Project Checkpoint

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

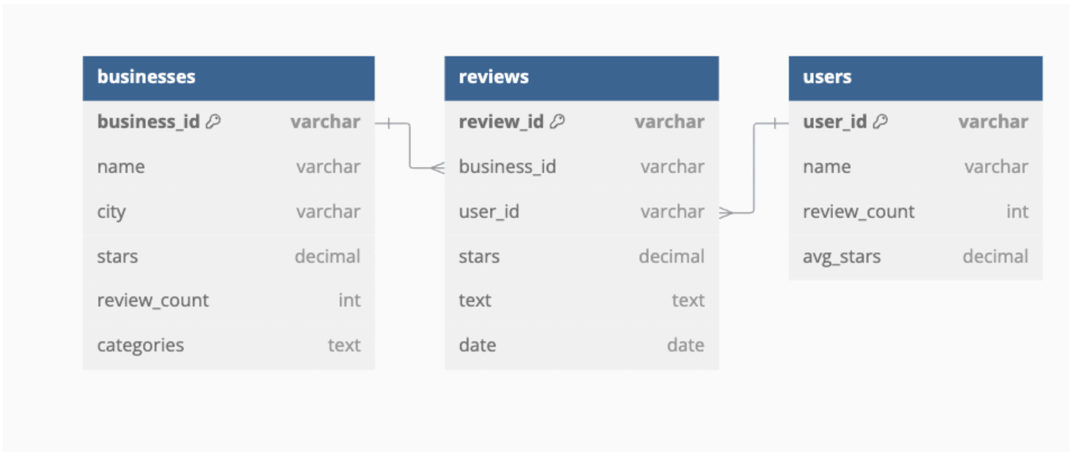
Sierra Dahiyat, Sophia Ladyzhensky, Ananya Chawla, Aidana Jakulina

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—exceeding the size limit for this project—we plan to reduce 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.

ER Diagram



System and Database Setup

We loaded the Yelp dataset into PostgreSQL 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` library was used to efficiently insert data into PostgreSQL, leveraging its `executemany` function for bulk operations. For both storage and compute, we utilized DataHub's PostgreSQL instance, which proved to be the most convenient option for this project.

For example, when loading data into the businesses table:
We first sampled 10,000 random businesses:
`sampled_businesses = random.sample(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]
```

PostgreSQL Tasks and Queries

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

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. 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.

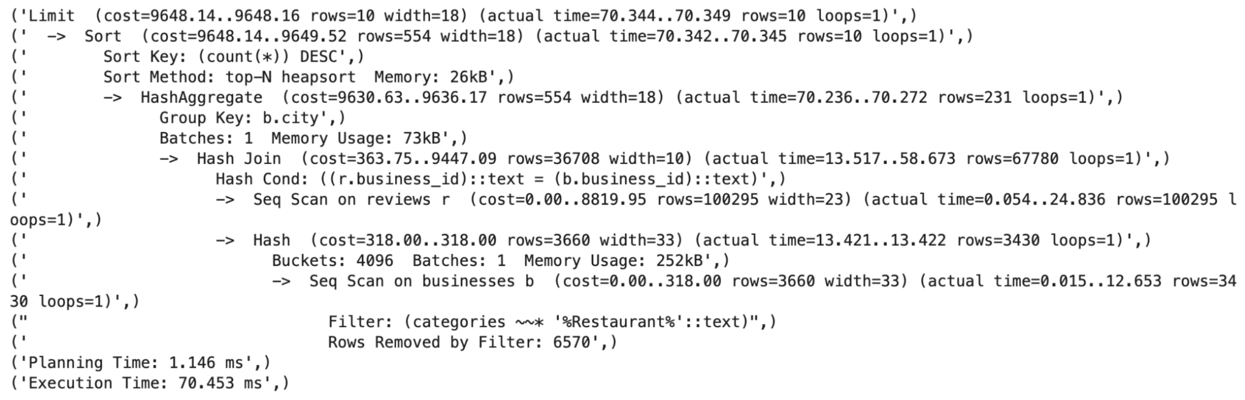
Query:

```
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', 9343)
('New Orleans', 6953)
('Nashville', 6902)
('Indianapolis', 3974)
('Tampa', 3967)
('Tucson', 3677)
('Saint Louis', 2144)
('Santa Barbara', 2098)
('Reno', 2081)
('Edmonton', 1286)
```

Upon running EXPLAIN ANALYZE, the total cost was



```
('Limit  (cost=9648.14..9648.16 rows=10 width=18) (actual time=70.344..70.349 rows=10 loops=1)',)
('  ->  Sort  (cost=9648.14..9649.52 rows=554 width=18) (actual time=70.342..70.345 rows=10 loops=1)',)
('        Sort Key: (count(*)) DESC',)
('        Sort Method: top-N heapsort  Memory: 26kB',)
('        ->  HashAggregate  (cost=9630.63..9636.17 rows=554 width=18) (actual time=70.236..70.272 rows=231 loops=1)',)
('              Group Key: b.city',)
('              Batches: 1  Memory Usage: 73kB',)
('              ->  Hash Join  (cost=363.75..9447.09 rows=36708 width=10) (actual time=13.517..58.673 rows=67780 loops=1)',)
('                    Hash Cond: ((r.business_id)::text = (b.business_id)::text)',)
('                    ->  Seq Scan on reviews r  (cost=0.00..8819.95 rows=100295 width=23) (actual time=0.054..24.836 rows=100295 l
oops=1)',)
('                    ->  Hash  (cost=318.00..318.00 rows=3660 width=33) (actual time=13.421..13.422 rows=3430 loops=1)',)
('                          Buckets: 4096  Batches: 1  Memory Usage: 252kB',)
('                          ->  Seq Scan on businesses b  (cost=0.00..318.00 rows=3660 width=33) (actual time=0.015..12.653 rows=34
30 loops=1)',)
("                                Filter: (categories ~~* '%Restaurant%'::text)",)
('                                Rows Removed by Filter: 6570',)
('Planning Time: 1.146 ms',)
('Execution Time: 70.453 ms',)
```

Evaluate the query performance:

We can evaluate the query performance by exploring the operation that consumes the most cost, time, and memory. The HashAggregate operation consumed the most memory when executing the query. The cost of the entire operation was cost = 9648.14, and the cost of the HashAggregate operation was 9630.63. The Hash Join and Sequential Scan consumed relatively less cost, time, and memory. The HashAggregate is most expensive because the operation creates a hash table, where each unique key is mapped to a corresponding aggregate result (grouped values). This consumes a lot of memory because each key stores aggregated results. The actual time for the HashAggregate operation was 70.342 ms, while the total execution time for the query was 70.453 ms. Again, the HashAggregate took the most time and shows that it is the most expensive operation in the query. Our next step would be to compare this execution time value to the MongoDB explain method explain.executionStats.executionTimeMillis when evaluating the MongoDB query performance.

Future Plan

We plan to use MongoDB as our non-relational database system due to its ability to handle semi-structured data like JSON. MongoDB will allow us to store the dataset with minimal transformations, maintaining the natural structure of the data.

Examples of queries we would be covering:

- Top-Rated Businesses by City
- Most Active Users
- Top Categories by Review Volume
- Business-Specific Reviews

We will compare PostgreSQL and MongoDB on:

1. Performance - query execution times for filtering and aggregations
2. Usability - ease of writing and running queries
3. Flexibility - handling semi-structured data (MongoDB) vs. structured schema (PostgreSQL).

This comparison will help us determine which system is more suitable for specific use cases, such as analytical tasks or real-time querying.