# A Comparison Study between PostgreSQL and MongoDB Using Yelp Dataset

Guotao Ma, Joyce Xu, Li Wan

## 1. Dataset Selection

### 1.1 Description of the Dataset

The dataset for this project is the Yelp dataset (`https://www.yelp.com/dataset/download`), offering comprehensive details about businesses, users, and their interactions. It includes five JSON files containing a different object type on each line.

- `business.json`: contains data about businesses, including their name, location, categories, and operational hours.
- `review.json`: stores user-written reviews about businesses, ratings, and timestamps.
- `user.json`: details about users, including their review activity, friends, and compliments received.
- `checkin.json`: captures check-in timestamps for businesses.
- `tip.json`: contains short user tips for businesses, often quick suggestions or notes.

### 1.2 Sampling Plans

Certain JSON files, such as `review` and `user`, are larger than 1GB, which can lead to efficiency challenges since the upper limit of the field size of PostgreSQL is 1GB,  and even risk crashing the local Jupyter Notebook kernel due to hardware performance limitations. To mitigate these issues, we have adopted the following strategies:

- The size of the `review` JSON file is about 5 GB, and its corresponding dataset has approximately 7 million lines of records. The dataset will be sampled to include only 600,000 reviews.
- The size of the `user` JSON file is about 2.7 GB, and its corresponding dataset has approximately 2 million lines of records. It will be reduced to a

manageable size by filtering out users whose review counts are less than 25.

## 1.3 ER Diagram

After carefully examining the attributes of different tables in the Yelp dataset, we have the following ER diagram to demonstrate the relationship among the tables.

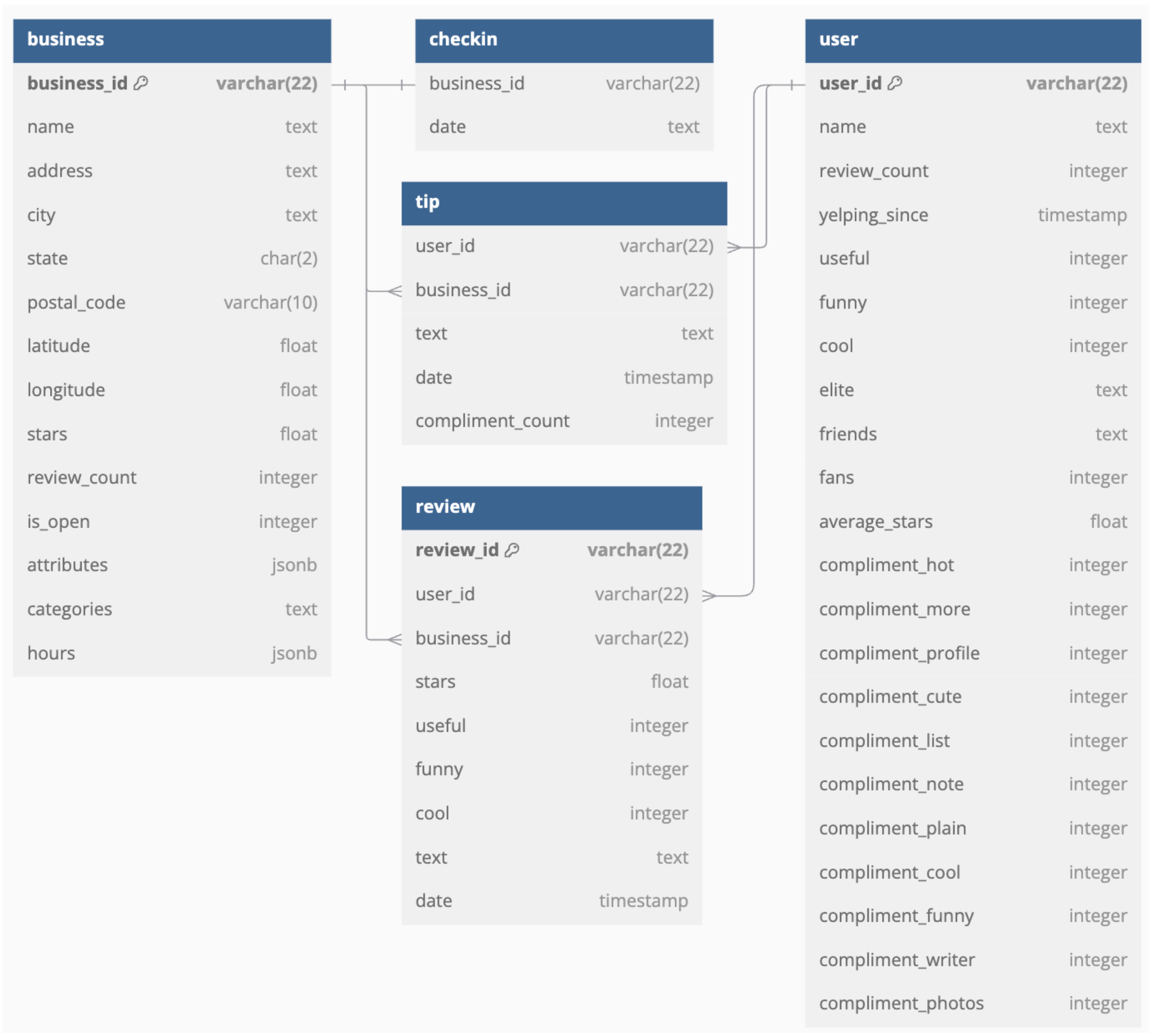

**Figure 1: ER diagram of the Yelp dataset**

## 1.4 Tables Created in PostgreSQL (`tip`, `checkin` excluded)

```
    Column    |       Type       | Collation | Nullable | Default
--------------+------------------+-----------+----------+---------
 business_id  | text             |           | not null |
 name         | text             |           |          |
 address      | text             |           |          |
 city         | text             |           |          |
 state        | text             |           |          |
 postal_code  | text             |           |          |
 latitude     | double precision |           |          |
 longitude    | double precision |           |          |
 stars        | double precision |           |          |
 review_count | integer          |           |          |
 is_open      | integer          |           |          |
 attributes   | text             |           |          |
 categories   | text             |           |          |
 hours        | text             |           |          |
Indexes: "business_pkey" PRIMARY KEY, btree (business_id)
```

**Table 1: business table of the Yelp dataset**

```
       Column       |       Type       | Collation | Nullable | Default
--------------------+------------------+-----------+----------+---------
 user_id            | text             |           | not null |
 name               | text             |           |          |
 review_count       | integer          |           |          |
 yelping_since      | text             |           |          |
 useful             | integer          |           |          |
 funny              | integer          |           |          |
 cool               | integer          |           |          |
 elite              | text             |           |          |
 friends            | text             |           |          |
 fans               | integer          |           |          |
 average_stars      | double precision |           |          |
 compliment_hot     | integer          |           |          |
 compliment_more    | integer          |           |          |
 compliment_profile | integer          |           |          |
 compliment_cute    | integer          |           |          |
 compliment_list    | integer          |           |          |
 compliment_note    | integer          |           |          |
 compliment_plain   | integer          |           |          |
 compliment_cool    | integer          |           |          |
 compliment_funny   | integer          |           |          |
 compliment_writer  | integer          |           |          |
 compliment_photos  | integer          |           |          |
Indexes: "users_pkey" PRIMARY KEY, btree (user_id)
```

**Table 2: user  table of the Yelp dataset**

```
   Column     |       Type       | Collation | Nullable | Default
-------------+------------------+-----------+----------+---------
 review_id   | text             |           | not null |
 user_id     | text             |           |          |
 business_id | text             |           |          |
 stars       | double precision |           |          |
 useful      | integer          |           |          |
 funny       | integer          |           |          |
 cool        | integer          |           |          |
 text        | text             |           |          |
 date        | text             |           |          |
Indexes: "reviews_pkey" PRIMARY KEY, btree (review_id)
```

**Table 3: review table of the Yelp dataset**

## 1.5 Collections Created in MongoDB (`tip`, `checkin` excluded)

```
{'_id': ObjectId('674d162dbdd3fc323197410a'),
 'business_id': 'Pns2l4eNsfO8kk83dixA6A',
 'name': 'Abby Rappoport, LAC, CMQ',
 'address': '1616 Chapala St, Ste 2',
 'city': 'Santa Barbara',
 'state': 'CA',
 'postal_code': '93101',
 'latitude': 34.4266787,
 'longitude': -119.7111968,
 'stars': 5.0,
 'review_count': 7,
 'is_open': 0,
 'attributes': {'ByAppointmentOnly': 'True'},
 'categories': '......', // trimmed
 'hours': None}
```

**Table 4: business collection of the Yelp dataset**

```
{'_id': ObjectId('674d1692bdd3fc3231a2b414'),
 'user_id': 'qVc8ODYU5SZjKXVBgXdI7w',
 'name': 'Walker',
 'review_count': 585,
 'yelping_since': '2007-01-25 16:47:26',
 'useful': 7217,
 'funny': 1259,
 'cool': 5994,
 'elite': '2007',
 'friends': '...', // trimmed
'fans': 267,
 'average_stars': 3.91,
 'compliment_hot': 250,
 'compliment_more': 65,
 'compliment_profile': 55,
 'compliment_cute': 56,
 'compliment_list': 18,
 'compliment_note': 232,
 'compliment_plain': 844,
 'compliment_cool': 467,
 'compliment_funny': 467,
 'compliment_writer': 239,
 'compliment_photos': 180}
```

**Table 5: user collection of the Yelp dataset**

```
{'_id': ObjectId('674d1641bdd3fc3231998c54'),
 'review_id': 'wuPC2rx9HmnVFHRWx6S0pQ',
 'user_id': 'Kc4Hh9Q0IK6mA4bTE-wfIA',
 'business_id': 'TlVR0GNI6MsJyS0e74OhnA',
 'stars': 5.0,
 'useful': 0,
 'funny': 0,
 'cool': 0,
 'text': '......', // trimmed
 'date': '2020-02-02 15:36:22'}
```

**Table 6: review collection of the Yelp dataset**

## 2. System and Database Setup

For this project, we utilized JupyterLab and Python libraries like `psycopg3` and `pymongo4` to simulate the PostgreSQL 14 and MongoDB 8.0 environments on each team member's local machine and manipulate the dataset. Given the large dataset sizes, we opted not to use cloud systems like DataHub or Deepnote, as they might cause the kernels to crash.

### 2.1 Data Preparation

Before loading the data into PostgreSQL, we arbitrarily sampled 600,000 records from the original `review` JSON file and filtered out users who had fewer than 25 reviews in the original user file. Using the `Pandas` library to sample the data tends to be a convenient choice. Yet, we ended up using two Python functions to sample and filter the documents in JSON files line by line due to the memory constraints of local machines. It is important to note that sampling and truncation can impact referential integrity, as described in our ER diagram. Specifically, by reducing the number of records, we risk breaking primary or foreign key relationships, which could lead to certain queries failing. While this approach allowed us to manage the dataset's size and performance, we acknowledge the potential pitfalls of this decision and the trade-offs associated with these limitations. Additionally, although we had not done so, setting a random seed is highly recommended since the sampling process would be reproducible.

Since JSON is a semi-structured format where data fields can vary between records and contain nested objects, it is not ideal for PostgreSQL's relational structure which requires consistent schema definitions. Therefore, we converted all JSON files to CSV format before loading them into the database. This conversion ensured data consistency and simplified the database loading process by transforming the flexible JSON structure into a tabular format that aligns with PostgreSQL's row-column architecture. To make the report succinct, all the codes for Part 2.1 can be found in the Jupyter Notebook.

### 2.2 Loading Data into PostgreSQL

In the Jupyter Notebook, we run `psql` commands to create tables in the database and copy data from CSV files into the corresponding tables. To make the notebook and the report succinct, the queries of creating tables and copying data are saved in two separate .`sql` files: `schema.sql` and `load.sql`; also, we decided not to demonstrate the codes of `schema.sql` in the report. For those who are curious, please refer to the attached file on GitHub.

```
!psql postgres < schema.sql
!psql postgres < load.sql
```

**Table 7: psql commands**

```
\copy business FROM 'business.csv' DELIMITER ',' CSV HEADER;
\copy checkins FROM 'checkin.csv' DELIMITER ',' CSV HEADER;
\copy reviews FROM 'sampled_review.csv' DELIMITER ',' CSV HEADER;
\copy users FROM 'sampled_user.csv' DELIMITER ',' CSV HEADER;
\copy tips FROM 'tip.csv' DELIMITER ',' CSV HEADER;
```

**Table 8: load.sql**

## 3. PostgreSQL Tasks and Queries

Based on a comprehensive analysis of the Yelp dataset's schema, we identified the `review`, `business`, and `user` relations as the most significant for our objectives as these core tables contain the essential data elements. In contrast, the remaining relations were deemed peripheral to our primary research focus.

### 3.1 Task / Query 1

The first PostgreSQL task is to find all users who have written reviews in more than one city and show their names and total counts of unique cities they've reviewed, sorted by the city count in descending order. For those wondering, the name of the most prolific user by this metric was Karen.

From the query plan, we noticed that although the query is relatively expensive, its execution plan is reasonable given the complexity of the operations required. Using parallel processing and hash joins shows PostgreSQL is making appropriate optimization choices.

In terms of optimization, this query is meaningful because it demonstrates efficient design by leveraging fundamental SQL operations that databases have been optimized to execute effectively, and avoids performance pitfalls by not using complex nested subqueries or unnecessarily complicated operations, instead taking a direct path to the desired results. This straightforward approach makes the query both easy to understand and efficient to process, which is ideal for database performance and maintainability.

```
SELECT r.user_id AS user_id,
    u.name AS name,
    COUNT(DISTINCT b.city) AS cities_count
FROM reviews AS r
INNER JOIN business AS b
ON r.business_id = b.business_id
INNER JOIN users AS u
ON r.user_id = u.user_id
GROUP BY r.user_id, u.name
HAVING COUNT(DISTINCT b.city) > 1
ORDER BY COUNT(DISTINCT b.city) DESC;
```

**Table 9: Query of PostgreSQL Task 1**

```
                                    QUERY PLAN
---------------------------------------------------------------------------------
 Sort  (cost=237601.84..238101.71 rows=199946 width=37) (actual
time=2249.243..2266.922 rows=32128 loops=1)
   Sort Key: (count(DISTINCT b.city)) DESC
   Sort Method: quicksort  Memory: 3280kB
   ->  GroupAggregate  (cost=132669.07..214526.84 rows=199946 width=37) (actual
time=2061.068..2260.457 rows=32128 loops=1)
         Group Key: r.user_id, u.name
         Filter: (count(DISTINCT b.city) > 1)
         Rows Removed by Filter: 85932
         ->  Gather Merge  (cost=132669.07..202530.08 rows=599838 width=39) (actual
time=2061.046..2132.972 rows=298058 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               ->  Sort  (cost=131669.04..132293.87 rows=249932 width=39) (actual
time=2049.384..2062.326 rows=99353 loops=3)
                     Sort Key: r.user_id, u.name
                     Sort Method: external merge  Disk: 5264kB
                     Worker 0:  Sort Method: external merge  Disk: 5088kB
                     Worker 1:  Sort Method: external merge  Disk: 4240kB
                     ->  Parallel Hash Join  (cost=39401.17..102425.67 rows=249932
width=39) (actual time=1955.540..2004.521 rows=99353 loops=3)
                           Hash Cond: (r.user_id = u.user_id)
                           ->  Parallel Hash Join  (cost=14825.49..72257.91
rows=249932 width=33) (actual time=1167.364..1210.572 rows=200000 loops=3)
                                 Hash Cond: (r.business_id = b.business_id)
                                 ->  Parallel Seq Scan on reviews r
(cost=0.00..51892.32 rows=249932 width=46) (actual time=1.307..978.576 rows=200000
loops=3)
                                 ->  Parallel Hash  (cost=13552.44..13552.44
rows=62644 width=33) (actual time=56.519..56.519 rows=50115 loops=3)
                                       Buckets: 65536  Batches: 4  Memory Usage:
3104kB
                                       ->  Parallel Seq Scan on business b
```

```
(cost=0.00..13552.44 rows=62644 width=33) (actual time=0.198..28.472 rows=50115
loops=3)
                           ->  Parallel Hash  (cost=21662.53..21662.53 rows=150652
width=29) (actual time=656.015..656.015 rows=120522 loops=3)
                                 Buckets: 65536  Batches: 8  Memory Usage: 3392kB
                                 ->  Parallel Seq Scan on users u
(cost=0.00..21662.53 rows=150652 width=29) (actual time=0.750..544.875 rows=120522
loops=3)
 Planning Time: 2.139 ms
 Execution Time: 2274.229 ms
```

**Table 10: Query Plan of PostgreSQL Task 1**

### 3.2 Task / Query 2

The second PostgreSQL task is to generate a ranked summary of users, including their names, total reviews, average star ratings, and an average star rating category, based on review data joined with user information, sorted by the total number of reviews in descending order. We define the labels of average stars with the following intervals: “excellent”: [4.5, 5], “good”: [3.5, 4.5), “fair”: [2.5, 3.5), “bad”: [0, 2.5). For those wondering, the name of the most prolific user by this metric was Karen with 257 total reviews and a “good” average star rating of 3.64. In the CTE, we filtered out the rows where the `name` column is null since the `review` and `user` tables are sampled, and `user_id` in the `review` table unmatching the one in the `user` table is possible.

The query plan is a reasonable solution for the given query and dataset structure. It correctly executes the operations necessary to join the `review` and `user` tables, group the data by `user_id` and `name`, calculate the total review counts, and sort the results in descending order of total reviews. The use of a `HashAggregate` for grouping and aggregating the data is an efficient choice given the lack of pre-sorted data. Similarly, the Hash Join operation is appropriate for combining the two tables on `user_id`, particularly since no indexes are specified. Sequential scans on both the `review` and `user` tables are also reasonable because the query accesses all rows from these tables, and no additional conditions (beyond filtering out null names) are applied to restrict the data.

The performance of this query plan is suited for the schema as it currently exists. Sequential scans and hash-based operations are effective for processing large datasets when indexes are not available. The query's design to handle 599,838 rows in the reviews table and 361,566 rows in the `user` table shows that PostgreSQL is leveraging parallelism, such as planned partitions during the `HashAggregate`, to speed up processing. However, the sorting step at the end, which processes all grouped results, adds significant overhead. Sorting large datasets can be computationally expensive, particularly when working with high cardinality and limited memory.

```sql
WITH review_with_username AS (
    SELECT r.user_id AS user_id, name, stars
    FROM reviews AS r
    INNER JOIN users AS u
    ON r.user_id = u.user_id
    WHERE name IS NOT NULL
)

SELECT user_id, name,
    COUNT(*) AS total_reviews,
    AVG(stars) AS average_stars,
    (CASE
        WHEN AVG(stars) >= 4.5 THEN 'excellent'
        WHEN AVG(stars) >= 3.5 THEN 'good'
        WHEN AVG(stars) >= 2.5 THEN 'fair'
        ELSE 'bad'
    END) AS star_category
FROM review_with_username
GROUP BY user_id, name
ORDER BY total_reviews DESC;
```

**Table 11: Query of PostgreSQL Task 2**

```
                                 QUERY PLAN
-----------------------------------------------------------------------------------
 Sort  (cost=259480.55..260980.14 rows=599838 width=77) (actual
time=957.471..967.336 rows=118060 loops=1)
   Sort Key: (count(*)) DESC
   Sort Method: external merge  Disk: 7544kB
   ->  HashAggregate  (cost=153888.23..175257.46 rows=599838 width=77) (actual
time=839.288..925.047 rows=118060 loops=1)
         Group Key: r.user_id, u.name
         Planned Partitions: 64  Batches: 65  Memory Usage: 4113kB  Disk Usage:
28448kB
         ->  Hash Join  (cost=30763.24..98403.21 rows=599838 width=37) (actual
time=210.016..736.851 rows=298058 loops=1)
               Hash Cond: (r.user_id = u.user_id)
               ->  Seq Scan on reviews r  (cost=0.00..55391.38 rows=599838
width=31) (actual time=0.897..244.410 rows=600000 loops=1)
               ->  Hash  (cost=23771.66..23771.66 rows=361566 width=29) (actual
time=208.438..208.438 rows=361566 loops=1)
                     Buckets: 65536  Batches: 8  Memory Usage: 3218kB
                     ->  Seq Scan on users u  (cost=0.00..23771.66 rows=361566
width=29) (actual time=0.558..117.630 rows=361566 loops=1)
                           Filter: (name IS NOT NULL)
 Planning Time: 0.975 ms
 Execution Time: 1031.835 ms
```

**Table 12: Query Plan of PostgreSQL Task 2**

### 3.3 Task / Query 3

The third PostgreSQL task is to find out all business information, and "unwind" the `hours` column, which is a JSON-like string. The result includes one row per day and business, even if the business has hours listed for multiple days. The query looks longer than the previous tasks because we need to extract and normalize each day's hours of operation.

The plan uses a sequential scan on the `business` table to iterate through all rows where the hours field is not null, ensuring all businesses with operating hours are considered. A nested loop is applied to join each business row with its corresponding set of hours, generated by splitting and unnesting the `hours` string. The use of `regexp_replace` and `split_part` ensures that the `hours` are parsed cleanly. This approach is logical as it leverages the unnesting of a delimited string into manageable components for efficient parsing and aggregation.

The sequential scan on the `business` table can be expensive for large datasets, as it reads all rows to identify those with non-null hours. Additionally, the nested loop and regular expression functions may introduce overhead due to repeated parsing operations, particularly if the `hours` strings are large or complex. Optimizing the `hours` storage format, such as indexing frequently queried fields could significantly improve performance for larger databases. Finally, the approach in Task 3.3 relies on extensive string manipulation using `regexp_replace` and `split_part`, which can be computationally expensive, particularly on large datasets. While the `LEFT JOIN LATERAL` mechanism is powerful, the performance could be improved by optimizing the storage format of the `hours` column.

```sql
SELECT
    b.business_id,
    b.name,
    b.address,
    b.city,
    SUBSTRING(
        trim(split_part(h.key_value, ':', 1)) FROM 2
        FOR LENGTH(trim(split_part(h.key_value, ':', 1))) - 2
    ) AS day,
    SUBSTRING(trim(
        regexp_replace(
            substring(h.key_value from ':.*'),
                '^:|"|:$', '', 'g'
            )
        ) FROM 2 For LENGTH(
            trim(
                regexp_replace(
                    substring(h.key_value from ':.*'),
                    '^:|"|:$', '', 'g'
                )
            )
        ) - 2
    ) AS hours
FROM business AS b
LEFT JOIN LATERAL (
    SELECT unnest(string_to_array(
        regexp_replace(b.hours, '[{}"]', '', 'g'), ','))  AS key_value
) AS h
ON b.hours IS NOT NULL
WHERE trim(split_part(h.key_value, ':', 1)) IS NOT NULL;
```

**Table 13: Query of PostgreSQL Task 3**

```
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..106351.69 rows=1272280 width=135) (actual
time=3.071..3834.244 rows=801015 loops=1)
   ->  Seq Scan on business b  (cost=0.00..14429.46 rows=127228 width=224) (actual
time=1.367..105.200 rows=127123 loops=1)
         Filter: (hours IS NOT NULL)
         Rows Removed by Filter: 23223
   ->  Subquery Scan on h  (cost=0.00..0.22 rows=10 width=32) (actual
time=0.003..0.005 rows=6 loops=127123)
         Filter: (TRIM(BOTH FROM split_part(h.key_value, ':'::text, 1)) IS NOT
NULL)
         ->  ProjectSet  (cost=0.00..0.07 rows=10 width=32) (actual
time=0.003..0.003 rows=6 loops=127123)
```

```
              ->  Result  (cost=0.00..0.01 rows=1 width=0) (actual
time=0.000..0.000 rows=1 loops=127123)
 Planning Time: 0.453 ms
 Execution Time: 3857.423 ms
```

**Table 14: Query Plan of PostgreSQL Task 3**

## 3.4 Task / Query 4

The fourth PostgreSQL task shares the identical prompt with Task 3.3 but has a different approach. Previously we treated the `hours` column in the `business` table as plain text, and this might not be appropriate to compare with MongoDB. Since PostgreSQL supports JSON type and has several JSON functions for users to manipulate and query JSON data more effectively, we can transform our data handling strategy. Before running the query of Task 3.4, we had to reprocess the `business` table so that the JSON functions of PostgreSQL could manipulate the `hours` column. We utilized `pandas` to replace all single quotes with double quotes in the sampled business CSV, generated a new CSV file, and imported it to the `business` table after clearing all the data that was previously loaded into the database.

```
df = pd.read_csv('business.csv')
df['hours'] = df['hours'].replace("'", '"', regex=True)
df.to_csv('business_dq.csv', index=False)
del df
!psql postgres < task4.sql
```

**Table 15: Data Cleaning  for PostgreSQL Task 4**

```
DELETE FROM business;
\copy business FROM 'business_dq.csv' DELIMITER ',' CSV HEADER;
```

**Table 16: Query of  task4.sql**

The query in Task 3.4 utilizes PostgreSQL's native JSON capabilities, which are well-suited for handling structured data like the `hours` column. Instead of treating `hours` as plain text, it casts the column to a JSON object and uses the `json_each_text` function to break it into key-value pairs. Including the `LATERAL` clause is particularly noteworthy, as it allows the `json_each_text` function to be executed for each row of the `business` table. This ensures that the decomposition of the JSON object into its keys (`day`) and values occurs dynamically and contextually for each business entry. Without the `LATERAL` keyword, the query would not be able to reference the `hours` column from the main `business` table, making this approach both practical and elegant. This design ensures the query is more maintainable and less prone to errors compared to manual parsing methods.

The performance of the Task 3.4 query shows a significant improvement over Task 3.3. While Task 3 involved sequential scans, nested loops, and regular expressions for parsing text, Task 4 leverages the optimized JSON functions built into PostgreSQL, reducing the computational overhead associated with text manipulation. The query execution time in Task 3.4 (600.89 ms) is considerably faster than Task 3.3 (3857.42 ms) due to the more efficient decomposition of JSON data.

```
SELECT business_id, name, address, city,
   key AS day, value AS hours
FROM business,
   LATERAL json_each_text(hours::json);
```

**Table 17: Query of PostgreSQL Task 4**

```
                                          QUERY PLAN
----------------------------------------------------------------------------------
 Nested Loop  (cost=0.01..315121.47 rows=15034600 width=136) (actual
time=3.776..565.416 rows=801015 loops=1)
   ->  Seq Scan on business  (cost=0.00..14429.46 rows=150346 width=225) (actual
time=3.685..48.248 rows=150346 loops=1)
   ->  Function Scan on json_each_text  (cost=0.01..1.01 rows=100 width=64) (actual
time=0.002..0.003 rows=5 loops=150346)
 Planning Time: 14.331 ms
 Execution Time: 600.890 ms
```

**Table 18: Query Plan of PostgreSQL Task 4**

## 4. Non-Relational Tools Comparison

In this section, we provide a focused exploration of MongoDB's querying capabilities and performance characteristics, and demonstrate practical MongoDB queries using the Yelp dataset in JSON format, analyzing their execution and efficiency to offer insights into the database's operational strengths and potential performance considerations. It is important to notice that in Part 4.2, we also had the same prompt as Part 3.3 to demonstrate the distinction between PostgreSQL and MongoDB in terms of performance and code cleanliness. The findings will be elaborated on in the later session.

### 4.1 Task / Query 1

The first MongoDB task is to find all 5-star reviews, and then calculate the total number of 5-star reviews and the average “usefulness” rating of those reviews. The output shows that in the `review` collection, there are 278,020 5-star reviews with an average “usefulness” rating of about 0.97.

The query provides an efficient and insightful approach to analyzing review data, demonstrating the database's powerful analytical capabilities. By examining a collection of 600,000 reviews, the query swiftly identifies and processes all 5-star reviews in just 398 milliseconds, which is remarkable for such a large dataset.

The query's performance is particularly noteworthy due to its comprehensive yet streamlined approach. It conducts a full collection scan, examining all 600,000 documents to filter and extract all 5-star reviews, representing approximately about 45% of the total reviews. This single-pass processing method allows for efficient data aggregation, computing both the total number of 5-star reviews and their average 'usefulness' metric in one seamless operation.

```
mongot1_pipeline = [
    {'$match': { 'stars': 5.0 }},
    {'$group': {
        '_id': 0,
        'totalFiveStarReviews': {'$sum': 1},
        'averageUseful': {'$avg': '$useful'}
        }
    },
    {'$project': {'_id': 0}}
]

list(review.aggregate(mongot1_pipeline))
```

**Table 19: Query of MongoDB Task 1**

```
{'explainVersion': '2',
 'stages': [{'$cursor': {'queryPlanner': {'namespace': 'yelp.review',
     'parsedQuery': {'stars': {'$eq': 5.0}},
......
     'executionTimeMillis': 398,
     'totalKeysExamined': 0,
     'totalDocsExamined': 600000,
     'executionStages': {'stage': 'project',
      'planNodeId': 3,
      'nReturned': 1,
      'executionTimeMillisEstimate': 395,
......
```

**Table 20: Query Plan of MongoDB Task 1 (trimmed)**

## 4.2 Task / Query 2

The second MongoDB task shares the same prompt as PostgreSQL Tasks 3.3 and 3.4, showing essential business information and "unwinding" the `hours` field in `business` collection. In contrast to the tremendous lines of functions in Part 3.3, in MongoDB, we converted the `hours` field into an array with the `$objectToArray` keyword, then deconstructed the array and assigned the elements to their corresponding entries with the `$unwind` keyword.

The provided pipeline effectively solves the problem of extracting and "unwinding" the `hours` field in the `business` collection while preserving essential business information. The `$unwind` stage separates these key-value pairs into individual documents, and a second `$match` ensures that none of the keys or values are null. Finally, the `$project` stage extracts relevant fields like `business_id`, `name`, `address`, and `city`, alongside the individual `day` and `hours` values, providing a clear, flat structure for downstream analysis or visualization.

The query plan reveals that the pipeline performs a collection scan (`COLLSCAN`), examining all 150,346 documents in the `business` collection since no index is used for filtering. While the execution successfully processes 127,123 relevant documents, the total execution time of 1,046 ms indicates room for optimization. Creating an index on the `hours` field could significantly reduce the number of documents examined and improve query performance. Also, the `$unwind` and subsequent stages, which handle the transformation of the `hours` field into a flat structure, operate efficiently and contribute minimal overhead. Overall, while the current approach is functional and handles the transformation well, indexing could make it much faster, especially for larger datasets.

```
mongot2_pipeline = [
    {"$match": {"hours": {"$exists": True, "$ne": None}}},
    {"$project": {"business_id": 1, "name": 1, "address": 1, "city": 1,
                  "hours_array": {"$objectToArray": "$hours"}
                 }
    },
    {"$unwind": "$hours_array"},
    {"$match": {"hours_array.k": {"$ne": None},"hours_array.v": {"$ne": None}}},
    {"$project": {"business_id": 1, "name": 1, "address": 1, "city": 1,
                  "day": "$hours_array.k", "hours": "$hours_array.v"}
    }
]

list(business.aggregate(mongot2_pipeline))[:10]
```

**Table 21: Query of MongoDB Task 2**

```
{'explainVersion': '1',
 'stages': [{'$cursor': {'queryPlanner': {'namespace': 'yelp.business',
     'parsedQuery': {'$and': [{'hours': {'$exists': True}},
       {'hours': {'$not': {'$eq': None}}}]},
    …
       'hours_array': {'$objectToArray': ['$hours']}},
      'inputStage': {'stage': 'COLLSCAN',
    …
     'nReturned': 127123,
     'executionTimeMillis': 1046,
     'totalKeysExamined': 0,
     'totalDocsExamined': 150346,
    …
  {'$unwind': {'path': '$hours_array'},
   'nReturned': 801015,
   'executionTimeMillisEstimate': 807},
 …
```

**Table 22: Query Plan of MongoDB Task 2 (trimmed)**

## 5. Tool Comparisons

### 5.1 Fitness & Ergonomics

PostgreSQL and MongoDB are two powerful tools with distinct strengths, making them suitable for different scenarios. PostgreSQL, a relational database, excels in structured data scenarios requiring complex relationships and schema enforcement. Its setup is straightforward for users familiar with SQL, and its extensive documentation makes it beginner-friendly. Learning PostgreSQL involves mastering SQL syntax and concepts like joins, aggregations, and indexing, which are supported robustly. In practice, our team members love writing SQL queries since it is easy to understand.

On the other hand, MongoDB's NoSQL structure is more flexible and better suited for semi-structured data, such as the JSON-based Yelp dataset. It allows rapid prototyping without a predefined schema and is particularly effective for document-oriented storage. MongoDB’s setup may pose a slight learning curve for users unfamiliar with NoSQL concepts, but its aggregation framework simplifies complex data transformations, such as unwinding nested fields.

### 5.2 Performance

PostgreSQL excels at handling relational queries, especially those involving multiple joins, aggregations, and filtering. For example, tasks like identifying users who are active across multiple cities are well-suited to PostgreSQL, thanks to its efficient query planner and support for indexing. It’s a reliable choice for scenarios requiring detailed exploration of relationships within structured datasets. On the other hand, MongoDB thrives when working with document-based data, particularly in tasks involving nested or semi-structured information. Its functions, such as `$objectToArray` and `$unwind`, make it highly effective for parsing and manipulating JSON data. However, while MongoDB can struggle with relational tasks requiring complex joins, PostgreSQL’s JSON capabilities often feel less fluid than MongoDB’s streamlined aggregation pipeline. Ultimately, the strengths of these tools complement each other: PostgreSQL is the go-to for structured data operations, while MongoDB is ideal for handling flexible, schema-less datasets.

## 6. Team Reflections

First and foremost, as a team, we found setting up the database for PostgreSQL to be challenging, particularly given the large and semi-structured nature of the Yelp dataset. Converting JSON files into a relational format required significant effort, including defining schemas, handling nested fields, and ensuring data consistency across tables. The sheer size of the dataset pushed the limits of our hardware, forcing us to sample and truncate the data to maintain performance.

Despite these initial hurdles, this experience taught us valuable lessons about the critical role of data preparation and schema design in relational databases. Once we overcame these challenges, PostgreSQL proved to be a powerful and efficient tool for our analytical tasks.

Similarly, our experience with MongoDB was insightful but not without its difficulties. We observed that its nested structure and dictionary-like pipeline syntax can pose challenges, particularly when working on complex data transformations. Writing and debugging queries often felt overwhelming at first, as the syntax could quickly become intricate, making it harder to trace operations or locate errors. For team members with a background in relational databases, the transition to MongoDB's NoSQL paradigm involved a steep learning curve. However, as we grew more familiar with the logic and structure of MongoDB's aggregation pipelines, we came to appreciate the flexibility and expressiveness they offer, particularly for manipulating JSON-based or semi-structured datasets. This adaptability made MongoDB an invaluable tool in scenarios where schema-less data management was key.

## 7. Individual Reflections

### 7.1 Team Member: Guotao Ma

During the project, I deeply realized how complex and messy real-world data can be, not only increasing the workload for data cleaning and preprocessing but also directly impacting the choice of database. For instance, the nested structure and irregular fields in the Yelp dataset required significant effort in designing schemas and transforming formats when using a relational database, while MongoDB's flexibility was better suited for handling such semi-structured data. This experience made me understand that the choice of database is not just about comparing technical performance but also about balancing data characteristics and practical needs. Learning to weigh the pros and cons is not only an important step on our journey to becoming successful data engineers but also a valuable lesson for life.

### 7.2 Team Member: Joyce Xu

There are definitely more things for us to learn within the data science world. During the project, we have faced many challenges. And to be honest, designing our own coding problem is much harder than answering a coding problem. Also, sometimes, thinking about a question, and writing a SQL query is easy, but it is much harder for us to convert it into non-relational languages (Ex. MongoDb), so before we write down a problem + query, we need first think about the possibility for the language conversion. Lastly, I think writing a report on Data Science is very different compared to Molecular Cell Biology; although both belong to science, they have many differences based on the data preference. Despite the differences, there are many similarities, and data significantly helps us by making it more convenient to analyze the information we need.

### 7.3 Team Member: Li Wan

This project was harder than I expected. At first, we struggled with converting JSON data into relational formats, which was more complex than we thought. We were also unsure about the difficulty of our queries, so I suggested that my teammates and I attend Professor Lisa and Michael's office hours together—and we did. Professors gave us really helpful advice on our queries and writing, which guided us in improving our work. I also checked the Ed forum daily and shared updates with my teammates, especially tips from professors and TAs. I enjoyed creating the ER diagram because visualizing the relationships between the tables helped us understand the dataset. I also tried writing a few queries myself and realized that some weren’t ideal for comparing performance with MongoDB. By analyzing query performance, I gained a deeper understanding of how MongoDB works differently from PostgreSQL. Overall, it was a challenging but rewarding experience that expanded my knowledge of database systems.