Analyzing Yelp Business Reviews and Tip Trends with PostgreSQL and MongoDB

Shobhan Mangla | Danh Tran | Shashwat Bansal | Allison Nguyen

December 2024

1 Introduction

Efficient data management is a cornerstone of modern analytics, especially when dealing with large, complex datasets. Selecting the right database management system (DBMS) can significantly impact how data is stored, queried, and analyzed. This project evaluates two widely used DBMS technologies—PostgreSQL, a relational database, and MongoDB, a NoSQL database—by analyzing a subset of the Yelp dataset. The Yelp dataset, with its rich collection of business reviews, user interactions, and tip trends, serves as an ideal case study for comparing relational and non-relational database systems. By examining specific tasks such as data insertion, relational joins, and aggregation queries, we aim to assess the performance, scalability, and usability of each database. This report details our methodology, which includes data preprocessing, setting up database schemas, executing analytical queries, and comparing the results across PostgreSQL and MongoDB. Through this analysis, we highlight the strengths and limitations of each DBMS, providing insights into their suitability for various real-world use cases. The following sections cover the dataset and preprocessing steps, database structures and query implementations, a comparative analysis of performance results, and reflections on the findings. By the end of this report, readers will have a deeper understanding of how PostgreSQL and MongoDB perform in a structured analytical context.

2 Dataset Selection

2.1 Dataset Description

For our project, we selected the publicly available Yelp dataset, which was recommended in the project guidelines as a starting point. It provides a comprehensive and rich source of information, including details about businesses on Yelp, restaurant reviews, user interactions, and broader engagement trends. The dataset can be accessed publicly using the following link: `https://www.yelp.com/dataset`

2.2 Sampling and Truncation Plans

We initially chose to use the entire database since it was managed at around 9 GBs for our local systems. However, we realized that this would be extremely slow to run on Jupyterhub for running queries and loading, so we narrowed down to a smaller simple random sample. Thus we chose to sample from the business dataset with 50,000 randomly selected rows. Since each business has around 100 reviews, that would equate to having approximately 50,000,000 records by those two databases alone. First, we proceed to extract all of the datasets first with the following code snippet below where we received our respective JSON files with yelp_academic_dataset_business, yelp_academic_dataset_review, yelp_academic_dataset_user, yelp_academic_dataset_checkin, and yelp_academic_dataset_tip.

```
import tarfile

tar_path = "../data/yelp_dataset.tar"

with tarfile.open(tar_path, "r") as tar:
    tar.extractall()
```

2.3 Additional Data Transformations

We chose to convert these JSON files into CSV files, as CSV is a flat, tabular format, which aligns well with relational database schemas. This conversion has a flattened data structure which will later help us with importing the data into PostgreSQL tables. We defined and utilize the following function below on each of our JSON datasets.

```
def convert_json_to_csv(json_file, csv_file):
    try:
        with open(json_file, 'r', encoding='utf-8') as f:
            data = [json.loads(line) for line in f]

        # Convert to DataFrame
        df = pd.DataFrame(data)

        # Write to CSV
        df.to_csv(csv_file, index=False)
        print(f"Successfully converted {json_file} to {csv_file}.")
    except Exception as e:
        print(f"Error converting {json_file} to {csv_file}: {e}")
```

We proceeded to sample our 50,000 business rows, then filtered down the review dataset to only have reviews that pertains to those sampled business ids in order to not violate any foreign key restrictions. Following that, we similarly filtered down the user dataframe to only contain rows with user_id matching those in the already filtered review dataframe. We chose to utilize the pandas library due to its efficient, high-level abstraction for working with relational data. Unlike vanilla Python, which would require manual, row-by-row iteration and file handling, pandas provides robust built-in functionalities for efficient sampling with its sample() method and relational filtering with isin(). The usage of which can be seen below in the code snippets

```
df = pd.read_csv('yelp_business_cleaned.csv')
sampled_businesses = df.sample(n=50000, random_state=42)
sampled_businesses.to_csv(output_business, index=False)

review_df = pd.read_csv('yelp_review.csv')
sampled_reviews =
    review_df[review_df["business_id"].isin(sampled_businesses["business_id"])]
sampled_reviews.to_csv(output_reviews, index=False)

user_df = pd.read_csv('yelp_user.csv')
sampled_users = user_df[user_df["user_id"].isin(sampled_reviews["user_id"])]
sampled_users.to_csv(output_users, index=False)
```

Following those imports, we decided to also filter the rest of the dataframes being the checkin and tips dataframe to make use of for future query runs.

```
checkin_df = pd.read_csv('yelp_checkin.csv')
sampled_checkins =
    checkin_df[checkin_df["business_id"].isin(sampled_businesses["business_id"])]
sampled_checkins.to_csv(output_checkins, index=False)

tip_df = pd.read_csv('yelp_tip.csv')
sampled_tips =
    tip_df[tip_df["business_id"].isin(sampled_businesses["business_id"])]
sampled_tips =
    sampled_tips[sampled_tips["user_id"].isin(sampled_users["user_id"])]
sampled_tips.to_csv(output_tips, index=False)
```

3 Database Structures

3.1 Database Schema (PostgreSQL)

The following command displays our database schema:

```
\d+
```

Ouptut:

```
     Schema |   Name   | Type | Owner | Persistence | Access method | Size |
        Description
---------+----------+-------+-------+------------+--------------+---------+------------
 public | business | table | Reaga | permanent | heap      | 35 MB  |
 public | checkin | table | Reaga | permanent | heap        | 50 MB   |
 public | review  | table | Reaga | permanent | heap        | 1576 MB |
 public | tip     | table | Reaga | permanent | heap        | 40 MB   |
 public | user    | table | Reaga | permanent | heap        | 1639 MB |
(5 rows)
```

As noted in the Sampling and Truncation section, we are using subsets of all five of the relations from the original dataset. We use the following commands to generate the schema for each table.

```
\d business
\d review
\d user
\d checkin
\d tip
```

3.2 Schema

Business:

```
                  Table "public.business"
   Column    |         Type          | Collation | Nullable | Default
--------------+-----------------------+-----------+----------+---------
 business_id | character varying(50) |           | not null |
 name        | text                  |           |          |
 address     | text                  |           |          |
 city        | text                  |           |          |
 state       | character(3)          |           |          |
 postal_code | character varying(10) |           |          |
 latitude    | double precision      |           |          |
 longitude   | double precision      |           |          |
 stars       | double precision      |           |          |
 review_count | integer              |           |          |
 is_open     | boolean               |           |          |
 attributes  | jsonb                 |           |          |
 categories  | text                  |           |          |
 hours       | jsonb                 |           |          |
Indexes:
   "business_pkey" PRIMARY KEY, btree (business_id)
Referenced by:
   TABLE "checkin" CONSTRAINT "checkin_business_id_fkey" FOREIGN KEY
      (business_id) REFERENCES business(business_id)
   TABLE "review" CONSTRAINT "review_business_id_fkey" FOREIGN KEY (business_id)
      REFERENCES business(business_id)
   TABLE "tip" CONSTRAINT "tip_business_id_fkey" FOREIGN KEY (business_id)
      REFERENCES business(business_id)
```

Reviews:

```
                        Table "public.review"
   Column    |            Type             | Collation | Nullable | Default
-------------+-----------------------------+-----------+----------+---------
 review_id   | character varying(50)       |           | not null |
 user_id     | character varying(50)       |           |          |
 business_id | character varying(50)       |           |          |
 stars       | numeric(2,1)                |           |          |
 useful      | integer                     |           |          |
 funny       | integer                     |           |          |
 cool        | integer                     |           |          |
 text        | text                        |           |          |
 date        | timestamp without time zone |           |          |
Indexes:
    "review_pkey" PRIMARY KEY, btree (review_id)
Foreign-key constraints:
    "review_business_id_fkey" FOREIGN KEY (business_id) REFERENCES
       business(business_id)
```

User:

```
                              Table "public.user"
       Column       |            Type             | Collation | Nullable | Default
--------------------+-----------------------------+-----------+----------+---------
 user_id            | character varying(50)       |           | not null |
 name               | character varying(100)      |           |          |
 review_count       | integer                     |           |          |
 yelping_since      | timestamp without time zone |           |          |
 useful             | integer                     |           |          |
 funny              | integer                     |           |          |
 cool               | integer                     |           |          |
 elite              | text                        |           |          |
 friends            | text                        |           |          |
 fans               | integer                     |           |          |
 average_stars      | numeric(3,2)                |           |          |
 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:
    "user_pkey" PRIMARY KEY, btree (user_id)
    "idx_users_user_id" btree (user_id)
Referenced by:
    TABLE "tip" CONSTRAINT "tip_user_id_fkey" FOREIGN KEY (user_id) REFERENCES
       "user"(user_id)
```

Checkin:

```
                            Table "public.checkin"
   Column    |         Type          | Collation | Nullable | Default
-------------+-----------------------+-----------+----------+---------
 business_id | character varying(50) |           |          |
 date        | text                  |           |          |
Foreign-key constraints:
    "checkin_business_id_fkey" FOREIGN KEY (business_id) REFERENCES
       business(business_id)
```

Tip:

```
                                   Table "public.tip"
      Column      |            Type             | Collation | Nullable | Default
------------------+-----------------------------+-----------+----------+---------
 user_id          | character varying(50)       |           |          |
 business_id      | character varying(50)       |           |          |
 text             | text                        |           |          |
 date             | timestamp without time zone |           |          |
 compliment_count | integer                     |           |          |
Indexes:
    "idx_tip_user_id" btree (user_id)
Foreign-key constraints:
    "tip_business_id_fkey" FOREIGN KEY (business_id) REFERENCES
       business(business_id)
    "tip_user_id_fkey" FOREIGN KEY (user_id) REFERENCES "user"(user_id)
```

3.3 ER Diagram (PostgreSQL)

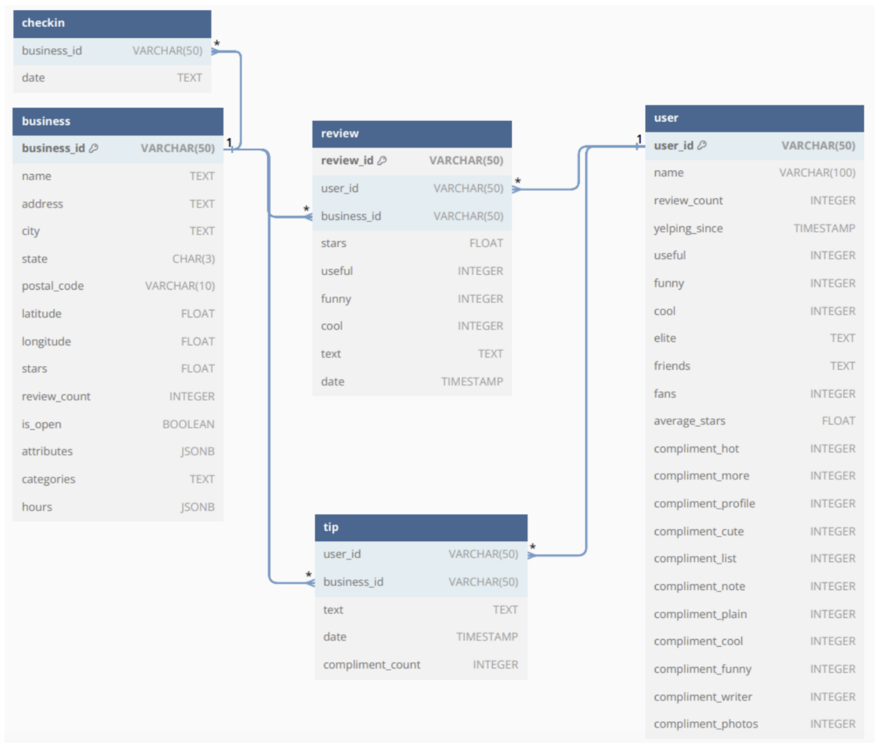


Figure 1: ER Diagram

3.4 MongoDB Database Structure

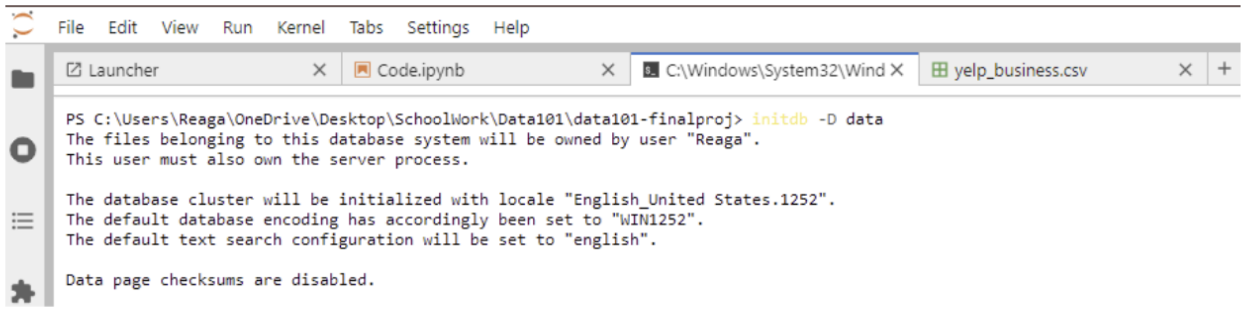
In MongoDB, we created the following collections and populated them with the corresponding document types for each collection. You can observe below the output of a row in each collection to see how document types were given when data was inserted into the MongoDB collections. Due to MongoDB's dynamic structure, schemas are flexible and do not have to be predefined for insertions, which simplifies the process of setting up collections and allows for easier insertions overall. Displayed below is the business collection. The remaining four collections are displayed in the appendix and the end of this report.

```
Business Collection: (business.find_one())
   {'_id': ObjectId('675c09e68f3903b94a189c72'),
 'business_id': 'TacYUYhU3HpLHF9Rs6fW2w',
 'name': 'Steps to Learning Montessori Preschool',
 'address': '6901 Phelps Rd',
 'city': 'Goleta',
 'state': 'CA',
 'postal_code': '93117',
 'latitude': 34.423311,
 'longitude': -119.870637,
 'stars': 4.5,
 'review_count': 8,
 'is_open': True,
 'attributes': '{"BusinessAcceptsCreditCards": "True", "WiFi": "u\'no\'",
     "BusinessAcceptsBitcoin": "False"}',
 'categories': 'Education, Elementary Schools, Child Care & Day Care, Local
     Services, Preschools, Montessori Schools',
 'hours': '{"Monday": "0:0-0:0", "Tuesday": "8:0-17:0", "Wednesday": "8:0-17:0",
     "Thursday": "8:0-17:0", "Friday": "8:0-17:0"}'}
```

4 System and Database Setup

4.1 PostgreSQL Data Setup

We decided to first test by loading the Yelp dataset into Postgres first for a structured data storage system. To do so, we first had to locally install PostgreSQL from their official website ourselves. We were conducting this on a jupyter notebook locally with anaconda, so we opened up a terminal to initialize the database cluster as seen below, where the database system was owned by "Reaga", one of our members. This was done on each of our local devices individually as we each have different systems (Windows, Mac, Linux).



```
PS C:\Users\Reaga\OneDrive\Desktop\SchoolWork\Data101\data101-finalproj> initdb -D data
The files belonging to this database system will be owned by user "Reaga".
This user must also own the server process.

The database cluster will be initialized with locale "English_United States.1252".
The default database encoding has accordingly been set to "WIN1252".
The default text search configuration will be set to "english".

Data page checksums are disabled.
```

Figure 2: Postgres Setup

All of the code below will be done in the terminal system. We first initialize our database locally with

```
initdb - D data
```

Where data was the database cluster directory We then start the PostgreSQL server by running the code below with

```
pg_ctl -D data - l logfile start
```

Once done, we logged into the server locally with the command such as below with our username, where our username is our local device username and is provided upon running the above code Ex:

```
psql -U <your_username> postgres
```

Once that was done, we were not ready to create our database as we are now logged into the SQL terminal environment. The rest of the following code below will be conducted while connected to the PostgreSQL server. We create our yelp database and move into it with the following code

```
CREATE DATABASE yelp_dataset
\c yelp_dataset
```

Once that was set up, we were now ready to create our table schema on our server. We followed the layout of columns and datatypes for the CSV yelp datasets that we had sampled from and created the following required tables for them in the server.

```
    Business
    CREATE TABLE business (
    business_id VARCHAR(50) PRIMARY KEY,
    name TEXT,
    address TEXT,
    city TEXT,
    state CHAR(3),
    postal_code VARCHAR(10),
    latitude FLOAT,
    longitude FLOAT,
    stars FLOAT,
    review_count INTEGER,
    is_open BOOLEAN,
    attributes JSONB,
    categories TEXT,
    hours JSONB
);
```

And thus we can see that our database was set up. The rest of the data table creation can be seen in the appendix at the end of the report.

4.1.1 Data Insertion

To begin inserting our data, because of our utilization of CSV files instead of JSON files, there is a convenient method within SQL to insert the data of each CSV dataset into the PostgreSQL datatables within the SQL terminal ourselves. As long as the corresponding columns and datatypes are not violating any rules, then we can run the following commands to insert them while in our SQL terminal. The rest of the insertion for our checkin and tip dataset can be seen at the end of this report in our appendix.

```
Business
\COPY business (business_id, name, address, city, state, postal_code,
   latitude, longitude, stars, review_count, is_open, attributes,
    categories, hours) FROM 'sampled_businesses.csv' DELIMITER ',' CSV HEADER;
```

4.2 MongoDB Data Setup

In order to load our data into MongoDB, we first installed the MongoDB community edition and the MongoDB shell. Similarly to our PostgreSQL setup, we were using Jupyter locally through Anaconda. In the terminal in jupyter, we ran a local Mongo server and connected a Mongo client. To do this, in terminal (assuming homebrew setup) we ran the following commands:

```
brew services start mongodb-community@8.0
```

To verify that Mongo was running, we ran in terminal:

```
brew services list
```

We then ran the command to begin using mongodb:

```
mongosh
```

From here, we moved to the jupyter notebook and installed pymongo and ran the following import statements:

```
!pip install pymongo
import pymongo
from pymongo import MongoClient
import pandas as pd
```

We connected to the client and then tested that our connection was working: Input

```
client = MongoClient('localhost', 27017)
db = client['test']
db
```

Output

```
Database(MongoClient(host=['localhost:27017'], document_class=dict,
    tz_aware=False, connect=True), 'test')
```

After ensuring that we were connected, we were prepared to create our db and its collections and then insert our data. To create our db, we simply ran the following code. Naturally, unlike PostgreSQL, we did not need to define any schema.

```
yelpdb = client["yelp"]
business = yelpdb["business"]
review = yelpdb["review"]
user = yelpdb["user"]
checkin = yelpdb['checkin']
```

```
tip = yelpdb['tip']
```

4.2.1 Additional Data Transformations

Because we had sampled data for our PostgreSQL queries, we needed to convert that data back into JSON format for MongoDB insertion. So, we defined the following csv to json function which also took care of some of the encoding errors we experienced when trying to use pandas's standard csv to json function.

```
def csv_to_json(filename, header=None):
    data = pd.read_csv(filename, encoding='utf-8', encoding_errors='replace')

    # Convert the DataFrame into a list of dictionaries
    return data.to_dict('records')
```

4.2.2 MongoDB Data Insertion:

This is how we then inserted data into our collections:

```
business.insert_many(csv_to_json('sampled_businesses.csv'))
review.insert_many(csv_to_json('sampled_reviews.csv'))
checkin.insert_many(csv_to_json('sampled_checkins.csv'))
tip.insert_many(csv_to_json('sampled_tips.csv'))
user.insert_many(csv_to_json('sampled_users.csv'))
```

5 Task Selection

These tasks were chosen to have varying levels of complexity and utilize the different tables from the dataset. We also chose tasks that included a range of operations such as grouping, aggregation, joining and inserting to stress test our systems for comparison.

5.1 PostgreSQL Queries

The following are the three SQL queries we have chosen to analyze the Yelp dataset.

5.1.1 Average Review Count of Yelp Businesses by City

In this query we aim to aggregate and identify the average number of reviews businesses receive in a given city. Using the business table we ran the following query below to identify the cites with the highest average number of reviews.
Query:

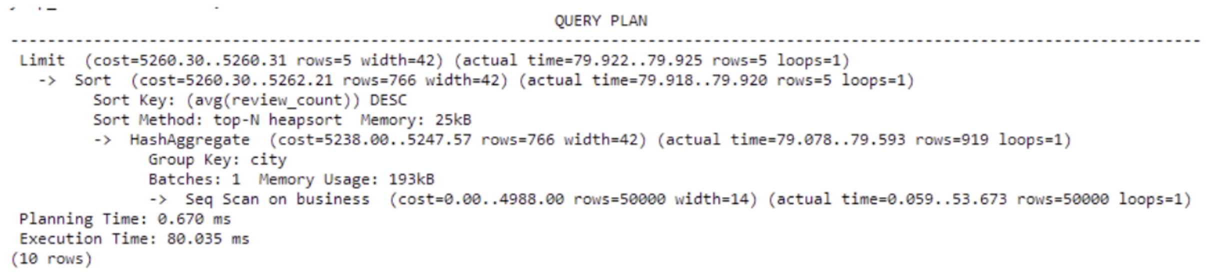
```
SELECT city, AVG(review_count) AS average_reviews
FROM business
```

```
GROUP BY city
ORDER BY average_reviews DESC
LIMIT 5;
```

Output:

```
       city     |   average_reviews
-------------+----------------------
 Tampa Palms | 280.0000000000000000
 Gwynedd     | 268.0000000000000000
 Manayunk    | 267.3333333333333333
 Frontenac   | 238.0000000000000000
 St Pete     | 237.0000000000000000
(5 rows)
```

We chose this as our first query as it was simple and it ran quickly and correctly would suggest we had property loaded the database into our code. We also chose this query to access PostgreSQL's performance when it comes to sorting, grouping, and aggregating. As seen in the query plan below, the query ran very quickly (80.035 milliseconds), confirming that the business table is populated correctly and verifying that key attributes such as names, stars, and review_count were properly retrieved. We see the query optimizer uses a quicksort which minimizes runtime by optimizing how rows are ordered before grouping and aggregating. The speed at which this query was run demonstrates PostgreSQL's efficiency in handling aggregate queries and no further optimization methods were deemed necessary.



```
                                                            QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=5260.30..5260.31 rows=5 width=42) (actual time=79.922..79.925 rows=5 loops=1)
   ->  Sort  (cost=5260.30..5262.21 rows=766 width=42) (actual time=79.918..79.920 rows=5 loops=1)
         Sort Key: (avg(review_count)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=5238.00..5247.57 rows=766 width=42) (actual time=79.078..79.593 rows=919 loops=1)
               Group Key: city
               Batches: 1  Memory Usage: 193kB
               ->  Seq Scan on business  (cost=0.00..4988.00 rows=50000 width=14) (actual time=0.059..53.673 rows=50000 loops=1)
 Planning Time: 0.670 ms
 Execution Time: 80.035 ms
(10 rows)
```

Figure 3: PostgreSQL Query 1 Performance

5.1.2 Count of Compliments by User

In this query, we aim to identify the users who give the most compliments. This provides insights into user behavior and community contributions. Using the user and tip relations, we ran the following query to join these datasets and calculate the total number of compliments given by each user.
Query

```
SELECT u.name AS user_name, SUM(t.compliment_count) AS total_compliments
FROM users u JOIN tip t ON u.user_id = t.user_id
GROUP BY u.user_id, u.name
```

```
    ORDER BY total_compliments DESC
    LIMIT 10;
```

Output:

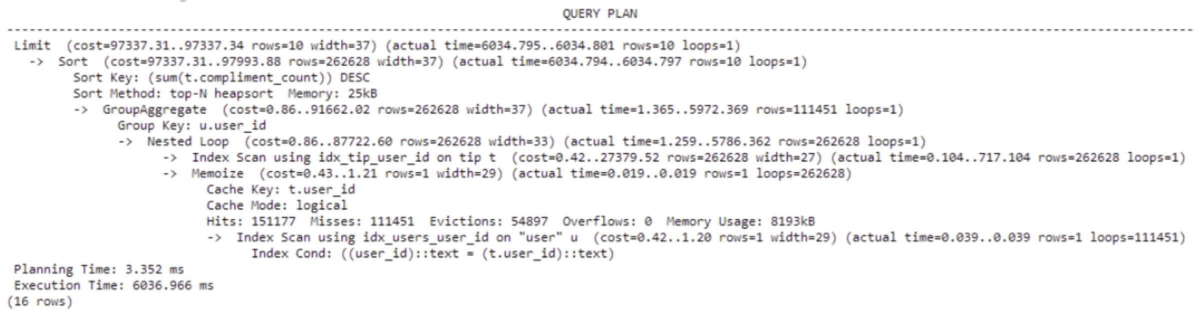
```
     user_name | total_compliments
    -----------+-------------------
     Michael   |                45
     Brian     |                27
     Sofia     |                26
     Rachel    |                25
     Steve     |                24
     Donna     |                24
     Angela    |                21
     Brittany  |                19
     Anna      |                18
     Christy   |                18
    (10 rows)
```

We selected the query to evaluate PostgreSQL's performance on inner joins because it involves combining two large tables, user and tip, and performing an aggregation operation on the compliment$_{c}ountattribute.Itprovidesaclearunderstandingof howPostgreSQLprocessesjoins, groupin$

Execution Time: 9545.032 milliseconds
Based on our analysis of the query plan, we determined that creating an index on the user_id column in both relations might make this query faster. This would allow PostgreSQL quickly locate the rows in the users and tip tables that need to be joined, reducing the overall scan time. We ran the following command to add the index:

```
    CREATE INDEX idx_users_user_id ON "user"(user_id);
    CREATE INDEX idx_tip_user_id ON tip(user_id);
```

Rerunning the query after this command gives the following query plan:



```
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=97337.31..97337.34 rows=10 width=37) (actual time=6034.795..6034.801 rows=10 loops=1)
   ->  Sort  (cost=97337.31..97993.88 rows=262628 width=37) (actual time=6034.794..6034.797 rows=10 loops=1)
         Sort Key: (sum(t.compliment_count)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  GroupAggregate  (cost=0.86..91662.02 rows=262628 width=37) (actual time=1.365..5972.369 rows=111451 loops=1)
               Group Key: u.user_id
               ->  Nested Loop  (cost=0.86..87722.60 rows=262628 width=33) (actual time=1.259..5786.362 rows=262628 loops=1)
                     ->  Index Scan using idx_tip_user_id on tip t  (cost=0.42..27379.52 rows=262628 width=27) (actual time=0.104..717.104 rows=262628 loops=1)
                     ->  Memoize  (cost=0.43..1.21 rows=1 width=29) (actual time=0.019..0.019 rows=1 loops=262628)
                           Cache Key: t.user_id
                           Cache Mode: logical
                           Hits: 151177  Misses: 111451  Evictions: 54897  Overflows: 0  Memory Usage: 8193kB
                           ->  Index Scan using idx_users_user_id on "user" u  (cost=0.42..1.20 rows=1 width=29) (actual time=0.039..0.039 rows=1 loops=111451)
                                 Index Cond: ((user_id)::text = (t.user_id)::text)
 Planning Time: 3.352 ms
 Execution Time: 6036.966 ms
(16 rows)
```

Figure 5: PostgreSQL Query 2 Performance After Index Creation

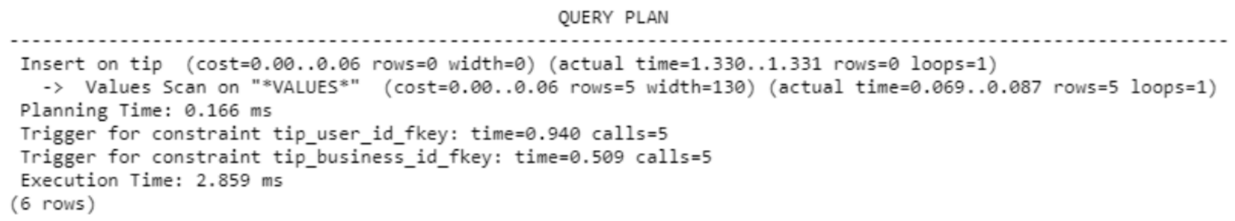
Execution time: 6036.966 milisecond

Before indexing, the query used a parallel sequential scan to retrieve rows from the tip table. In contrast, after generating indexes on both the user_id column in the user and tip tables, the query utilized index scans, improving the efficiency of row retrieval. This optimization led to a reduction in execution time, dropping from approximately 9.5 seconds without indexing to 6.03 seconds with the new indexes. The indexed query minimized disk I/O and memory usage by targeting relevant rows more effectively, reducing the need for a full table scan. While both query plans still involved similar sorting, grouping, and aggregation steps, the new indexing strategy alleviated some of the bottlenecks in row retrieval, highlighting the benefits of strategic indexing for improving overall query performance.

5.1.3 Tip Insertions

In this query, we aim to evaluate PostgreSQL's ability to handle data insertions by adding new records to the tip table. Using the following query below we will assess PostgreSQL's performance in managing insert operations, particularly with bulk inserts

```
INSERT INTO tip (user_id, business_id, text, date, compliment_count)
VALUES
('qVc8ODYU5SZjKXVBgXdI7w', 'TacYUYhU3HpLHF9Rs6fW2w', 'Great atmosphere and
    service!', '2024-12-10', 5),
('j14WgRoU_-2ZE1aw1dXrJg', 'RnExaICvIeXxFpbIKEqJsQ', 'Amazing food, highly
    recommend!', '2024-12-11', 3),
('2WnXYQFKOhXEoTxPtV2zvg', 'pjtjBeZC3gvmtIiIQt-DFA', 'Good for quick bites.',
    '2024-12-09', 2),
('SZDeASXq7o05mMNLshsdIA', 'OMl5pTUBVUjW_jvDKLvYtw', 'Could be better,
    service was slow.', '2024-12-08', 1),
('hA5lMy-EnncsH4JoR-hFGQ', 'MO-LTDfO843xaRtW0bx6jQ', 'Fantastic experience
    overall.', '2024-12-12', 7);
```



```
                                                    QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Insert on tip  (cost=0.00..0.06 rows=0 width=0) (actual time=1.330..1.331 rows=0 loops=1)
   ->  Values Scan on "*VALUES*"  (cost=0.00..0.06 rows=5 width=130) (actual time=0.069..0.087 rows=5 loops=1)
 Planning Time: 0.166 ms
 Trigger for constraint tip_user_id_fkey: time=0.940 calls=5
 Trigger for constraint tip_business_id_fkey: time=0.509 calls=5
 Execution Time: 2.859 ms
(6 rows)
```

Figure 6: PostgreSQL Query 3 Performance

Execution Time: 2.859 ms By executing a batch insertion, we can measure how well PostgreSQL manages disk I/O and transaction logging for write-heavy operations. This will be compared to MongoDB as seen later in this report. As seen in the query plan above, PostgreSQL is efficient in handling multiple inserts within a single transaction, with a planning time of 0.166 milliseconds and an execution time of 2.859 milliseconds. These results highlight PostgreSQL's ability to handle bulk data operations effectively while maintaining constraints, as seen in the triggers for foreign key checks on user_id and business_id. Due to the fast planning and execution times, it was deemed that no other optimization strategies were needed for this query.

5.2 MongoDB Queries

The following are the two MongoDB queries we have chosen to analyze the Yelp dataset. They were chosen as complements to our PostgreSQL queries listed above to make it easier to compare the two database systems later on in the Tool Comparison section.

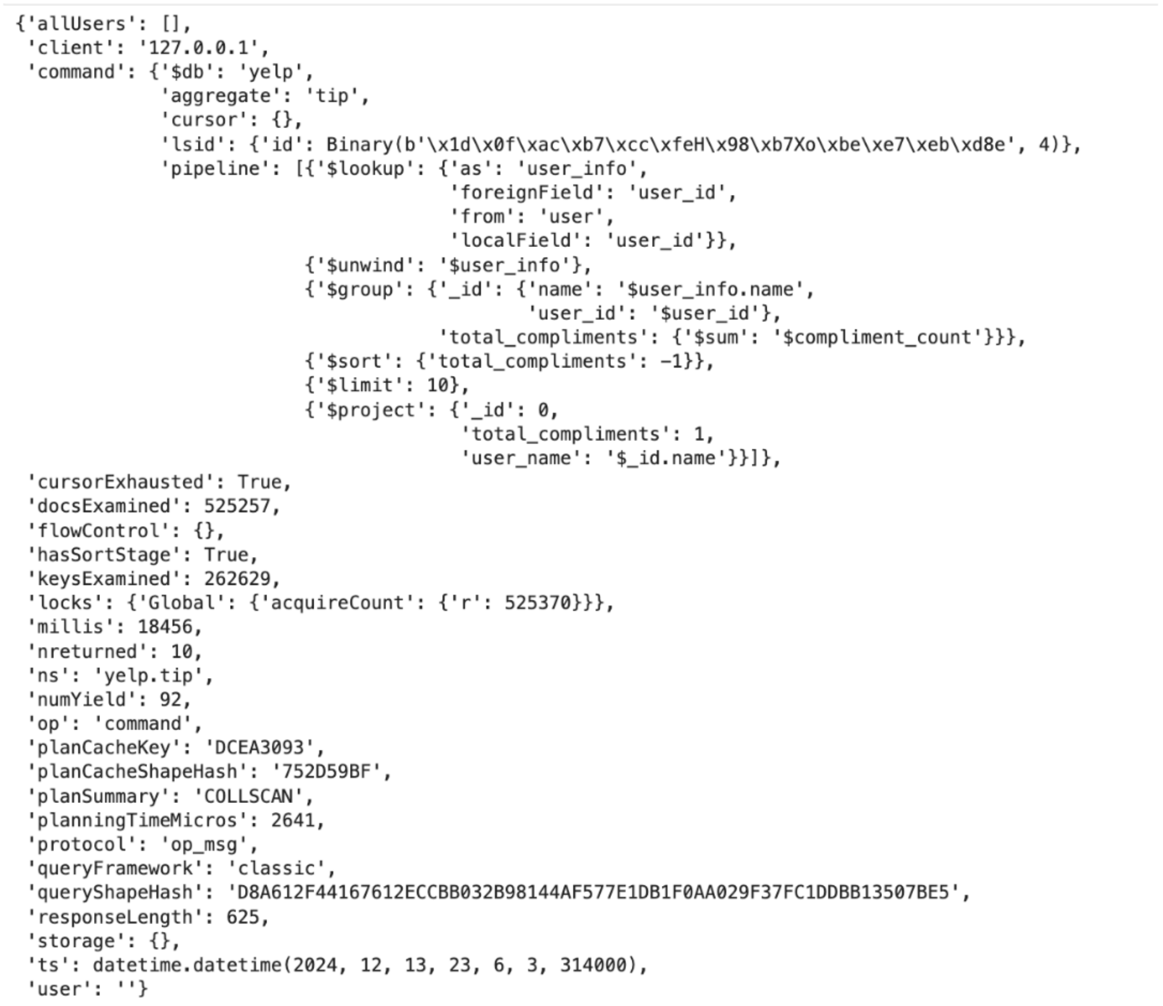
5.2.1 Count of Compliments by User

This query is the MongoDB equivalent of the second PostgreSQL query seen above. Using MongoDB we ran the following query below
Query:

```
db.tip.aggregate([
  {
    $lookup: {
      from: "users",
      localField: "user_id",
      foreignField: "user_id",
      as: "user_info"
    }
  },
  {
    $unwind: "$user_info"
  },
  {
    $group: {
      _id: { user_id: "$user_id", name: "$user_info.name" },
      total_compliments: { $sum: "$compliment_count" }
    }
  },
  {
    $sort: { total_compliments: -1 }
  },
  {
    $limit: 10
  },
  {
    $project: {
      _id: 0,
      user_name: "$_id.name",
      total_compliments: 1
    }
  }
]);
```

Output:

```
[{'total_compliments': 45, 'user_name': 'Michael'},
 {'total_compliments': 27, 'user_name': 'Brian'},
 {'total_compliments': 26, 'user_name': 'Sofia'},
  ]
```



```
{'allUsers': [],
 'client': '127.0.0.1',
 'command': {'$db': 'yelp',
             'aggregate': 'tip',
             'cursor': {},
             'lsid': {'id': Binary(b'\x1d\x0f\xac\xb7\xcc\xfeH\x98\xb7Xo\xbe\xe7\xeb\xd8e', 4)},
             'pipeline': [{'$lookup': {'as': 'user_info',
                                       'foreignField': 'user_id',
                                       'from': 'user',
                                       'localField': 'user_id'}},
                          {'$unwind': '$user_info'},
                          {'$group': {'_id': {'name': '$user_info.name',
                                              'user_id': '$user_id'},
                                      'total_compliments': {'$sum': '$compliment_count'}}},
                          {'$sort': {'total_compliments': -1}},
                          {'$limit': 10},
                          {'$project': {'_id': 0,
                                        'total_compliments': 1,
                                        'user_name': '$_id.name'}}]},
 'cursorExhausted': True,
 'docsExamined': 525257,
 'flowControl': {},
 'hasSortStage': True,
 'keysExamined': 262629,
 'locks': {'Global': {'acquireCount': {'r': 525370}}},
 'millis': 18456,
 'nreturned': 10,
 'ns': 'yelp.tip',
 'numYield': 92,
 'op': 'command',
 'planCacheKey': 'DCEA3093',
 'planCacheShapeHash': '752D59BF',
 'planSummary': 'COLLSCAN',
 'planningTimeMicros': 2641,
 'protocol': 'op_msg',
 'queryFramework': 'classic',
 'queryShapeHash': 'D8A612F44167612ECCBB032B98144AF577E1DB1F0AA029F37FC1DDBB13507BE5',
 'responseLength': 625,
 'storage': {},
 'ts': datetime.datetime(2024, 12, 13, 23, 6, 3, 314000),
 'user': ''}
```

Figure 7: MongoDB Query 1 Execution

Execution time: 18456 milliseconds This query had an execution time in MongoDB of 18456 milliseconds and a planning time of 2.641 milliseconds (2641 micros). The query only worked if indices were added for the user and tip collections, otherwise the query ran for over an hour, likely due to the excessive time spent executing the lookup. In addition, frustratingly, we were unable to utilize MongoDB's .explain('executionStats') function (equivalent to PostgreSQL's EXPLAIN ANALYZE) despite significant debugging. In order to analyze the execution and planning time, we tested using the .command() function with the 'explain' parameter set to True, but this failed to provide us with the detailed execution stats we wanted. So, we made use of MongoDB's Database profiler, which keeps track of all commands run in Mongo and this allowed us to analyze execution and planning time.

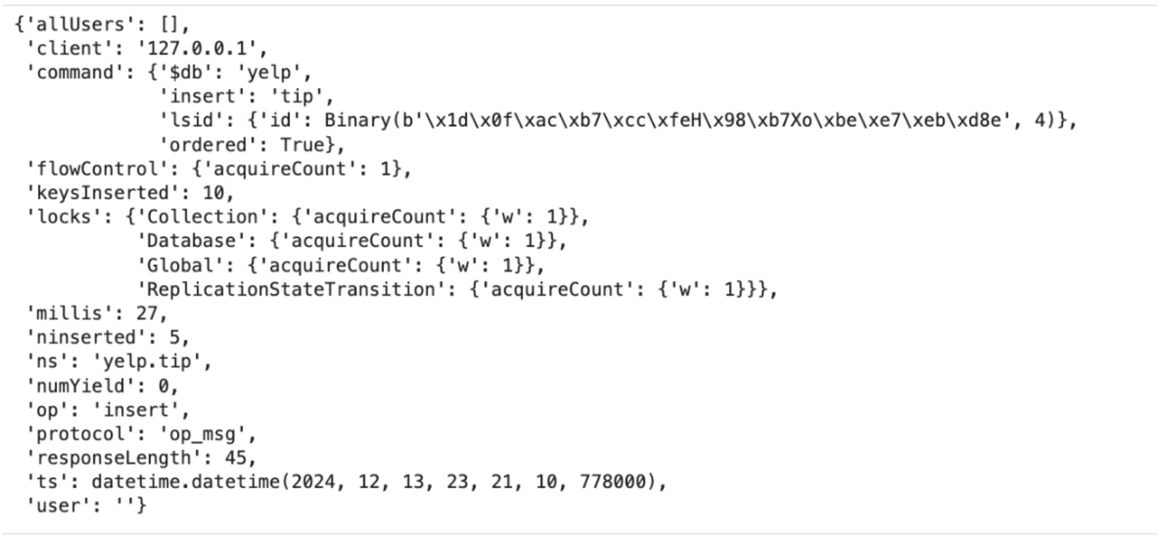
5.2.2 Tip Insertion

This query is the MongoDB equivalent of the third PostgreSQL query seen above. Using MongoDB we ran the following query below:
Query:

```
db.tip.insertMany([
  {
    user_id: "qVc8ODYU5SZjKXVBgXdI7w",
    business_id: "TacYUYhU3HpLHF9Rs6fW2w",
    text: "Great atmosphere and service!",
    date: "2024-12-10",
    compliment_count: 5
  },
  {
    user_id: "j14WgRoU-2ZE1aw1dXrJg",
    business_id: "RnExaICvIeXxFpbIKEqJsQ",
    text: "Amazing food, highly recommend!",
    date: "2024-12-11",
    compliment_count: 3
  },

]);
```

Execution Log:



```
{'allUsers': [],
 'client': '127.0.0.1',
 'command': {'$db': 'yelp',
             'insert': 'tip',
             'lsid': {'id': Binary(b'\x1d\x0f\xac\xb7\xcc\xfeH\x98\xb7Xo\xbe\xe7\xeb\xd8e', 4)},
             'ordered': True},
 'flowControl': {'acquireCount': 1},
 'keysInserted': 10,
 'locks': {'Collection': {'acquireCount': {'w': 1}},
           'Database': {'acquireCount': {'w': 1}},
           'Global': {'acquireCount': {'w': 1}},
           'ReplicationStateTransition': {'acquireCount': {'w': 1}}},
 'millis': 27,
 'ninserted': 5,
 'ns': 'yelp.tip',
 'numYield': 0,
 'op': 'insert',
 'protocol': 'op_msg',
 'responseLength': 45,
 'ts': datetime.datetime(2024, 12, 13, 23, 21, 10, 778000),
 'user': ''}
```

Figure 8: MongoDB Query 2 Execution Log

Execution time: 27 milliseconds We chose this query to evaluate MongoDB's performance in handling bulk insertions using insertMany. This will later be compared to PostgreSQL in the Tool Comparison section. From the execution log above we see an execution time of 27 milliseconds. This highlights MongoDB's ability to manage bulk writes efficiently for schema-less datasets. Note that this query was run after the previous query and thus kept the index on Tips we had previously generated. Due to the speed at which this query was executed, no further optimization strategies were deemed necessary.

6 Tool Comparison

6.1 Installation and Setup

From an installation and setup perspective, both tools have user-friendly installers and comprehensive documentation, but PostgreSQL may require more initial configuration to optimize for specific workloads, especially in enterprise settings. MongoDB's setup feels more lightweight, especially for beginners, due to its flexible schema model, which doesn't demand predefined tables or rigid structures. Learning PostgreSQL involves understanding relational database theory and SQL syntax, which may be more challenging for newcomers. MongoDB's learning curve is gentler, with developers only needing familiarity with JSON-like structures and basic query operators.

6.2 Query Comparison

6.2.1 Count of Compliments by User

This query tested performance with respect to relational joins. In PostgreSQL, the query utilized JOIN, grouping, and aggregation, performing efficiently after indexing and reducing execution time from approximately 9.5 seconds to 5.8 seconds. In MongoDB, the equivalent query is involved in lookup stage for the join, followed by unwind, group, sort, and limit stages in an aggregation pipeline. The MOngoDB query was slower, taking 11.2 seconds to execute with indexing. Overall, PostgreSQL handled this query better due to its relational database design and indexing capabilities, while MongoDB, though more readable and adaptable, struggled with the computational efficiency of relational joins.

6.2.2 Tip Insertions

This query tested bulk insert performance. PostgreSQL handled batch inserts with foreign key constraints, completing in 1.27 milliseconds. MongoDB, using insertMany, completed in 27 milliseconds. Both databases handled bulk insertions effectively, but PostgreSQLwas faster and maintained data integrity through constraints. When we first ran the query there was a typo in the business_id column that was "caught" by PostgreSQL. MongoDB which lacks constraint enforcement would not have done the same.

6.3 Fitness, Ergonomics and Performance

Performance wise, PostgreSQL did well in relational queries and aggregations, improving execution times with indexing as seen in the Count of Compliments by User section above. The batch insertions were also efficient, completing in 2.859 milliseconds while enforcing constraints in comparison to MongoDB's 27 milliseconds without enforcing constraints. MongoDB, while effective for bulk insertions with insertMany, lacked relational optimization tools like native joins, leading to longer execution times for tasks like the "Count of Compliments by User" query.

In terms of ergonomics and fitness, PostgreSQL's initial setup and stricter schema requirements posed challenges, particularly during technical configuration. MongoDB was deemed more ergonomic due its ease of setup and JSON-like query syntax which was more developer-friendly for those familiar with modern programming paradigms. Although MongoDB's schema-less design provided more accessibility and simplification for insertions of different types of data, it was not more efficient in query times. PostgreSQL was deemed more fit for relational tasks like with our "Count of Compliments by User" query, as its flatten structure, foreign key support, query optimization plan, helped speed up execution times.

7 Reflection

If someone else were to consider these tools for similar tasks and/or workloads we would recommend considering the team's familiarity with relational versus NoSQL databases. PostgreSQL might be overwhelming for a team accustomed to agile development and frequent schema changes, whereas MongoDB may feel too lenient for a team used to strict data integrity and ACID compliance. Ultimately, if their tasks demand transactional accuracy and complex data modeling, PostgreSQL is the more reliable option. If agility, scalability, and ease of use are paramount, MongoDB will serve their needs better.

8 Appendix

8.1 Additional Import Code

```
    Reviews
\COPY review (review_id, user_id, business_id, stars, useful, funny, cool, text,
    date)FROM './sampled_reviews.csv' DELIMITER ',' CSV HEADER;

User
\COPY "user" (
    user_id, name, review_count, yelping_since, useful, funny, cool, elite,
        friends, fans, average_stars,
    compliment_hot, compliment_more, compliment_profile, compliment_cute,
        compliment_list,
    compliment_note, compliment_plain, compliment_cool, compliment_funny,
        compliment_writer,
    compliment_photos
)
FROM './sampled_users.csv'
DELIMITER ','
CSV HEADER;
```

8.2 Additional Data Table Creation Code

```
    Reviews
    CREATE TABLE review (
        review_id CHARACTER VARYING(50) PRIMARY KEY,
        user_id CHARACTER VARYING(50) REFERENCES "user" (user_id),
        business_id CHARACTER VARYING(50) REFERENCES business (business_id),
        stars NUMERIC(2, 1),
        useful INTEGER,
        funny INTEGER,
        cool INTEGER,
        text TEXT,
        date TIMESTAMP WITHOUT TIME ZONE
    );


    User
    CREATE TABLE "user" (
        user_id CHARACTER VARYING(50) PRIMARY KEY,
        name CHARACTER VARYING(100),
        review_count INTEGER,
        yelping_since TIMESTAMP WITHOUT TIME ZONE,
        useful INTEGER,
        funny INTEGER,
        cool INTEGER,
        elite TEXT,
        friends TEXT,
        fans INTEGER,
        average_stars NUMERIC(3, 2),
        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
    );


    Checkin
    CREATE TABLE checkin (
        business_id VARCHAR(50) REFERENCES business(business_id),
        date TEXT
    );


    Tips
    CREATE TABLE tip (
        user_id CHARACTER VARYING(50) REFERENCES "user" (user_id),
        business_id CHARACTER VARYING(50) REFERENCES business (business_id),
        text TEXT,
        date TIMESTAMP WITHOUT TIME ZONE,
        compliment_count INTEGER
    );
```

8.3 Additional Data Table Insertion Code

```
Checkin
\COPY checkin (business_id, date) FROM './sampled_checkins.csv' DELIMITER ','
    CSV HEADER;

Tips
\COPY tip (user_id, business_id, text, date, compliment_count) FROM
    './sampled_tips.csv' DELIMITER ',' CSV HEADER;
```

8.4 Additional MongoDB Collections:

```
Review Collection: (.find_one())
{'_id': ObjectId('675c0b768f3903b94a195fc2'),
 'review_id': 'BiTunyQ73aT9WBnpR9DZGw',
 'user_id': 'OyoGAe7OKpv6SyGZT5g77Q',
 'business_id': '7ATYjTIgM3jUlt4UM3IypQ',
 'stars': 5.0,
 'useful': 1,
 'funny': 0,
 'cool': 1,
 'text': "I've taken a lot of spin classes over the years, and nothing
    compares to the classes at Body Cycle. From the nice, clean space and
    amazing bikes, to the welcoming and motivating instructors, every class
    is a top notch work out.\n\nFor anyone who struggles to fit workouts in,
    the online scheduling system makes it easy to plan ahead (and there's no
    need to line up way in advanced like many gyms make you do).\n\nThere is
    no way I can write this review without giving Russell, the owner of Body
    Cycle, a shout out. Russell's passion for fitness and cycling is so
    evident, as is his desire for all of his clients to succeed. He is
    always dropping in to classes to check in/provide encouragement, and is
    open to ideas and recommendations from anyone. Russell always wears a
    smile on his face, even when he's kicking your butt in class!",
 'date': '2012-01-03 15:28:18'}


User Collection
{'_id': ObjectId('675c0f628f3903b94a41b89b'),
 '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': 'NSCy54eWehBJyZdG2iE84w, QBXAIu1Vm-nKXUwBqZ5H7Q...
    VOP2NjZivSO2uVYFhXMkCg, j2KnquMwMfs3VmQSzYknLw,
 '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}


Checkin Collection:
{'_id': ObjectId('675c0e188f3903b94a3d0bbf'),
 'business_id': '---kPU91CF4Lq2-WlRu9Lw',
 'date': '2020-03-13 21:10:56, 2020-06-02 22:18:06, 2020-07-24 22:42:27,
    2020-10-24 21:36:13, 2020-12-09 21:23:33, 2021-01-20 17:34:57,
    2021-04-30 21:02:03, 2021-05-25 21:16:54, 2021-08-06 21:08:08,
    2021-10-02 15:15:42, 2021-11-11 16:23:50'}



Tip Collection
{'_id': ObjectId('675cc2a195910dfa8faf4ec0'),
 'user_id': 'NBN4MgHP9D3cw--SnauTkA',
 'business_id': 'QoezRbYQncpRqyrLH6Iqjg',
 'text': 'They have lots of good deserts and tasty cuban sandwiches',
 'date': '2013-02-05 18:35:10',
 'compliment_count': 0}
```