question 1c

In this question, we will explore aggregation and grouping further. We will also make use of the `$project` operator which allows us to output documents with certain fields of our choosing.

For this question, we would like to create an aggregation pipeline to find the town in each state with the highest average number of stars. **We will only consider towns with greater than or equal to 5 reviews in total across all the restaurants in that town so that the average is meaningful.** Your final output should contain exactly two fields: - `averageStars` which contains the average number of stars for the corresponding town. - `city_state` which is the name of the town with the highest value of average stars in the state concatenated with a comma followed by the state initials

To ensure your output is consistent with the autograder, **sort in descending order by averageStars and break ties by sorting second on city_state in alphabetical (ascending) order.**

As a concrete example, imagine that Berkeley and Austin have the highest average stars in California and Texas respectively (and both have more than or equal to 5 total reviews in this *truncated* dataset). If Berkeley and Austin both have an average star rating of 5.0, your final output should be:

```
{'averageStars': 5.0, 'city_state': 'Austin, TX'}
{'averageStars': 5.0, 'city_state': 'Berkeley, CA'}
```

Note: You will provide a pipeline to `business.aggregate(...)` as your solution. Save your pipeline to `q1c_pipeline`.

Hint: You may find the `concat` operator helpful (documentation here).

[17]:

```
#i think something is missing? COME BACK
q1c_pipeline = [
    {
        "$match": {
            "review_count": {"$gte": 5}
        }
    },
    {
        "$group": {
            "_id": {"city": "$city", "state": "$state"},
            "averageStars": {"$avg": "$stars"}
        }
    },
    {
        "$group": {
            "_id": "$_id.state",
            "city": {"$first": "$_id.city"},
            "averageStars": {"$first": "$averageStars"}
        }
    },
    {
        "$project": {
            "averageStars": 1,
            "city_state": {"$concat": ["$city", ", ", "$_id"]}
        }
    },
]

result_1c = list(business.aggregate(q1c_pipeline))
result_1c
```

```
[17]: [{'_id': 'FL', 'averageStars': 4.5, 'city_state': 'Loughman, FL'},
       {'_id': 'ME', 'averageStars': 3.5, 'city_state': 'North Reading, ME'},
       {'_id': 'AZ', 'averageStars': 3.5, 'city_state': 'Lake Oswego, AZ'},
       {'_id': 'MN', 'averageStars': 3.5, 'city_state': 'Austin, MN'},
       {'_id': 'CA', 'averageStars': 4.5, 'city_state': 'New Albany, CA'},
       {'_id': 'KS', 'averageStars': 2.0, 'city_state': 'Ellsworth, KS'},
       {'_id': 'WY', 'averageStars': 1.5, 'city_state': 'Sheridan, WY'},
       {'_id': 'BC', 'averageStars': 3.0, 'city_state': 'NEW WESTMINSTER, BC'},
       {'_id': 'NY', 'averageStars': 5.0, 'city_state': 'Spring Valley, NY'},
       {'_id': 'ABE', 'averageStars': 4.5, 'city_state': 'Vancouver, ABE'},
       {'_id': 'VA', 'averageStars': 2.0, 'city_state': 'Richmond, VA'},
       {'_id': 'WA', 'averageStars': 5.0, 'city_state': 'Vancover, WA'},
       {'_id': 'NH', 'averageStars': 1.0, 'city_state': 'Hollis, NH'},
       {'_id': 'NM', 'averageStars': 3.5, 'city_state': 'Rio Rancho, NM'},
       {'_id': 'OH', 'averageStars': 2.0, 'city_state': 'West Columbus, OH'},
       {'_id': 'OR',
        'averageStars': 3.56198347107438,
        'city_state': 'Clackamas, OR'},
       {'_id': 'NC', 'averageStars': 2.5, 'city_state': 'Sanford, NC'},
       {'_id': 'ON', 'averageStars': 2.5, 'city_state': 'Burnaby, ON'},
       {'_id': 'MA', 'averageStars': 4.0, 'city_state': 'West Concord, MA'},
       {'_id': 'DC', 'averageStars': 5.0, 'city_state': 'Vancouver, DC'},
       {'_id': 'KY', 'averageStars': 2.0, 'city_state': 'Liberty, KY'},
       {'_id': 'IL', 'averageStars': 4.0, 'city_state': 'Franklin Park, IL'},
       {'_id': 'HI', 'averageStars': 4.5, 'city_state': 'Portland, HI'},
       {'_id': 'TX', 'averageStars': 3.3280632411067192, 'city_state': 'Kyle, TX'},
       {'_id': 'GA',
        'averageStars': 3.0053475935828877,
        'city_state': 'College Park, GA'},
       {'_id': 'AL', 'averageStars': 2.5, 'city_state': 'Austin, AL'},
       {'_id': 'DE', 'averageStars': 4.5, 'city_state': 'Powell, DE'},
       {'_id': 'WI', 'averageStars': 3.5, 'city_state': 'Glendale, WI'},
       {'_id': 'CO', 'averageStars': 2.0, 'city_state': 'Aurora, CO'},
       {'_id': 'OK', 'averageStars': 2.5, 'city_state': 'Oklahoma City, OK'},
       {'_id': 'MI', 'averageStars': 3.5, 'city_state': 'Jackson, MI'}]
```

```
[18]: # Do not delete/edit this cell!
      # You must run this cell before running the autograder.
      grading_util.save_results("result_1c", list(business.aggregate(q1c_pipeline)));
```

```
[19]: grader.check("q1c")
```

```
[19]: q1c results: All test cases passed!
```

1.7.4 Question 1d

In class, we've described structured (rectangular) data as well as semi-structured data. We haven't quite covered unstructured data—this is basically free-form text. Often, in semi-structured JSON you may have unstructured text data embedded within, such as the text field in the review collection.

MongoDB allows us to build a so-called **text index** to retrieve the relevant document based on keywords found in text in a predefined field. This index converts our free-form text into a structure that allows us to easily look up documents by its contents. To leverage this text search capability, we build a text index on the `text` field in the `review` collection. This has been done for you.

We will then use this text index to do basic sentiment analysis and find all the restaurants we should avoid! Using the text index given, write a query to find all the reviews with "disgusting", "horrible", "horrid", "gross", "bad", or "hate". To use the text index, use the keywords `$text` and `$search` as detailed here.

Fill in your query into `result_1d` to count how many reviews contain any of these 6 words.

Hint: In general, you can count the number of documents returned by a `find` query result via `len(list(collection.find(...)))` or more simply `collection.count_documents(...)`. To count the number of documents returned by an `aggregate` query result, the best way is to directly use `len(list(collection.aggregate(...)))`.

[20]:

```
# We create a text index here
if 'text_text' not in review.index_information():
    review.create_index([('text', TEXT)])

result_1d = review.count_documents({
    "$text": {
        "$search": "disgusting horrible horrid gross bad hate"
    }
})
result_1d
```

[20]:

```
728
```

[21]:

```
# Do not delete/edit this cell!
# You must run this cell before running the autograder.
grading_util.save_results("result_1d", result_1d);
```

[22]:

```
grader.check("q1d")
```

[22]:

```
q1d results: All test cases passed!
```

1.7.5 Question 1e

Now let's learn Mongo updates, deletions, and creation. Create a new collection called `review_boolean` which is the exact same as `reviews` EXCEPT there is a new field called `to_avoid` which is the string “true” if the review `text` contains the words “disgusting”, “horrid”, “horrible”, “gross”, “bad”, or “hate” and the string “false” if not.

This is a tricky task! We have not discussed creation, updates, or insertions in great detail during lecture but luckily, Mongo uses a similar approach to SQL.

Insertions: In order to insert into a document, you may use the functions review_boolean.insert_one(…) or review_boolean.insert_many(…). These functions take in a document or a list of documents and inserts them into the collection.

Updates: In order to update a document, you may use the functions review_boolean.update_one(…) or review_boolean.update_many(…). These functions take in two parameters. The first specifies which documents should be modified. If the first parameter is `{}`, this indicates that all documents should be updated. However, you can put a more specific filter here if you would like. The second parameter specifies what you would like to update your field to (the $set operator may come in handy here). Recall that in our SQL model, updates are performed as `UPDATE ... SET ... WHERE ....` In our case, the first ellipsis corresponds to `review_boolean`, the second ellipsis corresponds to the second parameter of `update_*` where `*` can be `one` or `many`, and the third ellipsis corresponds to the first parameter of `update_*`.

Creation: We handle creation of the collection for you. But in Pymongo, creation of a collection is as simple as writing `variable_name = db[collection_name]` where db is the the Pymongo database object variable you have already created.

Some additional reminders and hints: - The empty collection `review_boolean` has already been created for you and is stored in the variable of the same name. - A text index has been created for you. You can use a similar search approach as the last question. - We want to start by inserting the documents from the `review` collection into the `review_boolean` collection. - Don’t forget that in order to pass the hidden tests, the `to_avoid` field must exist for every document in `review_boolean`! The $exists operator may be helpful.

[23]:

```python
review_boolean = mydb["review_boolean"]
review_boolean.drop()

# We create a text index here
if 'text_text' not in review_boolean.index_information():
    review_boolean.create_index([('text', TEXT)])

review_boolean.insert_many(list(review.find({})))

review_boolean.update_many(
    {"$text": {"$search": "disgusting horrid horrible gross bad hate"}},
    {"$set": {"to_avoid": "true"}}
)

review_boolean.update_many(
    {"to_avoid": {"$exists": False}},
    {"$set": {"to_avoid": "false"}}
)
```

[23]:
```
<pymongo.results.UpdateResult at 0x7fcbed0b2110>
```

[24]:
```python
review_boolean = mydb["review_boolean"]
review_boolean.find_one()
```

[24]:
```
{'_id': ObjectId('6566582aca20f68f10dd430e'),
 'review_id': 'lWC-xP3rd6obsecCYsGZRg',
 'user_id': 'ak0TdVmGKo4pwqdJSTLwWw',
 'business_id': 'buF9druCkbuXLX526sGELQ',
 'stars': 4.0,
 'useful': 3,
 'funny': 1,
 'cool': 1,
 'text': "Apparently Prides Osteria had a rough summer as evidenced by the
almost empty dining room at 6:30 on a Friday night. However new blood in the
kitchen seems to have revitalized the food from other customers recent visits.
Waitstaff was warm but unobtrusive. By 8 pm or so when we left the bar was full
and the dining room was much more lively than it had been. Perhaps Beverly
residents prefer a later seating. \n\nAfter reading the mixed reviews of late I
was a little tentative over our choice but luckily there was nothing to worry
about in the food department. We started with the fried dough, burrata and
prosciutto which were all lovely. Then although they don't offer half portions
of pasta we each ordered the entree size and split them. We chose the
tagliatelle bolognese and a four cheese filled pasta in a creamy sauce with
bacon, asparagus and grana frita. Both were very good. We split a secondi which
was the special Berkshire pork secreto, which was described as a pork skirt
steak with garlic potato purée and romanesco broccoli (incorrectly described as
a romanesco sauce). Some tables received bread before the meal but for some
reason we did not. \n\nManagement also seems capable for when the tenants in the
apartment above began playing basketball she intervened and also comped the
tables a dessert. We ordered the apple dumpling with gelato and it was also
quite tasty. Portions are not huge which I particularly like because I prefer to
order courses. If you are someone who orders just a meal you may leave hungry
depending on you appetite. Dining room was mostly younger crowd while the bar
was definitely the over 40 set. Would recommend that the naysayers return to see
the improvement although I personally don't know the former glory to be able to
compare. Easy access to downtown Salem without the crowds on this month of
October.",
 'date': '2014-10-11 03:34:02',
 'to_avoid': 'false'}
```

[25]:
```python
# Do not delete/edit this cell!
# You must run this cell before running the autograder.
review_boolean = mydb["review_boolean"]
grading_util.save_results("result_1e", list(review_boolean.find({}, {'_id':␣
  ↪0})));
```

```
[26]: grader.check("q1e")
```

```
[26]: q1e results: All test cases passed!
```

1.7.6 Question 1f

Now, you had a change of heart: you decide that it's unfair to label restaurants as `to_avoid` without at least giving them a chance! Remove the `to_avoid` field from the `review_boolean` collection. Calculate the `difference` between the data size of `review_boolean` with the `to_avoid` field and without it. The code for making this calculation is provided but it is up to you to actually remove the field.

Deletions: Deletions in Mongo make use of the `review_boolean.update_one(...)` or `review_boolean.update_many(...)` functionality discussed in Question 1e. However, this time, instead of using the `$set` operator which allows for the creation of new fields, we will use the $unset operator which deletes them! Very tidy!

Before running the next cell, make sure to re-run your cell for 1e so you don't get a difference of 0!

```
[27]: with_avoid = mydb.command("collstats", "review_boolean")['size']

      # YOUR ANSWER BEGINS HERE
      review_boolean.update_many({}, {"$unset": {"to_avoid": ""}})
      # END

      without_avoid = mydb.command("collstats", "review_boolean")['size']
      difference = with_avoid - without_avoid
      difference
```

```
[27]: 149272
```

```
[28]: # Do not delete/edit this cell!
      # You must run this cell before running the autograder.
      grading_util.save_results("result_1f", difference);
```

```
[29]: grader.check("q1f")
```

```
[29]: q1f results: All test cases passed!
```

1.8 Question 2: JSON and Relational Models

1.8.1 Question 2a

Now we have a good idea of how to do retrieval, aggregation, and updates in Mongo. But we haven't talked about why we would want to use Mongo to store JSON! In order to explore this, let's take another look at the `business` collection. We will look at the first two entries.

```
[30]: list(business.find({}).limit(2))
```

```
[30]: [{'_id': ObjectId('656657f8ca20f68f10dacfc5'),
        'business_id': '6iYb2HFDywm3zjuRg0shjw',
        'name': 'Oskar Blues Taproom',
        'address': '921 Pearl St',
        'city': 'Boulder',
        'state': 'CO',
        'postal_code': '80302',
        'latitude': 40.0175444,
        'longitude': -105.2833481,
        'stars': 4.0,
        'review_count': 86,
        'is_open': 1,
        'attributes': {'RestaurantsTableService': 'True',
         'WiFi': "u'free'",
         'BikeParking': 'True',
         'BusinessParking': "{'garage': False, 'street': True, 'validated': False,
      'lot': False, 'valet': False}",
         'BusinessAcceptsCreditCards': 'True',
         'RestaurantsReservations': 'False',
         'WheelchairAccessible': 'True',
         'Caters': 'True',
         'OutdoorSeating': 'True',
         'RestaurantsGoodForGroups': 'True',
         'HappyHour': 'True',
         'BusinessAcceptsBitcoin': 'False',
         'RestaurantsPriceRange2': '2',
         'Ambience': "{'touristy': False, 'hipster': False, 'romantic': False,
      'divey': False, 'intimate': False, 'trendy': False, 'upscale': False, 'classy':
      False, 'casual': True}",
         'HasTV': 'True',
         'Alcohol': "'beer_and_wine'",
         'GoodForMeal': "{'dessert': False, 'latenight': False, 'lunch': False,
      'dinner': False, 'brunch': False, 'breakfast': False}",
         'DogsAllowed': 'False',
         'RestaurantsTakeOut': 'True',
         'NoiseLevel': "u'average'",
         'RestaurantsAttire': "'casual'",
         'RestaurantsDelivery': 'None'},
        'categories': 'Gastropubs, Food, Beer Gardens, Restaurants, Bars, American
      (Traditional), Beer Bar, Nightlife, Breweries',
        'hours': {'Monday': '11:0-23:0',
         'Tuesday': '11:0-23:0',
         'Wednesday': '11:0-23:0',
         'Thursday': '11:0-23:0',
         'Friday': '11:0-23:0',
   'Saturday': '11:0-23:0',
   'Sunday': '11:0-23:0'}},
 {'_id': ObjectId('656657f8ca20f68f10dacfc6'),
  'business_id': 'tCbdrRPZA0oiIYSmHG3J0w',
  'name': 'Flying Elephants at PDX',
  'address': '7000 NE Airport Way',
  'city': 'Portland',
  'state': 'OR',
  'postal_code': '97218',
  'latitude': 45.5889058992,
  'longitude': -122.5933307507,
  'stars': 4.0,
  'review_count': 126,
  'is_open': 1,
  'attributes': {'RestaurantsTakeOut': 'True',
   'RestaurantsAttire': "u'casual'",
   'GoodForKids': 'True',
   'BikeParking': 'False',
   'OutdoorSeating': 'False',
   'Ambience': "{'romantic': False, 'intimate': False, 'touristy': False,
'hipster': False, 'divey': False, 'classy': False, 'trendy': False, 'upscale':
False, 'casual': True}",
   'Caters': 'True',
   'RestaurantsReservations': 'False',
   'RestaurantsDelivery': 'False',
   'HasTV': 'False',
   'RestaurantsGoodForGroups': 'False',
   'BusinessAcceptsCreditCards': 'True',
   'NoiseLevel': "u'average'",
   'ByAppointmentOnly': 'False',
   'RestaurantsPriceRange2': '2',
   'WiFi': "u'free'",
   'BusinessParking': "{'garage': True, 'street': False, 'validated': False,
'lot': False, 'valet': False}",
   'Alcohol': "u'beer_and_wine'",
   'GoodForMeal': "{'dessert': False, 'latenight': False, 'lunch': True,
'dinner': False, 'brunch': False, 'breakfast': True}"},
  'categories': 'Salad, Soup, Sandwiches, Delis, Restaurants, Cafes,
Vegetarian',
  'hours': {'Monday': '5:0-18:0',
   'Tuesday': '5:0-17:0',
   'Wednesday': '5:0-18:0',
   'Thursday': '5:0-18:0',
   'Friday': '5:0-18:0',
   'Saturday': '5:0-18:0',
   'Sunday': '5:0-18:0'}}]
```

What are **two** benefts of storing this data in MongoDB with JSON over a relational database management system such as Postgres? Please reference specific examples from the `business` collection to back up your claims. - Format your answer as follows: 1. Benefit #1, Example #1. 2. Benefit #2, Example #2.

Limit each benefit to 1 sentence and each example to 1 sentence for a total of at most four sentences.

1. Benefit #1: MongoDB seems is excellent in dealing with semi-structured data, for example like having the attributes field that contains a nested, varied key-pair value that you can't do in a relational database without creating more than one table or column, Example #1: Under the attributes field in the business documents such as BusinessParking, Ambience, or GoodForMeal, you would need a very complex schema design in a relational database, while it is easily stored as a nested object in Mongo.
2. Benefit #2: MongoDB lacking a schema style lets it have better flexibity when presenting data, and allows to show high variability throughout different documents which as explained in lecture, is common in real life data, Example #2: the two business entries show varaiabiltih in their field having different categories and attributes, and is difficult to handle in a relational database which might need to do different joins, however it is easily dealt with be accessed in Mongo.

1.8.2 Question 2b

It seems like MongoDB is getting all the love when it comes to JSON support! However, modern iterations of relational databases such as Postgres 9.3+ also have excellent JSON functionality as we will soon explore in this task. First, let's set up a bit of scaffolding. The following cell will import the `yelp_academic_dataset_review.json` data into a table called `reviews` in Postgres yelp database.

[31]:
```
%reload_ext sql
%sql postgresql://jovyan@127.0.0.1:5432/postgres

!psql -h localhost -c 'DROP DATABASE IF EXISTS yelp'
!psql -h localhost -c 'CREATE DATABASE yelp'
!psql -h localhost -d yelp -c 'DROP TABLE IF EXISTS reviews'
!psql -h localhost -d yelp -c 'CREATE TABLE reviews(data TEXT);'
!cat data/yelp_academic_dataset_review.json | psql -h localhost -d yelp -c␣
 ↪"COPY reviews (data) FROM STDIN;"
%sql \l
```

```
DROP DATABASE
CREATE DATABASE
NOTICE:  table "reviews" does not exist, skipping
DROP TABLE
CREATE TABLE
COPY 7500
```

```
Running query in 'postgresql://jovyan@127.0.0.1:5432/postgres'
```

[31]:
```
+---------------+--------+----------+------------+------------+-----------------
--+
|      Name     | Owner  | Encoding |  Collate   |   Ctype    | Access
privileges |
+---------------+--------+----------+------------+------------+-----------------
--+
|    baseball   | jovyan |   UTF8   | en_US.utf8 | en_US.utf8 |        None
|
|      imdb     | jovyan |   UTF8   | en_US.utf8 | en_US.utf8 |        None
|
|     jovyan    | jovyan |   UTF8   | en_US.utf8 | en_US.utf8 |        None
|
|    postgres   | jovyan |   UTF8   | en_US.utf8 | en_US.utf8 |        None
|
|   template0   | jovyan |   UTF8   | en_US.utf8 | en_US.utf8 |     =c/jovyan
|
|               |        |          |            |            |
jovyan=CTc/jovyan |
|   template1   | jovyan |   UTF8   | en_US.utf8 | en_US.utf8 |     =c/jovyan
|
|               |        |          |            |            |
jovyan=CTc/jovyan |
| ucb_buildings | jovyan |   UTF8   | en_US.utf8 | en_US.utf8 |        None
|
|      yelp     | jovyan |   UTF8   | en_US.utf8 | en_US.utf8 |        None
|
+---------------+--------+----------+------------+------------+-----------------
--+
```

Now, run the following cell to connect to the Postgres yelp database. There should be no errors after running the following cell.

[32]:
```
%sql postgresql://jovyan@127.0.0.1:5432/yelp
```

```
Connecting and switching to connection postgresql://jovyan@127.0.0.1:5432/yelp
```

Run the following cell to observe how this new `reviews` table looks. Note that the `data` column is stored as TEXT and not as JSON.

[33]:
```
%%sql
SELECT * FROM reviews LIMIT 2;
```

```
Running query in 'postgresql://jovyan@127.0.0.1:5432/yelp'
```

```
2 rows affected.
```

[33]:
```
+-------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
----------------------------------+
|
data
|
+-------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
----------------------------------+
|
{"review_id":"lWC-xP3rd6obsecCYsGZRg","user_id":"ak0TdVmGKo4pwqdJSTLwWw","busine
ss_id":"buF9druCkbuXLX526sGELQ","stars":4.0,"useful":3,"funny":1,"cool":1,"text"
:"Apparently Prides Osteria had a rough summer as evidenced by the almost empty
dining room at 6:30 on a Friday night. However new blood in the kitchen seems to
have revitalized the food from other customers recent visits. Waitstaff was warm
but unobtrusive. By 8 pm or so when we left the bar was full and the dining room
was much more lively than it had been. Perhaps Beverly residents prefer a later
seating.
|
|
|
| After reading the mixed reviews of late I was a little tentative over our
choice but luckily there was nothing to worry about in the food department. We
started with the fried dough, burrata and prosciutto which were all lovely. Then
although they don't offer half portions of pasta we each ordered the entree size
and split them. We chose the tagliatelle bolognese and a four cheese filled
pasta in a creamy sauce with bacon, asparagus and grana frita. Both were very
good. We split a secondi which was the special Berkshire pork secreto, which was
described as a pork skirt steak with garlic potato purée and romanesco broccoli
(incorrectly described as a romanesco sauce). Some tables received bread before
the meal but for some reason we did not.  |
|
|
|                Management also seems capable for when the tenants in the
apartment above began playing basketball she intervened and also comped the
tables a dessert. We ordered the apple dumpling with gelato and it was also
quite tasty. Portions are not huge which I particularly like because I prefer to
order courses. If you are someone who orders just a meal you may leave hungry
depending on you appetite. Dining room was mostly younger crowd while the bar
was definitely the over 40 set. Would recommend that the naysayers return to see
the improvement although I personally don't know the former glory to be able to
compare. Easy access to downtown Salem without the crowds on this month of
October.","date":"2014-10-11 03:34:02"}                |
|
{"review_id":"8bFej1QE5LXp4O05qjGqXA","user_id":"YoVfDbnISlWOf7abNQACIg","busine
ss_id":"RA4V8pr014UyUbDvI-
LW2A","stars":4.0,"useful":1,"funny":0,"cool":0,"text":"This store is pretty
good. Not as great as Walmart (or my preferred, Milford Target), but closer and
in a easier area to get to.
|
|
The store itself is pretty clean and organized, the staff are friendly (most of
the time), and BEST of all is the Self Checkout this store has!
|
|
Great clearance sections throughout, and great prices on everything in the
store, in general (they pricematch too!).
|
|
Christian, Debbie, Jen and Hanna are all very friendly, helpful, sensitive to
all customer needs. Definitely one of the better Target locations in the area,
and they do a GREAT job assisting customers for being such a busy store. Located
directly in the Framingham Mall on Cochituate Rd / Route 30. 4
stars.","date":"2015-07-03 20:38:25"}
|
+-------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
----------------------------------+
```

Observe how the reviews table consists of one column named `data`. This column contains all the JSON documents in the reviews collection *in text format*. Use Postgres' JSON functions to write a query that converts the JSON fields into their own `TEXT` columns (**hint:** one of the operators in Table 9-40 may be useful). To be more concrete, your query should contain 8 columns in this particular order: `review_id`, `user_id`, `business_id`, `stars`, `useful`, `funny`, `cool`, and `text`. Each row should correspond to one JSON document. Some skeleton code (that does the mundane work of converting data to JSON properly) is provided to you—you will only need to fill in the SELECT clause.

```
[34]: %%sql --save query_2b result_2b <<
SELECT
    values->>'review_id' AS review_id,
    values->>'user_id' AS user_id,
    values->>'business_id' AS business_id,
    values->>'stars' AS stars,
    values->>'useful' AS useful,
    values->>'funny' AS funny,
    values->>'cool' AS cool,
    values->>'text' AS text
FROM (SELECT CAST(regexp_replace(data, E'[\\n\\r]+', '','g') AS JSON) AS values␣
  ↪FROM reviews) b
ORDER BY review_id
LIMIT 10;
```

```
Running query in 'postgresql://jovyan@127.0.0.1:5432/yelp'

10 rows affected.
```

```
[35]: # Do not delete/edit this cell!
# You must run this cell before running the autograder.
query_2b = %sqlcmd snippets query_2b
grading_util.save_results("result_2b", query_2b, result_2b)
result_2b.DataFrame().head()
```

```
[35]:                review_id                 user_id             business_id  \
0  000bviMESLXmlIFKDzCEfw  f7LnyAbhP5OSXvv_xiuZhw  SNuCspoI3HKcwJpZL5FcjQ
1  003VeQn6SrVQS4sYHlc0gg  ba8ZSYE11LVepGCxwP9Vpg  rdS7hBBeukiX4Led9OT8sg
2  00HovWV7VcZZPx5IleoeWA  Nzaq0bJcE3q_bRdFrsFRsA  Ln-8CbKGZGmF-GCqMoMcpA
3  00pmZ82_w6Mpky6dl2jpiA  r7e-6OS8A_gE_0CTUjQR3g  IDxaD_0_9TlWyKKXBFwjMA
4  021UtGruSN1RA5YRS92E7w  6cyn5sP2OCarYV02KWtuGQ  5HMXgD_gui5n0Tc_hadesg

  stars useful funny cool                                               text
0   3.0      2     0    0  I went there not having read any reviews.  Fro…
1   5.0      0     0    0  Molana has one of the most tender Chicken Barg…
2   4.0      0     0    0  This place is really new, but they were pretty…
3   1.0      0     0    0  The food was great .. we had breakfast… the …
4   3.0      0     0    0  The atmosphere is great for drinks but I'd say…
```

```
[36]: grader.check("q2b")
```

```
[36]: q2b results: All test cases passed!
```

1.8.3 Question 2c

One important aspect of data engineering that we have not referred to yet are joins. We saw, through the use of indices, selection/projection pushdown, and various physical implementations (as well as orderings), joins could be done quite efficiently in relational SQL based databases. How do joins fare in Mongo where the data stored is inherently semistructured? Let's investigate! For this question, we have provided you access to the tables `business_complete` and `review_complete` which contain the business and review collections in relational form as described in 2b (the columns of the relations are fields in the JSON document). Each relation has its respective id (`business_id` or `review_id`) column as its primary key.

[37]:

```
!psql -h localhost -d yelp -c 'DROP TABLE IF EXISTS business_complete'
!psql -h localhost -d yelp -c 'CREATE TABLE business_complete(business_id TEXT␣
 ↪PRIMARY KEY, name TEXT, address TEXT, city TEXT, state TEXT, postal_code␣
 ↪TEXT, latitude TEXT,longitude TEXT, stars TEXT, review_count TEXT, is_open␣
 ↪TEXT, attributes TEXT, categories TEXT, hours TEXT);'
!psql -h localhost -d yelp -c 'DROP TABLE IF EXISTS review_complete'
!psql -h localhost -d yelp -c 'CREATE TABLE review_complete(review_id TEXT␣
 ↪PRIMARY KEY, user_id TEXT, business_id TEXT, stars TEXT, useful TEXT, funny␣
 ↪TEXT, cool TEXT,text TEXT);'
!cat data/business.csv | psql -h localhost -d yelp -c "COPY business_complete␣
 ↪(business_id,name,address,city,state,postal_code,latitude,longitude,stars,review_count,is_o
 ↪FROM STDIN CSV HEADER;"
!cat data/review.csv | psql -h localhost -d yelp -c "COPY review_complete␣
 ↪(review_id, user_id, business_id, stars, useful, funny, cool, text) FROM␣
 ↪STDIN CSV HEADER;"
```

```
NOTICE:  table "business_complete" does not exist, skipping
DROP TABLE
CREATE TABLE
NOTICE:  table "review_complete" does not exist, skipping
DROP TABLE
CREATE TABLE
COPY 35
COPY 7500
```

Let's take a look at how `review_complete` looks.

[38]:

```
%%sql
SELECT * FROM review_complete LIMIT 1;
```

```
Running query in 'postgresql://jovyan@127.0.0.1:5432/yelp'
```

```
1 rows affected.
```

[38]:

```
+------------------------+------------------------+------------------------+----
---+--------+-------+------+----------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------------------+
|       review_id        |        user_id          |      business_id         |
stars | useful | funny | cool |
text
|
+------------------------+------------------------+------------------------+----
---+--------+-------+------+----------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------------------+
| lWC-xP3rd6obsecCYsGZRg | akOTdVmGKo4pwqdJSTLwWw | buF9druCkbuXLX526sGELQ |
4.0  |   3    |   1   |  1   | Apparently Prides Osteria had a rough summer as
evidenced by the almost empty dining room at 6:30 on a Friday night. However new
blood in the kitchen seems to have revitalized the food from other customers
recent visits. Waitstaff was warm but unobtrusive. By 8 pm or so when we left
the bar was full and the dining room was much more lively than it had been.
Perhaps Beverly residents prefer a later seating. After reading the mixed
reviews of late I was a little tentative over our choice but luckily there was
nothing to worry about in the food department. We started with the fried dough,
burrata and prosciutto which were all lovely. Then although they don't offer
half portions of pasta we each ordered the entree size and split them. We chose
the tagliatelle bolognese and a four cheese filled pasta in a creamy sauce with
bacon, asparagus and grana frita. Both were very good. We split a secondi which
was the special Berkshire pork secreto, which was described as a pork skirt
steak with garlic potato purée and romanesco broccoli (incorrectly described as
a romanesco sauce). Some tables received bread before the meal but for some
reason we did not. Management also seems capable for when the tenants in the
apartment above began playing basketball she intervened and also comped the
tables a dessert. We ordered the apple dumpling with gelato and it was also
quite tasty. Portions are not huge which I particularly like because I prefer to
order courses. If you are someone who orders just a meal you may leave hungry
depending on you appetite. Dining room was mostly younger crowd while the bar
was definitely the over 40 set. Would recommend that the naysayers return to see
the improvement although I personally don't know the former glory to be able to
compare. Easy access to downtown Salem without the crowds on this month of
October. |
+------------------------+------------------------+------------------------+----
---+--------+-------+------+----------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
------------------------------------------+
```

At this current moment in time, Mongo only supports left joins (via the lookup aggregation stage). This is what we will compare against SQL.

Let's start by writing a SQL query that displays all the reviews along with their associated business information. You should perform a **left join** between the `review_complete` table and the `business_complete` table on the `business_id` column, and you may project all columns. Keep a mental note of the **execution time** that you see in the query plan.

[39]:
```
result_2c_str = 'SELECT * FROM review_complete LEFT JOIN business_complete ON␣
 ↪review_complete.business_id = business_complete.business_id;'
!psql -h localhost -d yelp -c "explain analyze $result_2c_str"
```

```
                                                          QUERY PLAN
--------------------------------------------------------------------------------
---------------------------------------------
 Hash Left Join  (cost=13.82..672.53 rows=7500 width=1020) (actual
time=0.060..3.250 rows=7500 loops=1)
   Hash Cond: (review_complete.business_id = business_complete.business_id)
   ->  Seq Scan on review_complete  (cost=0.00..639.00 rows=7500 width=572)
(actual time=0.003..0.925 rows=7500 loops=1)
   ->  Hash  (cost=11.70..11.70 rows=170 width=448) (actual time=0.039..0.041
rows=35 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 32kB
         ->  Seq Scan on business_complete  (cost=0.00..11.70 rows=170
width=448) (actual time=0.004..0.012 rows=35 loops=1)
 Planning Time: 0.719 ms
 Execution Time: 3.513 ms
(8 rows)
```

Now, let's perform the equivalent left join in Mongo between `review` and `business`. **The output array field should be named as `business_info`**. Feel free to refer to the `$lookup` documentation.

Note: You will provide a single-stage pipeline to `review.aggregate(...)` as your solution. Save your pipeline to `q2c_pipeline`.

[40]:
```
# We first create an index on business_id in the business collection
business.create_index('business_id', unique=True)

q2c_pipeline = [
    {
        "$lookup": {
            "from": "business",
            "localField": "business_id",
            "foreignField": "business_id",
            "as": "business_info"
        }
    }
]

result_2c = list(review.aggregate(q2c_pipeline))[:5]
# Uncomment the line below to see your output
result_2c
```

```
[40]: [{'_id': ObjectId('6566582aca20f68f10dd430e'),
        'review_id': 'lWC-xP3rd6obsecCYsGZRg',
        'user_id': 'ak0TdVmGKo4pwqdJSTLwWw',
        'business_id': 'buF9druCkbuXLX526sGELQ',
        'stars': 4.0,
        'useful': 3,
        'funny': 1,
        'cool': 1,
        'text': "Apparently Prides Osteria had a rough summer as evidenced by the
      almost empty dining room at 6:30 on a Friday night. However new blood in the
      kitchen seems to have revitalized the food from other customers recent visits.
      Waitstaff was warm but unobtrusive. By 8 pm or so when we left the bar was full
      and the dining room was much more lively than it had been. Perhaps Beverly
      residents prefer a later seating. \n\nAfter reading the mixed reviews of late I
      was a little tentative over our choice but luckily there was nothing to worry
      about in the food department. We started with the fried dough, burrata and
      prosciutto which were all lovely. Then although they don't offer half portions
      of pasta we each ordered the entree size and split them. We chose the
      tagliatelle bolognese and a four cheese filled pasta in a creamy sauce with
      bacon, asparagus and grana frita. Both were very good. We split a secondi which
      was the special Berkshire pork secreto, which was described as a pork skirt
      steak with garlic potato purée and romanesco broccoli (incorrectly described as
      a romanesco sauce). Some tables received bread before the meal but for some
      reason we did not. \n\nManagement also seems capable for when the tenants in the
      apartment above began playing basketball she intervened and also comped the
      tables a dessert. We ordered the apple dumpling with gelato and it was also
      quite tasty. Portions are not huge which I particularly like because I prefer to
      order courses. If you are someone who orders just a meal you may leave hungry
      depending on you appetite. Dining room was mostly younger crowd while the bar
      was definitely the over 40 set. Would recommend that the naysayers return to see
      the improvement although I personally don't know the former glory to be able to
      compare. Easy access to downtown Salem without the crowds on this month of
      October.",
  'date': '2014-10-11 03:34:02',
  'business_info': [{'_id': ObjectId('656657f9ca20f68f10dadb7c'),
    'business_id': 'buF9druCkbuXLX526sGELQ',
    'name': 'Prides Osteria',
    'address': '240 Rantoul St',
    'city': 'Beverly',
    'state': 'MA',
    'postal_code': '01915',
    'latitude': 42.5496089,
    'longitude': -70.884046,
    'stars': 3.5,
    'review_count': 83,
    'is_open': 0,
    'attributes': {'OutdoorSeating': 'False',
     'RestaurantsGoodForGroups': 'True',
     'Alcohol': "u'full_bar'",
     'RestaurantsDelivery': 'False',
     'RestaurantsReservations': 'True',
     'BusinessAcceptsCreditCards': 'True',
     'BikeParking': 'True',
     'HasTV': 'True',
     'BusinessParking': "{'garage': False, 'street': True, 'validated': False,
'lot': False, 'valet': False}",
     'RestaurantsAttire': "u'casual'",
     'GoodForKids': 'False',
     'Ambience': "{'romantic': False, 'intimate': False, 'classy': True,
'hipster': False, 'divey': False, 'touristy': False, 'trendy': False, 'upscale':
False, 'casual': True}",
     'NoiseLevel': "u'average'",
     'RestaurantsTakeOut': 'True',
     'RestaurantsPriceRange2': '3'},
    'categories': 'Restaurants, Wine Bars, Nightlife, Farmers Market, Food,
Bars, Italian',
    'hours': {'Tuesday': '17:0-22:0',
     'Wednesday': '17:0-22:0',
     'Thursday': '17:0-22:0',
     'Friday': '17:0-23:0',
     'Saturday': '17:0-23:0',
     'Sunday': '17:0-22:0'}}]},
 {'_id': ObjectId('6566582aca20f68f10dd430f'),
  'review_id': '8bFej1QE5LXp4O05qjGqXA',
  'user_id': 'YoVfDbnISlW0f7abNQACIg',
  'business_id': 'RA4V8pr014UyUbDvI-LW2A',
  'stars': 4.0,
  'useful': 1,
  'funny': 0,
  'cool': 0,
  'text': 'This store is pretty good. Not as great as Walmart (or my preferred,
Milford Target), but closer and in a easier area to get to.  \nThe store itself
is pretty clean and organized, the staff are friendly (most of the time), and
BEST of all is the Self Checkout this store has! \nGreat clearance sections
throughout, and great prices on everything in the store, in general (they
pricematch too!). \nChristian, Debbie, Jen and Hanna are all very friendly,
helpful, sensitive to all customer needs. Definitely one of the better Target
locations in the area, and they do a GREAT job assisting customers for being
such a busy store. Located directly in the Framingham Mall on Cochituate Rd /
Route 30. 4 stars.',
  'date': '2015-07-03 20:38:25',
  'business_info': [{'_id': ObjectId('656657fcca20f68f10db0b24'),
    'business_id': 'RA4V8pr014UyUbDvI-LW2A',
    'name': 'Target',
    'address': '400 Cochituate Rd',
    'city': 'Framingham',
    'state': 'MA',
    'postal_code': '01701',
    'latitude': 42.305328,
    'longitude': -71.398387,
    'stars': 3.0,
    'review_count': 65,
    'is_open': 1,
    'attributes': {'ByAppointmentOnly': 'False',
     'BikeParking': 'True',
     'BusinessAcceptsCreditCards': 'True',
     'CoatCheck': 'False',
     'RestaurantsPriceRange2': '2',
     'DogsAllowed': 'False',
     'OutdoorSeating': 'False',
     'RestaurantsReservations': 'False',
     'WiFi': "u'no'",
     'HappyHour': 'False',
     'WheelchairAccessible': 'True',
     'BusinessParking': "{'garage': False, 'street': False, 'validated': False,
'lot': True, 'valet': False}",
     'HasTV': 'False',
     'Caters': 'False',
     'RestaurantsTableService': 'False',
     'RestaurantsDelivery': 'False',
     'RestaurantsTakeOut': 'None'},
    'categories': 'Department Stores, Optometrists, Home & Garden, Discount
Store, Fashion, Furniture Stores, Grocery, Food, Shopping, Drugstores,
Electronics, Health & Medical',
    'hours': {'Monday': '0:0-0:0',
     'Tuesday': '7:0-22:0',
     'Wednesday': '8:0-22:0',
     'Thursday': '8:0-21:0',
     'Friday': '7:0-23:59',
     'Saturday': '8:0-22:0',
     'Sunday': '8:0-22:0'}}]},
 {'_id': ObjectId('6566582aca20f68f10dd4310'),
  'review_id': 'NDhkzczKjLshODbqDoNLSg',
  'user_id': 'eC5evKn1TWDyHCyQAwguUw',
  'business_id': '_sS2LBIGNT5NQb6PD1Vtjw',
  'stars': 5.0,
  'useful': 0,
  'funny': 0,
  'cool': 0,
  'text': "I called WVM on the recommendation of a couple of friends who had
used them in the past and thought they did a nice job. I'm a fan now,
too.\n\nEvan and Cody showed up right on time for my move this past weekend.
They were friendly and energetic, working quickly but carefully to get all my
things moved out of the old place and into the new one in less than 2.5 hours.
All of my (heavy) furniture arrived in perfect condition, and they took extra
care not to scratch the wood floors in the process.\n\nI hope not to move again
anytime soon, but next time I do, I'll be calling WVM.",
  'date': '2013-05-28 20:38:06',
  'business_info': [{'_id': ObjectId('656657f9ca20f68f10dade62'),
    'business_id': '_sS2LBIGNT5NQb6PD1Vtjw',
    'name': 'Willamette Valley Moving',
    'address': '3055 NW Yeon Ave',
    'city': 'Portland',
    'state': 'OR',
    'postal_code': '97210',
    'latitude': 45.5437383,
    'longitude': -122.7043941,
    'stars': 5.0,
    'review_count': 277,
    'is_open': 1,
    'attributes': {'BusinessAcceptsCreditCards': 'True',
     'ByAppointmentOnly': 'True'},
    'categories': 'Home Services, Packing Services, Movers',
    'hours': {'Monday': '8:0-17:0',
     'Tuesday': '8:0-17:0',
     'Wednesday': '8:0-17:0',
     'Thursday': '8:0-17:0',
     'Friday': '8:0-17:0',
     'Saturday': '8:0-17:0',
     'Sunday': '8:0-13:0'}}]},
 {'_id': ObjectId('6566582aca20f68f10dd4311'),
  'review_id': 'T5fAqjjFooT4VOOeZyuk1w',
  'user_id': 'SFQ1jcnGguOOLYWnbbftAA',
  'business_id': '0AzLzHfOJgL7ROwhdww2ew',
  'stars': 2.0,
  'useful': 1,
  'funny': 1,
  'cool': 1,
  'text': "I've stayed at many Marriott and Renaissance Marriott's and this was
a huge disappointment! The front desk and atrium is nice..there is a starbucks
on site which is nice.\n\nThe rooms are run down and old.  There is a flat
screen but that is to be expected of a Renaissance.\n\nWe got this hotel via
Priceline at a rate of $75/night…good deal for the price but this is not a
true Renaissance.",
  'date': '2010-01-08 02:29:15',
  'business_info': [{'_id': ObjectId('656657f9ca20f68f10dae31e'),
    'business_id': '0AzLzHfOJgL7ROwhdww2ew',
    'name': 'Renaissance Orlando at SeaWorld',
    'address': '6677 Sea Harbor Dr',
    'city': 'Orlando',
    'state': 'FL',
    'postal_code': '32821',
    'latitude': 28.4117333957,
    'longitude': -81.4684724808,
    'stars': 3.5,
    'review_count': 290,
    'is_open': 1,
    'attributes': {'RestaurantsDelivery': 'False',
     'NoiseLevel': "u'quiet'",
     'BusinessAcceptsCreditCards': 'True',
     'RestaurantsGoodForGroups': 'True',
     'RestaurantsReservations': 'True',
     'Ambience': "{'romantic': False, 'intimate': False, 'touristy': False,
'hipster': False, 'divey': False, 'classy': False, 'trendy': False, 'upscale':
False, 'casual': False}",
     'Alcohol': "u'full_bar'",
     'RestaurantsTakeOut': 'False',
     'BusinessParking': "{'garage': False, 'street': False, 'validated': False,
'lot': False, 'valet': False}",
     'RestaurantsAttire': "u'casual'",
     'RestaurantsPriceRange2': '3',
     'OutdoorSeating': 'True',
     'GoodForKids': 'True',
     'WiFi': "u'paid'",
     'WheelchairAccessible': 'True',
     'HasTV': 'True',
     'DogsAllowed': 'False',
     'BusinessAcceptsBitcoin': 'False',
     'ByAppointmentOnly': 'True'},
    'categories': 'Hotels, Hotels & Travel, Restaurants, Event Planning &
Services, Venues & Event Spaces',
    'hours': {'Monday': '0:0-0:0',
     'Tuesday': '0:0-0:0',
     'Wednesday': '0:0-0:0',
     'Thursday': '0:0-0:0',
     'Friday': '0:0-0:0',
     'Saturday': '0:0-0:0',
     'Sunday': '0:0-0:0'}}]},
 {'_id': ObjectId('6566582aca20f68f10dd4312'),
  'review_id': 'sjm_uUcQVxab_EeLCqsYLg',
  'user_id': '0kA0PAJ8QFMeveQWHFqz2A',
  'business_id': '8zehGz9jnxPqXtOc7KaJxA',
  'stars': 4.0,
  'useful': 0,
  'funny': 0,
  'cool': 0,
  'text': "The food is always great here. The service from both the manager as
well as the staff is super. Only draw back of this restaurant is it's super
loud. If you can, snag a patio table!",
  'date': '2011-07-28 18:05:01',
  'business_info': [{'_id': ObjectId('656657f9ca20f68f10dade2f'),
    'business_id': '8zehGz9jnxPqXtOc7KaJxA',
    'name': 'Brasserie Ten Ten',
    'address': '1011 Walnut St',
    'city': 'Boulder',
    'state': 'CO',
    'postal_code': '80302',
    'latitude': 40.0166983,
    'longitude': -105.2820678,
    'stars': 4.5,
    'review_count': 977,
    'is_open': 0,
    'attributes': {'BusinessAcceptsCreditCards': 'True',
     'RestaurantsReservations': 'True',
     'Ambience': "{'romantic': False, 'intimate': False, 'touristy': False,
'hipster': False, 'divey': False, 'classy': True, 'trendy': False, 'upscale':
False, 'casual': False}",
     'RestaurantsAttire': "'casual'",
     'RestaurantsGoodForGroups': 'True',
     'Alcohol': "'full_bar'",
     'RestaurantsPriceRange2': '2',
     'HasTV': 'True',
     'RestaurantsDelivery': 'False',
     'RestaurantsTakeOut': 'True',
     'BusinessParking': "{'garage': True, 'street': True, 'validated': True,
'lot': False, 'valet': False}",
     'WiFi': "u'free'",
     'OutdoorSeating': 'True',
      'NoiseLevel': "u'average'",
      'GoodForKids': 'True',
      'ByAppointmentOnly': 'False',
      'Caters': 'False',
      'BikeParking': 'True',
      'DogsAllowed': 'False',
      'HappyHour': 'True',
      'GoodForMeal': "{'dessert': None, 'latenight': False, 'lunch': None,
 'dinner': True, 'brunch': None, 'breakfast': False}"},
     'categories': 'Restaurants, French',
     'hours': {'Monday': '11:0-22:0',
      'Tuesday': '11:0-22:0',
      'Wednesday': '11:0-22:0',
      'Thursday': '11:0-22:0',
      'Friday': '11:0-23:0',
      'Saturday': '9:0-23:0',
      'Sunday': '9:0-21:0'}}]}]
```

```
[41]: # Do not delete/edit this cell!
      # You must run this cell before running the autograder.
      result_2c = list(review.aggregate(q2c_pipeline))[:5]
      grading_util.save_results("result_2c", result_2c);
```

```
[42]: grader.check("q2c")
```

```
[42]: q2c results: All test cases passed!
```

Run the following cell to examine the query plan for the Mongo query that you just wrote. Again, make a mental note of the execution time that you see (you can find the value corresponding to the key executionTimeMillis).

```
[43]: mydb.command('explain', {'aggregate': 'review', 'pipeline': q2c_pipeline,␣
       ↪'cursor': {}}, verbosity='executionStats')
```

```
[43]: {'explainVersion': '1',
       'stages': [{'$cursor': {'queryPlanner': {'namespace': 'yelp.review',
          'indexFilterSet': False,
          'parsedQuery': {},
          'queryHash': '8B3D4AB8',
          'planCacheKey': 'D542626C',
          'maxIndexedOrSolutionsReached': False,
          'maxIndexedAndSolutionsReached': False,
          'maxScansToExplodeReached': False,
          'winningPlan': {'stage': 'COLLSCAN', 'direction': 'forward'},
          'rejectedPlans': []},
         'executionStats': {'executionSuccess': True,
          'nReturned': 7500,
          'executionTimeMillis': 388,
          'totalKeysExamined': 0,
          'totalDocsExamined': 7500,
          'executionStages': {'stage': 'COLLSCAN',
           'nReturned': 7500,
           'executionTimeMillisEstimate': 0,
           'works': 7502,
           'advanced': 7500,
           'needTime': 1,
           'needYield': 0,
           'saveState': 10,
           'restoreState': 10,
           'isEOF': 1,
           'direction': 'forward',
           'docsExamined': 7500}}},
        'nReturned': 7500,
        'executionTimeMillisEstimate': 4},
       {'$lookup': {'from': 'business',
         'as': 'business_info',
         'localField': 'business_id',
         'foreignField': 'business_id'},
        'totalDocsExamined': 7500,
        'totalKeysExamined': 7500,
        'collectionScans': 0,
        'indexesUsed': ['business_id_1'],
        'nReturned': 7500,
        'executionTimeMillisEstimate': 388}],
      'serverInfo': {'host': 'jupyter-saultopete',
       'port': 27017,
       'version': '5.0.11',
       'gitVersion': 'd08c3c41c105cde798ca934e3ac3426ac11b57c3'},
      'serverParameters': {'internalQueryFacetBufferSizeBytes': 104857600,
       'internalQueryFacetMaxOutputDocSizeBytes': 104857600,
       'internalLookupStageIntermediateDocumentMaxSizeBytes': 104857600,
       'internalDocumentSourceGroupMaxMemoryBytes': 104857600,
       'internalQueryMaxBlockingSortMemoryUsageBytes': 104857600,
       'internalQueryProhibitBlockingMergeOnMongoS': 0,
       'internalQueryMaxAddToSetBytes': 104857600,
       'internalDocumentSourceSetWindowFieldsMaxMemoryBytes': 104857600},
      'command': {'aggregate': 'review',
       'pipeline': [{'$lookup': {'from': 'business',
          'localField': 'business_id',
          'foreignField': 'business_id',
          'as': 'business_info'}}],
       'cursor': {},
       '$db': 'yelp'},
      'ok': 1.0}
```

1.8.4 Question 2d

In the last question, you performed equivalent left joins in both Postgres and Mongo. Now, examine their query plans, paying special attention to `executionTimeMillis`. Which join was faster? What gives that database system you chose an advantage over the other? Keep your response to at most three sentences.

THe PostgreSQL join was faster which has an execution time of 3.896 ms, while mongoDB had an execution time of 403 ms. One of the main reasons PostgreSQL had a faster execution time was because of the fact that it tries to find the most optimized join algorithm, while mongoDB uses $lookup which can be less efficient for something like a larger dataset of a more complex join.

1.9 Question 3: Dataframes / Pandas

1.9.1 Question 3a

So far, we've talked about NoSQL / document databases like Mongo and relational databases like Postgres. Now, we will explore data transformation with a different data model: dataframes. Dataframes are similar to relations with some differences as we will dive into here. To that end, we will use Pandas which is a Python package that allows you to work with dataframes. Pandas is widely adopted by data scientists for data loading, wrangling, cleaning, and analysis. To start, let us export our MongoDB collections into Pandas using a function called `json_normalize`. We need to truncate `business` before we can use it to meet the memory constraints set by Jupyter. The variable `business_trunc` will contain the reference the truncated business collection.

[44]:

```python
business_trunc = mydb["business_trunc"]
count = 0
if business_trunc.count_documents({}) != 1000:
    for document in business.find({}):
        count += 1
        business_trunc.insert_one(document)
        if count == 1000:
            break

business_cursor = business_trunc.find({})
review_cursor = mydb["reviews"].find({})
user_cursor = mydb["users"].find({})

# Load the collections into Pandas.
from pandas import json_normalize
user_df = json_normalize(user_cursor)
review_df = json_normalize(review_cursor)
business_df = json_normalize(business_cursor)
```

For the rest of Question 3, please use the 3 dataframes we just created: `user_df`, `review_df`, and `business_df`. Let's take a look at the first 5 rows of `business_df`.

[45]:

```
business_df.head()
```

[45]:

```
                        _id             business_id                   name  \
0  656657f8ca20f68f10dacfc5  6iYb2HFDywm3zjuRg0shjw      Oskar Blues Taproom
1  656657f8ca20f68f10dacfc6  tCbdrRPZA0oiIYSmHG3J0w  Flying Elephants at PDX
2  656657f8ca20f68f10dacfc7  bvN78flM8NLprQ1a1y5dRg           The Reclaimory
3  656657f8ca20f68f10dacfc8  oaepsyvc0J17qwi8cfr0Wg              Great Clips
4  656657f8ca20f68f10dacfc9  PE9uqAjdw0E4-8mjGl3wVA        Crossfit Terminus

               address         city state postal_code   latitude   longitude  \
0         921 Pearl St      Boulder    CO       80302  40.017544 -105.283348
1  7000 NE Airport Way     Portland    OR       97218  45.588906 -122.593331
2   4720 Hawthorne Ave     Portland    OR       97214  45.511907 -122.613693
3   2566 Enterprise Rd  Orange City    FL       32763  28.914482  -81.295979
4  1046 Memorial Dr SE      Atlanta    GA       30316  33.747027  -84.353424

   stars  …  attributes.GoodForDancing  attributes.BestNights  \
0    4.0  …                        NaN                    NaN
1    4.0  …                        NaN                    NaN
2    4.5  …                        NaN                    NaN
3    3.0  …                        NaN                    NaN
4    4.0  …                        NaN                    NaN

  attributes.Music attributes.BYOB attributes.CoatCheck attributes.Smoking  \
0              NaN             NaN                  NaN                NaN
1              NaN             NaN                  NaN                NaN
2              NaN             NaN                  NaN                NaN
3              NaN             NaN                  NaN                NaN
4              NaN             NaN                  NaN                NaN

  attributes.DriveThru attributes.BYOBCorkage attributes.Corkage  \
0                  NaN                    NaN                NaN
1                  NaN                    NaN                NaN
2                  NaN                    NaN                NaN
3                  NaN                    NaN                NaN
4                  NaN                    NaN                NaN

  attributes.RestaurantsCounterService
0                                  NaN
1                                  NaN
2                                  NaN
3                                  NaN
4                                  NaN

[5 rows x 58 columns]
```

What do you notice about how the columns of `business_df` are constructed? How are values that are not found in every document handled in the pandas dataframe? Compare and contrast this dataframe representation with the document representation we saw with Mongo. Keep your response to at most two sentences.

Here, you can see that the values that aren't present in every document are presented as NaN, and it shows how Pandas deals with missing data while MongoDB can have certain fields be missing in the document. The large difference is that Pandas has a table like representation, and there needs to be consistent schema across all rows while Mongo being a document, we can have lots of variabiltiy in the data.

1.9.2 Question 3b

In the previous question, we talked about how Mongo and Postgres approach joins. Pandas is also capable of performing joins using the merge() function! For this task, perform a inner join on `business_df` with itself on `stars`. The final dataframe should be saved to a variable called `result_3b` and should only contain 3 columns in this particular order: the name of the first restaurant, the name of the second restaurant, and the number of the stars. The column names can be arbitrary.

Hint: Check out this tutorial on selecting a subset of the Dataframe. This will be helpful in the rest of Question 3 as well!

[46]:
```
result_3b = business_df.merge(
    business_df,
    on='stars',
    suffixes=('_first', '_second'))[['name_first', 'name_second', 'stars']]
result_3b
```

[46]:
```
                  name_first                      name_second  stars
0        Oskar Blues Taproom              Oskar Blues Taproom    4.0
1        Oskar Blues Taproom         Flying Elephants at PDX    4.0
2        Oskar Blues Taproom               Crossfit Terminus    4.0
3        Oskar Blues Taproom         Capital City Barber Shop    4.0
4        Oskar Blues Taproom   Star Kreations Salon and Spa    4.0
…                        …                              …    …
153959      White Egret Farm             Bluffs at Town Lake    1.5
153960      White Egret Farm                    Taco Cabana    1.5
153961      White Egret Farm                          Shaws    1.5
153962      White Egret Farm                   Steak 'n Shake    1.5
153963      White Egret Farm                White Egret Farm    1.5

[153964 rows x 3 columns]
```

[47]:
```
# Do not delete/edit this cell!
# You must run this cell before running the autograder.
result_3b.columns = ['first', 'second', 'stars']
```

```
grading_util.save_results("result_3b", result_3b.sort_values(['first',␣
 ↪'second', 'stars'])[:50]);
```

```
[48]: grader.check("q3b")
```

```
[48]: q3b results: All test cases passed!
```

1.9.3 Question 3c

Due to the nested representation of the data, there are a lot of missing fields with NaN values in the `business_df` dataframe as you may have noticed in 3a. Construct a dataframe `missing_value_df` with two columns: `column_name` and `percent_missing`. `percent_missing` should be the percentage of NaN values in the corresponding column in `business_df`.

Hint: use Pandas' isnull() function followed by sum().

```
[49]: missing_value_df = pd.DataFrame({'column_name': business_df.columns,
                                       'percent_missing': (business_df.isnull().sum()␣
       ↪/ len(business_df)) * 100
                                     })
      missing_value_df.head()
```

```
[49]:              column_name  percent_missing
      _id                  _id              0.0
      business_id  business_id              0.0
      name                name              0.0
      address          address              0.0
      city                city              0.0
```

```
[50]: # Do not delete/edit this cell!
      # You must run this cell before running the autograder.
      grading_util.save_results("result_3c", missing_value_df);
```

```
[51]: grader.check("q3c")
```

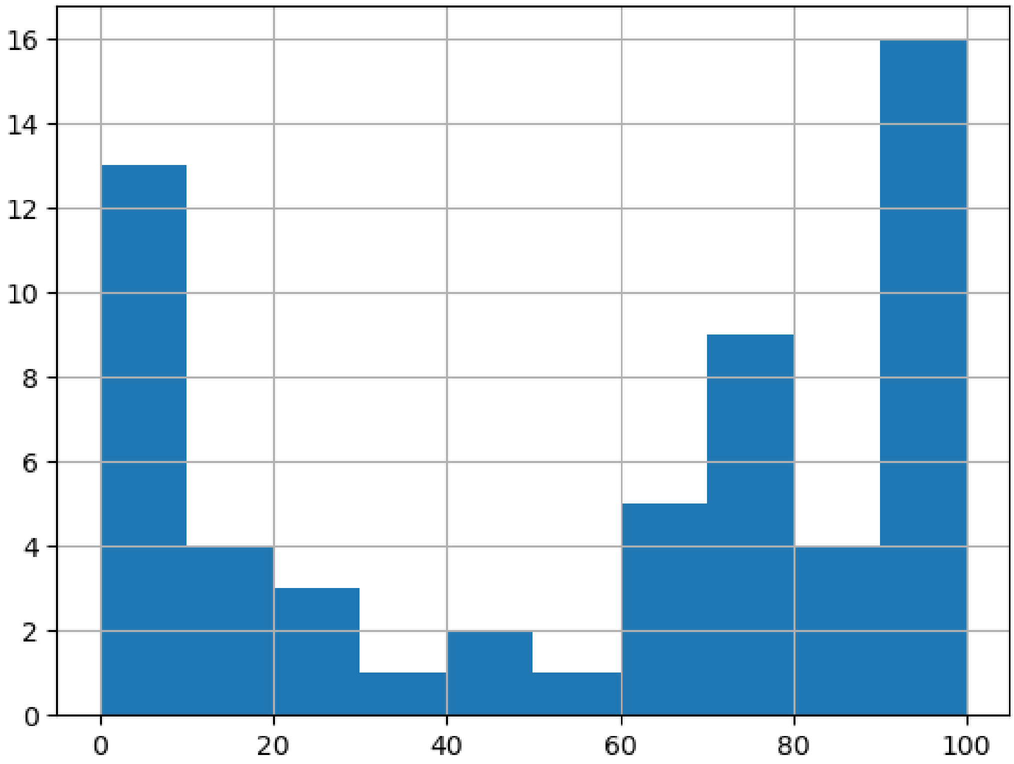
```
[51]: q3c results: All test cases passed!
```

1.9.4 Question 3d

Plot a histogram distribution of the percentage of NaN values across all columns (via Pandas hist() function). Don't worry about putting titles / making it look nice—we won't be grading the plot.

```
[52]: missing_value_df['percent_missing'].hist()
```

```
[52]: <Axes: >
```



Examine the histogram that you just plotted. How many columns are 90%+ NaN? Input your answer into `result_q3d` as an integer (e.g. if your answer is 6, then `result_q3d = 6`)

[53]:
```
result_q3d = 16
```

[54]:
```
grader.check("q3d")
```

[54]:
```
q3d results: All test cases passed!
```

1.9.5 Question 3e

Let us now alter `business_df` to exclude the columns with more than 80%+ null values (keep columns with 80% null values or less). This likely means the corresponding attributes are not an important factor for most businesses so we can get rid of them in our `business_df`. Create a new dataframe called `important_attribute_business_df` which only contains these columns.

Hint: check out this section from the tutorial linked in Q3b.

[55]:
```
#is it .8 or 80???

per_missing = (business_df.isnull().sum() / len(business_df)) * 100
per_missing = per_missing[per_missing <= 80].index
```

```
# per_missing = per_missing[per_missing <= .8].index
important_attribute_business_df = business_df[per_missing]

important_attribute_business_df.head()
```

[55]:

```
                         _id             business_id                    name  \
0  656657f8ca20f68f10dacfc5  6iYb2HFDywm3zjuRg0shjw      Oskar Blues Taproom
1  656657f8ca20f68f10dacfc6  tCbdrRPZA0oiIYSmHG3J0w  Flying Elephants at PDX
2  656657f8ca20f68f10dacfc7  bvN78flM8NLprQ1a1y5dRg           The Reclaimory
3  656657f8ca20f68f10dacfc8  oaepsyvc0J17qwi8cfr0Wg              Great Clips
4  656657f8ca20f68f10dacfc9  PE9uqAjdw0E4-8mjGl3wVA        Crossfit Terminus

              address         city state postal_code   latitude   longitude  \
0        921 Pearl St      Boulder    CO       80302  40.017544 -105.283348
1  7000 NE Airport Way     Portland    OR       97218  45.588906 -122.593331
2   4720 Hawthorne Ave     Portland    OR       97214  45.511907 -122.613693
3   2566 Enterprise Rd  Orange City    FL       32763  28.914482  -81.295979
4  1046 Memorial Dr SE      Atlanta    GA       30316  33.747027  -84.353424

   stars  …  attributes.RestaurantsDelivery  hours.Monday hours.Tuesday  \
0    4.0  …                            None     11:0-23:0     11:0-23:0
1    4.0  …                           False      5:0-18:0      5:0-17:0
2    4.5  …                             NaN           NaN           NaN
3    3.0  …                             NaN           NaN           NaN
4    4.0  …                             NaN     16:0-19:0     16:0-19:0

  hours.Wednesday hours.Thursday hours.Friday hours.Saturday hours.Sunday  \
0       11:0-23:0      11:0-23:0    11:0-23:0      11:0-23:0    11:0-23:0
1        5:0-18:0       5:0-18:0     5:0-18:0       5:0-18:0     5:0-18:0
2             NaN      11:0-18:0    11:0-18:0      11:0-18:0    11:0-18:0
3             NaN            NaN          NaN            NaN          NaN
4       16:0-19:0      16:0-19:0    16:0-19:0       9:0-11:0          NaN

  attributes.GoodForKids attributes.ByAppointmentOnly
0                    NaN                          NaN
1                   True                        False
2                    NaN                        False
3                   True                        False
4                  False                          NaN

[5 rows x 39 columns]
```

[56]:

```
# Do not delete/edit this cell!
# You must run this cell before running the autograder.
grading_util.save_results("result_3e", important_attribute_business_df);
```

[57]:

```
grader.check("q3e")
```

[57]:
```
q3e results: All test cases passed!
```

1.9.6 Question 3f

At this point, you have had experience with manipulating data on Mongo, Postgres, and Pandas. In this question, we will provide 3 scenarios and using the lessons you've learned so far, please specify which of the three (Mongo, Postgres, or Pandas) would work best for this specific use case.

1. You are doing a data journalism piece on college sports. You collect a list of colleges and for each collegiate sport program within that college, you find the budget assigned for that program. You have a choice between the following:

   A) Representing this data in JSON (e.g.

   ```
   {
       "UC Berkeley": {
           "football": "10000000",
           "wrestling": "344582",
           ...}
   }
   ```

) and importing into Mongo.

   B) Representing this data as a schema in Postgres where the columns are the names of the sports.

   C) Representing this data as a dataframe in Pandas where the columns are the names of the sports.

You would like to find the aggregate of budgets across different sports (average, sum, median, mode). What would be the best option for storing this data?

NOTE: Your answer should look like `q3fi_str = ['A']` or `q3fi_str = ['B']` or `q3fi_str = ['C']`.

[58]:
```
q3fi_str = ['C']
```

[59]:
```
grader.check("q3fi")
```

[59]:
```
q3fi results: All test cases passed!
```

2. You would now like to investigate what effect does budget have on student-athlete scholarships. After doing some research, you find a dataset that contains a list of every single athlete at every single college and their sport and scholarship levels (this is a massive 10GB+ dataset with millions of rows). You find another dataset that contains a list of colleges, their sports programs, and the program budget. This is another massive dataset with hundreds of thousands of rows. You would like to perform an inner join between the two datasets on school and program so you can view each student-athlete's scholarship with their sport's budget. You have a choice between the following:

   A) Representing each dataset in JSON (e.g.

```
{"athletes": [
    {"Chase Garbers": {
        "school": "UC Berkeley",
        "scholarship": "full",
        "sport": "football",
        ...
        }
    },
    ...
]}
```

and {"schools": [{"UC Berkeley": { "football": { "budget": "10000000" }, ... } }, ...]}), importing into Mongo, and doing a join there.

B) Representing this data as 2 schemas in Postgres where the columns for the first schema are [`student_name`, `school`, `sport`, `scholarship`] and for the second [`school`, `sport`, `budget`].

C) Representing this data as 2 dataframes in Pandas with the same columns as Postgres.

What would be the best option for storing this data?

NOTE: Your answer should look like `q3fii_str = ['A']` or `q3fii_str = ['B']` or `q3fii_str = ['C']` or `q3fii_str = ['D']`

```
[60]: q3fii_str = ['B']
```

```
[61]: grader.check("q3fii")
```

```
[61]: q3fii results: All test cases passed!
```

3. Finally, you are ready to start writing your article! You decide to focus on just the data from UC Berkeley. You have access to a dataset of just UC Berkeley athletes along with their sports and scholarship levels. The scholarship level data was improperly cleaned: some scholarships are recorded as strings “full”, “half”, or “none” and some are recorded as integer percentages 0-100. You would like to provide this data to your readers in a format that is susceptible to easy visualizations: e.g. graphs that show how many athletes have a full vs. half vs. no scholarship, which sports have the highest percentages of athletes with full scholarships etc. What is the best way to store this data for this purpose?

A) Represent the dataset in JSON e.g.

```
{"athletes": [
    {
      "Chase Garbers": {
        "scholarship": "full",
        "sport": "football"
      }
    },
    {
            "Danielle Vosk": {
              "scholarship": 25,
              "sport": "basketball"
            }
        },
        ...
        ]
    }
```

B) Represent this data as a schema in Postgres where the columns are [`student_name`, `sport`, `scholarship`]

C) Represent this data as a dataframe in Pandas with the same columns as Postgres.

NOTE: Your answer should look like `q3fiii_str = ['A']` or `q3fiii_str = ['B']` or `q3fiii_str = ['C']` or `q3fiii_str = ['D']`

[62]:
```
q3fiii_str = ['C']
```

[63]:
```
grader.check("q3fiii")
```

[63]:
```
q3fiii results: All test cases passed!
```

1.10 Question 4: Messy JSON

Many of the queries you've seen or written thus far were relatively reliable: aggregating and collecting over fields that you know exist for sure. But the nature of Mongo documents is that they are inherently flexible and semi-structured. Not every document will share every single field! In this question, we will explore how Mongo handles these use cases using the `business` collection.

1.10.1 Question 4a

Imagine you are in charge of managing your family reunion. You would like to book a private room at a restaurant. However, you would also like to optimize for chaos. You notice that there is an attribute called `RestaurantsGoodForGroups`. You would like to write a query that returns all restaurants that **do not** have the `RestaurantsGoodForGroups` attribute so that the trajectory of the reunion is determined by fate (**hint:** search up the `$exists` keyword).

How many restaurants do not have the `RestaurantsGoodForGroups` attribute? You may either enter input this as a function with respect to your query or hardcode in either the String or the numeric version of the answer you computed. Ensure that your output for the autograder is the **number of restaurants that do not have the `RestaurantsGoodForGroups` attribute** stored in `q4a_str` as an integer.

Note: You would like this list to consist solely of restaurants. This means that the business must have `Restaurants` in the `categories` field. You may perform a similar text search as question 1d. **This holds true for the rest of the Question 4 as well!**

[64]:
```
# The following text index may be useful!
if 'categories_text' not in business.index_information():
    business.create_index([('categories', TEXT)])
```

```
count_doc = business.count_documents({
    "categories": {"$regex": "Restaurants"},
    "attributes.RestaurantsGoodForGroups": {"$exists": False}
})
q4a_str = count_doc
```

[65]:
```
# Do not delete/edit this cell!
# You must run this cell before running the autograder.
grading_util.save_results("result_4a", q4a_str)[0]
```

[65]:
```
8207
```

[ ]:
```
grader.check("q4a")
```

[ ]:
```
grader.check("q4b")
```

[ ]:
```
grader.check("q4c")
```

[ ]:
```
grader.check("q4d")
```

1.11 Congratulations! You have finished Project 4.

Run the following cell to zip and download the results of your queries. You will also need to run the export cell at the end of the notebook.

For submission on Gradescope, you will need to submit the `proj4.zip` file generated by the export cell. Please ensure that your submission includes `proj4.pdf`.

Please ensure that public tests pass upon submission. It is your responsibility to wait until the autograder finishes running. We will not be accepting regrade requests for submission issues.

Common submission issues: You MUST submit the generated zip files (not folders) to the autograder. However, Safari is known to automatically unzip files upon downloading. You can fix this by going into Safari preferences, and deselect the box with the text “Open safe files after downloading” under the “General” tab.

[ ]:
```
grading_util.prepare_submission_and_cleanup()
```

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```

1.12 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

[ ]:

```
# Save your notebook first, then run this cell to export your submission.
grader.export(files=['results.zip'])
```