Project Title: Analyzing User Reviews and Business Trends with PostgreSQL and MongoDB

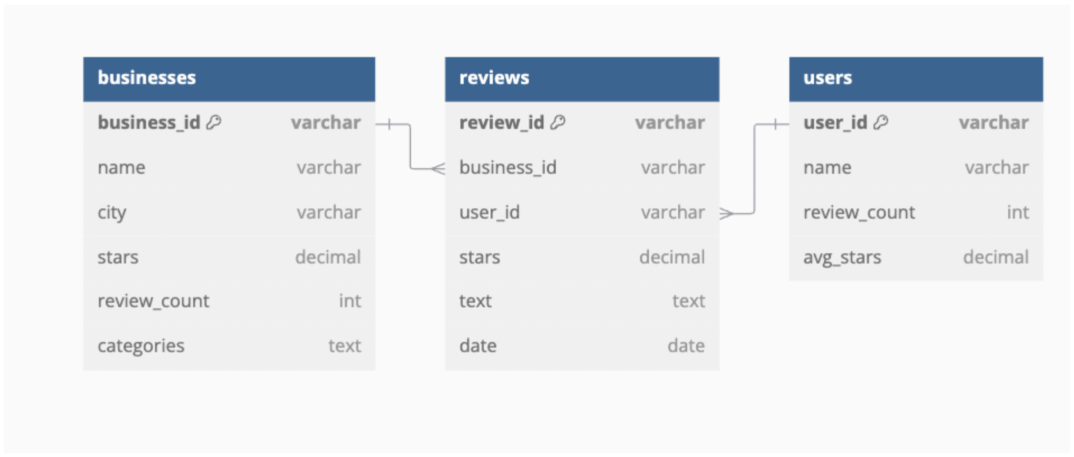
Sierra Dahiyat, Sophia Ladyzhensky, Ananya Chawla, Aidana Jakulina

Dataset Selection

The Yelp dataset comprises multiple JSON files, including yelp_academic_dataset_business.json, which provides detailed information about businesses with attributes such as business_id (primary key), name, address, city, review_count, and categories. Another important file, yelp_academic_dataset_review.json, contains user reviews for these businesses, including a primary key, review_id, and a foreign key, business_id, linking to the business dataset. Additionally, the dataset includes yelp_academic_dataset_user.json, which provides user-specific details with the primary key user_id. Finally, the dataset also features yelp_academic_dataset_checkin.json and yelp_academic_dataset_tip.json, both of which reference business_id and user_id as foreign keys.

Since the full Yelp academic dataset is approximately 10GB, which is large and would use a lot of compute, we reduced it to 1GB through sampling. This will involve selecting a random subset of 10,000 businesses and including only the related users and their reviews. In doing so, we will focus the data on a specific subset of businesses while preserving the relationships between users, businesses, and reviews. With an average of 100 reviews per business, this approach will yield approximately 1 million records, satisfying the requirements for dataset size and multiple foreign key relationships. To sample the data, we will randomly select 10,000 businesses and filter the reviews and users to include only those associated with the selected businesses.

Database structure



businesses	
business_id	varchar
name	varchar
city	varchar
stars	decimal
review_count	int
categories	text

reviews	
review_id	varchar
business_id	varchar
user_id	varchar
stars	decimal
text	text
date	date

users	
user_id	varchar
name	varchar
review_count	int
avg_stars	decimal

ER Diagram

System and Database Setup

Postgres

We loaded the Yelp dataset into PostgreSQL (version 14.14) using a Python script executed on JupyterHub, which served as both our compute and storage resource. We began by defining the schema for the `businesses`, `users`, and `reviews` tables, creating a cursor object and utilizing code provided on the course website as a guide. To populate the tables, we wrote a script to parse the JSON data files (business, review, and user files). We filtered the data by randomly sampling 10,000 businesses and including only the reviews and users associated with those businesses. To maintain the integrity of foreign key constraints, we inserted data in the appropriate order: businesses first, followed by reviews, and then users. The `psycopg` (Version: 3.2.3) library was used to efficiently insert data into PostgreSQL, leveraging its `executemany` function for bulk operations. We utilized our local computers to compute and store our database, and we used Github to share our intermediate versions.

For example, when loading data into the businesses table:
We were first unable to load the files, so we opened the file in read mode and attempted to parse each line. Because of the 'yield' statement, a value is produced one at a time and the entire file is not loaded into memory entirely at once. Then, if there is an invalid JSON, we skip it and print the line that was skipped. Upon analysis there were only a few lines that were skipped, so this was a particularly useful way to deal with the malformed line in the Yelp Dataset. After this was run, we had data that was cleaner to work with:

```
def stream_json(file_path):
    with open(file_path, 'r') as f:
        for line_num, line in enumerate(f, start=1):
            try:
                yield json.loads(line)
            except json.JSONDecodeError as e:
                print(f"Skipping malformed line {line_num}: {e}")
```

We set a random seed in order to ensure replicability and loaded the JSON objects from business_file where each line is one JSON object. This is perhaps memory intensive, but it worked for us. After that, we sampled 10,000 random businesses:

```
random.seed(42)
businesses = list(stream_json(business_file))
sampled_businesses = random.sample(businesses, min(len(businesses), 10000))
```

We prepared data for insertion (and used a similar process to populate the `users` and `reviews` tables) :

```
business_data = [ ( business["business_id"], business.get("name", None),
business.get("city", None), business.get("stars", None),
business.get("review_count", None), business.get("categories", None) ) for
business in sampled_businesses ]
```

Lastly, ran cur.executemany to insert the data:

```
cur.executemany("""
    INSERT INTO businesses (business_id, name, city, stars, review_count,
categories)
    VALUES (%s, %s, %s, %s, %s, %s)
""", business_data)
```

It is also important to note that we had to filter both the reviews and users in order to respect foreign key constraints. First, we had to ensure that only reviews with business_id's referenced in the 10,000 random samples were included:

```
sampled_business_ids = {b["business_id"] for b in sampled_businesses}
filtered_reviews = [review for review in reviews if review["business_id"] in
sampled_business_ids]
```

Then, we filtered the users to only include user_id's who authored a review for one of the sampled businesses: `sampled_user_ids = {review["user_id"] for review in filtered_reviews}`

```
filtered_users = [user for user in users if user["user_id"] in
sampled_user_ids]
```

CSV

To streamline our workflow and avoid redundant computational overhead, we opted to export the sampled dataset into CSV files at the end of our setup notebook. This decision allows us to reuse the prepared data without re-executing the setup script every time a query is modified or new analysis is performed.

We did not include the code to regenerate the CSV files in our repository, as we aimed to ensure that the dataset remains consistent across iterations without the risk of unintentionally generating a new set of files. Additionally, the CSV files themselves are not attached to the GitHub repository due to their large size, which exceeds typical storage limits for version control systems and can lead to performance issues in the repository.

Instead, we rely on a shared local copy of these files for development and execution, ensuring consistency and efficiency in our project workflow.

MongoDB

We loaded the data into MongoDB using the following steps:

1. Setup MongoDB Connection
   - Installed and started the MongoDB service using Homebrew:
     ```
     brew install mongodb-community@6.0
     brew services start mongodb-community@6.0
     ```
   - Connected to MongoDB using the `pymongo` library in Python:
     ```
     from pymongo import MongoClient
     client = MongoClient("mongodb://localhost:27017/")
     db = client["yelp_dataset"]
     ```

2. Loading CSV Data into MongoDB
   - The dataset consisted of three CSV files: `sampled_businesses.csv`, `filtered_reviews.csv`, and `filtered_users.csv`. These were loaded into MongoDB collections as follows:
     ```
     import pandas as pd

     businesses_collection = db["businesses"]
     reviews_collection = db["reviews"]
     users_collection = db["users"]

     def load_csv_to_mongo(file_path, collection):
         data = pd.read_csv(file_path)
         records = data.to_dict(orient="records")  # Convert
     DataFrame to list of dicts
         collection.insert_many(records)
         print(f"Inserted {len(records)} records into
     {collection.name} collection.")

     business_file = "sampled_businesses.csv"
     review_file = "filtered_reviews.csv"
     user_file = "filtered_users.csv"

     load_csv_to_mongo(business_file, businesses_collection)
     load_csv_to_mongo(review_file, reviews_collection)
     load_csv_to_mongo(user_file, users_collection)
     ```

3. Validation

```
print("Number of businesses:",
businesses_collection.count_documents({}))
print("Number of reviews:",
reviews_collection.count_documents({}))
print("Number of users:", users_collection.count_documents({}))
```

Output:

```
Number of businesses: 10000
Number of reviews: 445853
Number of users: 295709
```

PostgreSQL Tasks and Queries

PLEASE USE THIS TEMPLATE WHEN DOING THE WRITE UP

Query Template
[Purpose of the Query] Explain what task the query is trying to accomplish
[Relevance to the Data System] Explain why this is a reasonable tasks for understanding or evaluating your chosen data system
[Query] Include the code
[Output] (If useful for further explanation) include the output (or sample output if it is too large)
[Query Performance]An analysis of its performance. What could make it faster? What did you try to make it faster? For example, if you added an index, show the before and after.

The problems we are trying to solve involve exploring whether certain cities have higher business engagement, greater user diversity per business, and higher review rates per business. We will be taking a look at businesses' abilities to market themselves to a wide population of people while maintaining a positive interaction with those customers.

Task / Query 1

Purpose of the Query
In our first problem, we are attempting to figure out cities with high restaurant rating activity. We would like to see if there is any geographic relation in terms of reviews and/or user engagement. In order to address this question, we group by city and extract the top ten cities with the highest number of reviews.

Relevance to the Data System
This is a reasonable solution as aggregating by cities is a useful level of granularity that allows us to visualize restaurant activity intuitively. Additionally, grouping by city enables us to conduct relevant external research in response to the query outputs.

Query
```
SELECT city, COUNT(*) AS total_reviews
FROM businesses b
JOIN reviews r ON b.business_id = r.business_id
WHERE b.categories ILIKE '%Restaurant%'
GROUP BY city
ORDER BY total_reviews DESC
LIMIT 10;
```

Output
```
('Philadelphia', 41743)
```

```
('New Orleans', 36705)
('Nashville', 18115)
('Tucson', 17201)
('Tampa', 15682)
('Indianapolis', 14534)
('Reno', 12557)
('Santa Barbara', 9665)
('Saint Louis', 8992)
('St. Louis', 5217)
```

Query Performance
Upon running EXPLAIN ANALYZE, we received this output:

```
('Limit  (cost=39189.96..39189.98 rows=10 width=18) (actual time=227.255..227.657 rows=10 loops=1)',)
('              ->  Sort  (cost=38037.61..38038.99 rows=554 width=18) (actual time=217.973..218.025 rows=412 loops=3)',)
('                        ->  Hash Join  (cost=361.65..37682.44 rows=64877 width=10) (actual time=12.445..204.105 rows=99757 loops=3)',)
('                            Hash Cond: ((r.business_id)::text = (b.business_id)::text)',)
('                            ->  Parallel Seq Scan on reviews r  (cost=0.00..36832.88 rows=185788 width=23) (actual time=1.273..164.148 rows=148618 loops=3)',)
('                            ->  Hash  (cost=318.00..318.00 rows=3492 width=33) (actual time=11.031..11.032 rows=3430 loops=3)',)
('                                ->  Seq Scan on businesses b  (cost=0.00..318.00 rows=3492 width=33) (actual time=0.105..9.916 rows=3430 loops=3)',)

('Planning Time: 7.373 ms',)
('Execution Time: 228.323 ms',)
```

Explanation
We can evaluate the query performance by exploring the operation that consumes the most cost, time, and memory. The HashJoin operation consumes the most cost and time when executing the query. The cost of the entire query 1 operation was **cost =39189.96** with an ***actual time=227.255ms.*** The Hash Join operation has a ***cost=361.65..37682.44*** with an ***actual time=12.445..204.105ms.***

First, a sequential scan is performed to identify the restaurant businesses from running the filtering condition: business.categories ILIKE '%Restaurant%.' This identified ***3430*** restaurants in our `businesses` table. Then, a parallel sequential scan operation finds rows in the `reviews` table with business_id that match those in the 3430 restaurant rows. This parallel sequential scan matched **rows=99757.**

Finally, the HashJoin operation matches **rows=99757** joined entries to execute this join operation: JOIN reviews r ON b.business_id = r.business_id and populated the Hash table. Ultimately the HashJoin operation is expensive in time and cost since it is responsible for joining two large datasets, `businesses` and `reviews` based on business_id.

Task / Query 2

Purpose of the Query

In this second query, the goal is to pinpoint the top 10 cities with the highest average rating for businesses labeled as "Restaurants." It gathers the city name, average rating of restaurants (`average_rating`), and the total review count received by all restaurants (`total_reviews`) within each city. This information is valuable for gauging both the popularity and perceived quality of restaurants across different cities. Additionally, the query tallies the total number of reviews for businesses in each city, offering insights into both the city-wide performance and the popularity of the restaurant industry.

Relevance to the data system

This query is important because it allows us to break down restaurant activity by city, offering a clearer picture of local trends. Looking at cities separately provides a level of detail that is more digestible and actionable for further analysis. By focusing on cities with high ratings and review counts, we can identify standout locations, guiding decisions for investments or expansions. Moreover, this level of analysis connects the database structure to practical insights that can help with external research, like comparing outputs to local demographics or tourism patterns.

Query

```
SELECT b.city,
AVG(b.stars) AS average_rating,
SUM(b.review_count) AS total_reviews
FROM businesses b
WHERE  b.categories ILIKE '%Restaurant%'
GROUP BY b.city
ORDER BY average_rating DESC
LIMIT 10;
```

Output

```
('La Vergne', Decimal('5.0000000000000000'), 8)
('San Antonio', Decimal('5.0000000000000000'), 14)
('Harrison Township', Decimal('5.0000000000000000'), 16)
('Greater Northdale', Decimal('4.5000000000000000'), 26)
('Twn N Cntry', Decimal('4.5000000000000000'), 18)
('Camby', Decimal('4.5000000000000000'), 12)
('Green Valley', Decimal('4.5000000000000000'), 6)
('North Redington Beach', Decimal('4.5000000000000000'), 62)
('Hamilton', Decimal('4.5000000000000000'), 528)
('Redington Shores', Decimal('4.5000000000000000'), 45)
```

<u>Query Performance</u>

Upon running EXPLAIN ANALYZE, we received this output

```
('Limit  (cost=39189.96..39189.98 rows=10 width=18) (actual time=288.539..289.612 rows=10 loops=1)',)
('  ->  Sort  (cost=39189.96..39191.34 rows=554 width=18) (actual time=288.537..289.611 rows=10 loops=1)',)
('        Sort Method: top-N heapsort  Memory: 25kB',)
('                        ->  Partial HashAggregate  (cost=38006.82..38012.36 rows=554 width=18) (actual
time=277.161..277.188 rows=413 loops=3)',)
('                              Group Key: b.city',)
('                              ->  Hash Join  (cost=361.65..37682.44 rows=64877 width=10) (actual
time=22.418..263.223 rows=99757 loops=3)',)
('                                    ->  Parallel Seq Scan on reviews r  (cost=0.00..36832.88 rows=185788 width=23)
(actual time=0.440..211.831 rows=148618 loops=3)',)
('                                    ->  Hash  (cost=318.00..318.00 rows=3492 width=33) (actual time=21.881..21.881
rows=3430 loops=3)',)
('                                          Buckets: 4096  Batches: 1  Memory Usage: 252kB',)
('                                          ->  Seq Scan on businesses b  (cost=0.00..318.00 rows=3492 width=33) (actual
time=1.365..18.797 rows=3430 loops=3)',)
("                                                Filter: (categories ~~* '%Restaurant%'::text)",)
('                                                Rows Removed by Filter: 6570',)
('Planning Time: 2.065 ms',)
('Execution Time: 289.970 ms',)
```

<u>Explanation</u>

The query execution plan begins with a sequential scan of the `businesses` table, filtering for entries where the category includes "Restaurant." This scan identifies 3,430 relevant rows out of 10,000, discarding 6,570 that do not meet the criteria. Given that about 34% of the rows meet the filtering condition, adding an index on the `categories` column could be beneficial. An index would enhance performance by enabling PostgreSQL to more efficiently locate the relevant rows through an index scan and by avoiding a full table scan. The benefits of this index would be even more pronounced if a smaller proportion of rows met the filter condition because fewer rows would require examination. Following the filtering, a hash table is constructed from these results, which is important for the subsequent join operation with the `reviews` table.

We recommend adding an index to the `categories` column to reduce the time taken to filter rows and improve overall query efficiency. This would be advantageous as the dataset grows, although it's important to note that maintaining an index does incur overhead in terms of storage and maintenance.

After the filtering, parallel sequential scans are executed on the `reviews` table to collect reviews matching the business IDs from the filtered `businesses` table entries. A hash join follows, utilizing the prepared hash table to efficiently match these reviews to the corresponding businesses based on their `business_id`, resulting in 99,757 joined entries. This hash join, by leveraging a hash table keyed on business IDs from entries categorized under "Restaurant," facilitates a more efficient matching process, underscoring the effectiveness of hash joins in handling large datasets with complex join conditions.

Task / Query 3

Purpose of the Query

In this third query, we are working to find the most active users in each of the top reviewed cities. Query 1 identified the top 10 cities with the highest number of reviews in the “Restaurant” businesses category. So building upon query 1, query 3 identifies the most active users in the top cities by counting how many reviews they have written. This query finds the UserID, number of reviews written in that city and the name of the city.

Relevance to the data system

This query utilizes data from the businesses, reviews, and users tables within the database. It links the businesses table (which stores information about each business, including city and category) with the reviews table (which stores user reviews) to calculate the number of reviews per city. It also connects the reviews table with user data to identify the most frequent reviewers in the top cities. This query relies on the relationships between these tables to provide insights into reviewer behavior and engagement in specific geographic areas.

Query

```
WITH Top10Cities AS (
   SELECT city, COUNT(*) AS total_reviews
   FROM businesses b
   JOIN reviews r ON b.business_id = r.business_id
   WHERE b.categories ILIKE '%Restaurant%'
   GROUP BY city
   ORDER BY total_reviews DESC
   LIMIT 10

)

SELECT r.user_id,
       COUNT(r.review_id) AS user_reviews,
       b.city
FROM reviews r
JOIN businesses b ON r.business_id = b.business_id
JOIN Top10Cities tc ON b.city = tc.city
WHERE b.categories ILIKE '%Restaurant%'
GROUP BY r.user_id, b.city
ORDER BY user_reviews DESC
LIMIT 10;
```

Output

```
('_BcWyKQL16ndpBdggh2kNA', 65, 'Philadelphia')
('ET8n-r7glWYqZhuR6GcdNw', 55, 'Philadelphia')
('1HM81n6n4iPlFU5d2Lokhw', 49, 'New Orleans')
('bJ5FtCtZX3ZZacz2_2PJjA', 49, 'Philadelphia')
('E4BsVQnG5zetbwv2x8QIWg', 47, 'New Orleans')
('Xw7ZjaGfr0WNVt6s_5KZfA', 47, 'New Orleans')
('vHc-UrI9yfL_pnnc6nJtyQ', 45, 'Reno')
('fr1Hz2acAb3OaL3l6DyKNg', 44, 'Tampa')
('vffKQc_WQMYFGY4JS5VAOw', 44, 'Tampa')
('pou3BbKsIozfH50rxmnMew', 39, 'Tampa')
```

Query Performance

Upon running EXPLAIN ANALYZE, we received this output:

('Limit (**cost=80846.59..80846.61** rows=10 width=41) (actual time=481.515..481.558 rows=10 loops=1)',)

The overall **cost** of the entire query was **80846.59** and the **time** that was taken to execute the query was **481.515ms**.

('   -> **Seq Scan on reviews** r (**cost=0.00..39433.92** rows=445892 width=69) (actual time=0.063..59.220 rows=445853 loops=1)',)

('   -> Seq Scan on businesses b (cost=0.00..318.00 rows=3492 width=33) (actual time=0.022..8.568 rows=3430 loops=1)',)

The sequential scanning on reviews and businesses increase the cost and indicate that there is no indexing used within this query. Specifically, in regards to reviews, there are 445892 rows to sort through, and the fact that there is no index significantly affects the cost.

('   -> **GroupAggregate** (**cost=80729.62..80785.84** rows=2811 width=41) (**actual time=427.390..471.924 rows=136484** loops=1)',)

('   Group Key: r.user_id, b.city',)

The group aggregation is a result of the GROUP BY clause in the query, and the execution time of the group aggregation is incredibly **high at ~44.5 ms**. This is likely due to the fact that the group aggregation has to process 136484 rows when making the groups which require a great deal of computation.

```
('                    Sort Method: external merge  Disk: 11792kB',)
```

In order to execute the ORDER BY clause, there is sorting required as PostgreSQL needs to sort the rows to find the top 10 users. The sorting method is quite costly (likely due to the amount of sorting that is needed across the groups), and required disk space due to the amount of memory this query requires. It specifically uses 11792kB of disk space.

```
Hash Join  (cost=39551.86..80568.60 rows=2811 width=56) (actual time=239.419..379.376
rows=180411 loops=1)',)
('                          Hash Cond: ((b.city)::text = (tc.city)::text)',)
('                          ->  Hash Join  (cost=361.65..40966.54 rows=155705 width=56) (actual
time=9.785..126.521 rows=299271 loops=1)',)
('                                Hash Cond: ((r.business_id)::text = (b.business_id)::text)',)
```

The Hash Join that addresses the joining of Top10Cities tc ON b.city = tc.city in the query is quite costly as well. The cost of this operation is **cost=39551.86..80568.60** There is likely a hash join because of how small the Top10Cities dataset is (with only 10 rows), but there were 299,271 rows after joining with the `businesses` table and the hash join is more efficient than a nested loop. The size of the businesses table increased the cost of the Hash Join, and therefore, the Hash Join is quite costly.

MongoDB Tasks and Queries

Task / Query 1

Purpose of the Query

Like **Query 2** from our PostgreSQL section, this query aims to determine the top 10 cities with the highest average restaurant ratings and their corresponding total review counts. Our goal was to compare the performance of the PostgreSQL query with this MongoDB query to evaluate which database system handles the task more efficiently.

Relevance to Data System
This insight is valuable for understanding customer satisfaction and engagement patterns across different locations. By analyzing this data, Yelp could highlight cities with standout restaurant scenes in marketing campaigns, help users discover ‘up and coming’ areas, and guide business owners in selecting optimal locations for new ventures. Comparing the efficiency of PostgreSQL and MongoDB is important for determining the best way to manage Yelp's large dataset, ensuring their system remains scalable.

Query:

```
# Query 1
import random
def get_top_cities_by_rating():
    return [
        {
            "$match": {
                "categories": { "$regex": "Restaurant", "$options": "i" }
            }
        },
        {
            "$group": {
                "_id": "$city",
                "average_rating": { "$avg": "$stars" },
                "total_reviews": { "$sum": "$review_count" }
            }
        },
        {
            "$match": {
                "total_reviews": { "$gte": 100 }
            }
        },
        {
            "$sort": { "average_rating": -1 }
        },
        {
            "$limit": 10  # Limit to top 10 cities
        }
    ]
```

Sampled Output:

```
City: San Antonio, Average Rating: 5.00, Total Reviews: 182
City: Camby, Average Rating: 4.50, Total Reviews: 156
City: Dresher, Average Rating: 4.50, Total Reviews: 1443
City: Twn N Cntry, Average Rating: 4.50, Total Reviews: 234
City: Belle Chasse, Average Rating: 4.50, Total Reviews: 143
```

Query Performance we ran : `db.command({ "explain": {"aggregate": "businesses", "pipeline": pipeline, "cursor": {} }, "verbosity": "executionStats"})`

```
"inputStage": { "stage": "COLLSCAN", "filter": { "categories":
            { "$regex": "Restaurant", "$options": "i" }}, "direction": "forward" }
```

The query starts with a **COLLSCAN** on the businesses collection, where it filters documents based on a regex condition that searches for 'Restaurant' in the categories field. Despite the absence of an index, which leads to examining all 130,000 documents in the collection, the execution is relatively efficient with a total duration (**executionTimeMillis**) of 101 milliseconds. This stage is the most resource-intensive part of the query, as it involves scanning every document in the collection.

```
{"inputStage": {"stage": "PROJECTION_SIMPLE",
          "transformBy": {"city": 1, "review_count": 1, "stars": 1, "_id": 0}}}
```

Following the collection scan, the query proceeds to a projection stage (**PROJECTION_SIMPLE**), where only the relevant fields (city, review_count, stars) are retained and the _id field is omitted. This transformation is crucial for streamlining the dataset for easier processing in subsequent stages

```
"$group": {
  "_id": "$city", "average_rating": {"$avg": "$stars" },
  "total_reviews": {
    "$sum": "$review_counts} }
```

In the $group stage, the query aggregates the data by city, calculating the average rating ($avg: "$stars") and the total number of reviews ($sum: "$review_count"). This aggregation is processed in memory with the execution time estimated(executionTimeMillisEstimate) at 95 milliseconds.

```
"$match": {
  "total_reviews": {
    "$gte": 100}}
```

After grouping, the query applies a `$match` stage to filter cities with at least 100 total reviews. This further refines the results to 400 cities, matching the specific criteria with an execution time also estimated at 95 milliseconds, which is consistent with the time taken for grouping, indicating a streamlined filtering process.

```
{ "$sort": { "sortKey":
        { "average_rating": -1 }, "limit": 10 } }
```

The final significant operation is the `$sort` stage where the results are sorted by average rating in descending order. This operation sorts the data based on the calculated average ratings, ensuring that the top-rated cities are returned first. The estimated execution time for this stage is 95 milliseconds.

Comparison to SQL Query 2:

PostgreSQL involves several layers of operations, starting with a parallel sequence scan on the `reviews` table and a sequential scan on the `businesses` table to filter rows matching "Restaurant" in the categories field. This results in a total execution time of approximately 289.97 milliseconds.

In contrast, the MongoDB query utilizes a collection scan (COLLSCAN) without the aid of indexes on the `categories` field, scanning all documents in the collection to apply the regex filter for "Restaurant." This leads to a significant data throughput with a notable inefficiency, as it has to evaluate each document individually, tallying a total execution time of 101 milliseconds just for the scan and 95 milliseconds for grouping.

In direct comparison, the PostgreSQL query, though seemingly slower in overall execution time compared to the individual stages of the MongoDB query, benefits from a more structured approach to data handling, where sorting and aggregation are optimized through algorithms like top-N heapsort. To optimize performance, both systems would probably benefit from indexing the `categories` field, which would reduce the initial data filtering times. While PostgreSQL shows an edge in handling structured queries efficiently within its transactional framework, MongoDB offers flexibility and scalable solutions in unstructured data environments The choice between the two could depend on the specific requirements of the restaurant analysts and how much this dataset may grow.

Task / Query 2

Purpose of the Query

This query is written in order to identify what the most popular category of business is within each city in the dataset. This question is addressed first by breaking down each category, counting the number of businesses in each category and then determining the top category within a city. Then, the data is listed in alphabetical order.

Relevance to Data System

This task is useful in regards to evaluating MongoDB because we can observe MongoDB handling arrays after $unwind in regards to the categories. This flattens the array and shifts it into a form that MongoDB is specifically suited to deal with. The task requires MongoDB to aggregate and sort items as well which we can further evaluate.

Query:

```
def get_top_category_by_city():

    return [
        {
            "$unwind": "$categories"
        },
        {
            "$group": {
                "_id": {
                    "city": "$city",
                    "category": "$categories"
                },
                "business_count": {"$sum": 1}
            }
        },
        {
            "$sort": {
                "_id.city": 1,
                "business_count": -1
            }
        },
        {
            "$group": {
                "_id": "$_id.city",
                "top_category": {"$first": "$_id.category"},
                "business_count": {"$first": "$business_count"}
            }
        },
        {
            "$sort": {"_id": 1}
        }
    ]


pipeline = get_top_category_by_city()
sampled_results = get_sampled_results(businesses_collection, pipeline)

print("Sampled Results (Top Category by City):")
for result in sampled_results:
    print(f"City: {result['_id']}, Top Category: {result['top_category']}, Business Count:
{result['business_count']}")

explain_result = db.command(
    {
        "explain": {
            "aggregate": "businesses",
            "pipeline": pipeline,
            "cursor": {}
        },
        "verbosity": "executionStats"
    }
)

print("\nPerformance Analysis:")
print(dumps(explain_result, indent=4))
```

Sampled Output:

```
City: Kuna, Top Category: Grocery, Food, Business Count: 13
City: Castleton, Top Category: Ophthalmologists, Shopping, Doctors, Optometrists, Health &
Medical, Eyewear & Opticians, Business Count: 13
City: Willow Grove, Top Category: Photographers, Event Planning & Services, Video/Film
Production, Professional Services, Business Count: 13
City: Millstadt, Top Category: Food, Coffee & Tea, Business Count: 13
City: Meridian, Top Category: Motorcycle Dealers, Automotive, Auto Parts & Supplies, Shopping,
Motorcycle Gear, Motorcycle Repair, Tires, Business Count: 13
```

Query Performance NOTE: this is not the full output, we cut out some of it to highlight relevant pieces of information.

```
 "inputStage": {
                    "stage": "COLLSCAN",
                    "nReturned": 50000,
                    "executionTimeMillisEstimate": 0,
                    "docsExamined": 50000
}
"nReturned": 50000,
        "executionTimeMillisEstimate": 48
```

Here, we can see that the query uses a **collection scan** in order to analyze all 50,000 documents due to the fact that there is no indexing. Without indexing, this query time and cost significantly increases, and this is likely the most costly component of the query (in the sense that it causes all downstream processes to become more costly as well). In this case, there is an execution time of 48ms. We could make this faster by adding an index on city and categories.

```
"$unwind": {
           "path": "$categories"
        },
        "nReturned": 50000,
        "executionTimeMillisEstimate": 75
      }
```

The unwind stage works on expanding the categories array which increases the number of documents for all stages after this - that would likely increase the execution time as all subsequent stages would have many more documents to sift through. As the code dictates, the execution time is 75ms which is over 1.5x more than the collection scan.

```
“$group”:
         "maxAccumulatorMemoryUsageBytes": { "business_count": 1302064 },
         "nReturned": 9574,
         "executionTimeMillisEstimate": 75
```

The above code is isolating some relevant metrics from the aggregation/grouping stage of the query. In this we can see that there are 9,574 groups that are returned and that this requires 1302064 bytes (approximately 1.3 MB) of memory for the count component of the query. This again, could potentially be optimized using an index.

```
"$sort":
        "totalDataSizeSortedBytesEstimate": 6351656,
        "nReturned": 9574,
        "executionTimeMillisEstimate": 95
```

For the sorting component of the query, we can see that the 9574 groups from the grouping stage were sorted, and we can also see that this is very cost intensive. There is an execution time of 95ms - greater than all the other processes - and this process takes 6351656 bytes (approximately 6.3 MB) of memory. Sorting is quite expensive without any indexing especially when there are large datasets.

Tool Comparison:

We chose Postgresql and MongoDB as our two tools for this project.

Installation:

Postgresql and MongoDB installation involved downloading brew on macOS and running the tool on a local server by running the commands brew install postgresql and brew tap mongodb/brew.

Database design

PostgreSQL uses a relational schema that uses foreign keys to preserve relationships between the business, reviews, and user tables. The normalized relational schema prevents redundancy in our schema. MongoDB uses a flexible non relational schema that stores data with little structure.

The process of loading and inserting data into our PostgreSQL database involved creating the tables businesses, reviews and users and loading JSON data from (business_file, review_file, and user_file). The PostgreSQL relational schema design organized the data into these tables while minimizing redundancy with foreign keys. Additionally, the use of cur.executemany improved performance during data insertion by performing batch insertions in the setup process.

The process of loading and inserting data into MongoDB involved connecting to local MongoDB localhost and creating collections for businesses, reviews and users. We converted our CSV files (sampled_businesses.csv, filtered_reviews.csv, and filtered_users.csv) into a list of dictionaries, which are inserted into their respective collections.

Loading and inserting data into MongoDB was more simple than in PostgreSQL, since MongoDB does not require defining relationships between collections. In contrast, PostgreSQL required us to define relationships between tables, which helped prevent redundancy in our schema and reduced the complexity of queries.

Query Performance - query execution times for filtering and aggregations

Initially, our plan was to write queries that achieved the game goal in PostgreSQL and MongoDB. This would allow us to compare performance fairly in terms of execution time for filtering, execution time for aggregations and storage.
For example, we attempted to complete our PostgreSQL query 1 task in MongoDB. However the MongoDB query did not run, which highlights the challenge of executing complex queries in an unstructured database system compared to a relational one like PostgreSQL.So, MongoDB is better suited for projects that involve unstructured data, unlike our project which involved static structured data.

Reflections

Both MongoDB and PostgreSQL have their inherent strengths and weaknesses. In regards to ease of use and setup, both are fairly straightforward. MongoDB and SQL are both incredibly well used and well documented which results in a lot of resources out there in regards to debugging and proceeding with these systems. Both can be used with python and are easy to extend into various projects. At the end of the day, the real decision when it comes to choosing between the two is specifically in regards to the tasks one would like to accomplish.

In a scenario where there are well established relations and a more structured dataset, PostgreSQL is the clear winner when it comes to choosing where to set up the data. The ability to define certain relationships and the ease of execution when it comes to aggregation makes it the obvious choice. When it comes to transaction or inventory management or any set of data that needs to be logged with more specificity and such, PostgreSQL works very well. The structure of PostgresQL also offers a space to manage more complex relationships, dealing with them via subqueries and CTEs and such. This is not readily available in MongoDB. If there is also a desire to analyze internal data that requires VIEWs and TABLEs, then PostgreSQL is set up incredibly well to handle that. PostgreSQL also offers various organizational structures such as B-Trees and partitioning and such in order to filter and run through the data, as well as Materialized Views which make the data easy and quick to access when needed.

However, when the data is less structured, then MongoDB is the obvious choice to use. When the schema in the data is not required to be as structured, then MongoDB has a wonderful way to set up the data. This can be useful for smaller startups that are working on a product with varying attributes, stores that sell multiple items without unifying schema, and other sets of data without many unifying attributes. MongoDB can deal with nested data quite well and address any changes that are required when dealing with the unstructured data. The ability to access and parse through the hierarchical and nested data is unparalleled. If there is a requirement to deal with data that has more abstract connections, MongoDB is better equipped to deal with it.

Personal Reflections

Sierra Dahiyat : I think the most valuable thing I learned was how to load data into a database and share it with other people. It sounds like a simple task, but I had never really thought about it before because all of the setup steps are taken care of for us in our projects. I really struggled writing the code for that and getting it to run due to memory constraints on DataHub. Then we realized we needed to switch to running things locally. The hardest part of this project by far was figuring out how to all have the same setup on four different computers. We spent a lot of time just trying to run things locally and load the data in a reproducible way, but I feel much more competent now. That being said, I actually found solving these problems a lot more fun than I expected. I also learned a lot of randomly valuable things about running stuff locally, which I rarely ever do in classes. It was hard but I learned a lot from this project.

Ananya Chawla : The skills I feel that I learned the most from this project were how to set up your system such that it was scalable and there is an ability to send it across teams. Setting up a project in a way that makes it easier for people to work on it is something that is not immediately obvious, and the toughest part about the project was definitely figuring out how to make sure that everything was compatible across all team members when it comes to the software and accessibility. I also learned a little more about how to integrate different systems like Jupyter and Git and terminal interactions. I think that there is a lot that I took for granted, but there are so many microdecisions in regards to one's set up that can change the way that you can interact with the data. It was frustrating at times not knowing how to deal with things and having to restart portions of the project, but, in the end, once everything was figured out, it was a lot more fun.

Sophia Ladyzhensky: The most important thing I learned was understanding that relational databases like Postgresql are best suited for structured data and non-relational databases like MongoDB are best suited for rapidly changing unstructured data. Another valuable skill I learned was loading json files into a database using Postgresql. The hardest part of this project was learning how to connect locally to the Postgresql server and running terminal commands. It was also difficult to execute consistent query outputs on all my teammates' local setups, especially when loading the database locally on our computers. Despite these challenges, I feel more confident with loading data into a database, connecting to servers locally, and examining "explain analyze" output.

Aidana Jakulina: This project was a valuable opportunity to deepen my understanding of both relational and non-relational database systems. One of the most significant takeaways for me was the importance of designing a robust schema and optimizing queries to balance performance with readability. The most challenging aspect of the project was ensuring consistency across team setups, especially as we transitioned to working locally. Managing foreign key constraints and filtering datasets to maintain their relationships was another intricate but rewarding task. I also appreciated the collaborative aspect of this project; troubleshooting and sharing findings with the team enriched my learning experience. Overall, this project not only enhanced my technical skills but also reaffirmed the importance of adaptability and teamwork in data engineering workflows.