Project CheckPoint Yelp

Michelle Dong, Kelly He, Claudia Wu, Emma Ye

Dataset Selection

Describe the dataset: https://www.yelp.com/dataset/documentation/main

The dataset is split into multiple collections including business, review, user_info, and tip. The business collection provides information about the business like the name, location, time, and some review information . The review collection shows the business, rating, a description, and date of the review by the user. The user_info provides information on the users and their activity on yelp and when they started their account. Finally, the tip collection shows the users tips along with other information for the business. With the current variables, the data allows others to understand the business and user reviews.

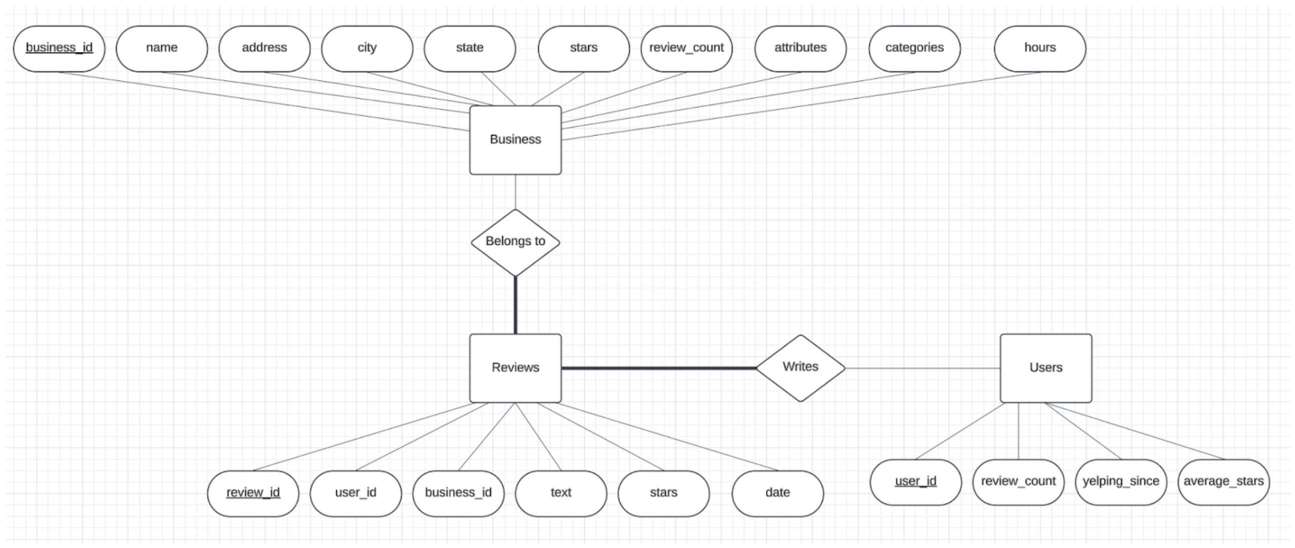
Describe any sampling or truncation plans

The main purpose of the truncation plans was to reduce the data set's size and ensure our process systems could complete the task without crashing. For example, the business.json was the smallest. Still, the latitude and longitude might be duplicated because we already know the business's city and state of the business so we decided to remove that. The rest of the data set was significantly larger because we removed all votes received for useful, funny, and cool because it is a bit redundant to manage. We also removed compliment_information for the same reason. However, we kept average_stars to keep the core message from all reviews. Lastly, we rearranged the rows by excluding the unmatched user IDs between review.json and user_info.json to reduce data size and maintain data consistency.

Collections and their attributes:

- Business: Business_id (primary key, Name, Address, City, State, Stars, Review_count, Attributes, Categories, Hours
- Review (6gb): Review_id (primary), User_id (foreign key to Business business_id), Business_id (foreign key to Business business_id), Text , Stars, Date
- User_info (3gb): User_id (primary), Review_count, Yelping_since, Average_stars
- Tip: Text, Date, Compliment_count, Business_id, User_id

ER diagram



System and Database Setup

The Postgres loading process starts with initializing a virtual environment, creating a database user, and establishing a database to store the business, user, and review data. Using a local setup on a personal computer, the conda environment manager is employed to install PostgreSQL. After initializing a database directory and starting the PostgreSQL service, a non-superuser account is created along with a database called 'myinner_db'.

```
dbname = "myinner_db"
user = "mynonsuperuser"

 try:
     conn = psycopg2.connect(f"dbname={dbname} user={user}")
     cur = conn.cursor()
```

Batch Insertion Logic

Data loading into PostgreSQL is accomplished through Python scripts utilizing the psycopg2 library. The scripts read JSON files, each representing different types of business-related data, and execute SQL commands to create and populate tables in the database. For example, the load_business_json_to_psql, load_review_json_to_psql, and load_user_json_to_psql functions create the tables if it does not already exist,

```
cur.execute(f'''
         CREATE TABLE IF NOT EXISTS "{table_name}" (
            business_id VARCHAR(22) PRIMARY KEY,
            name TEXT NOT NULL,
            address TEXT,
            city TEXT, ...
        );
    ''')
```

then perform data insertions of batches with size 1000 from the JSON file as well as the remaining entries under batch size of 1000. This algorithm reduces the number of round trips between the Python application and the database server and improves the efficiency of handling large datasets.

```
data_to_insert = []

        for line_number, line in enumerate(file, 1):
            try:
                data = json.loads(line.strip())

                categories_list = data.get('categories', '').split(', ') if isinstance(data.get('categories'), str) else []

                data_to_insert.append({
                    'business_id': data.get('business_id'),
                    'name': data.get('name'),
                    'address': data.get('address'),
                    'city': data.get('city'),
                    ... })

                if len(data_to_insert) >= 1000:
                    cur.executemany(f'''
                        INSERT INTO "{table_name}" (...

if data_to_insert:
            cur.executemany(f'''
                INSERT INTO "{table_name}" (...
```

Foreign Key Constraints and Data Integrity
The **business_id** column in the ‘Review’ table references the **business_id** in the ‘Business’ table. It ensures that every review is associated with an existing business. This constraint ensures that reviews cannot exist without a corresponding business. Similarly, the **user_id** column references the **user_id** in the ‘User’ table, linking each review to a specific user. This link ensures that every review is written by a user who exists in the database. The **ON DELETE CASCADE** directive ensures that if a referenced ‘User’ or ‘Business’ is deleted, all corresponding reviews are also automatically deleted.

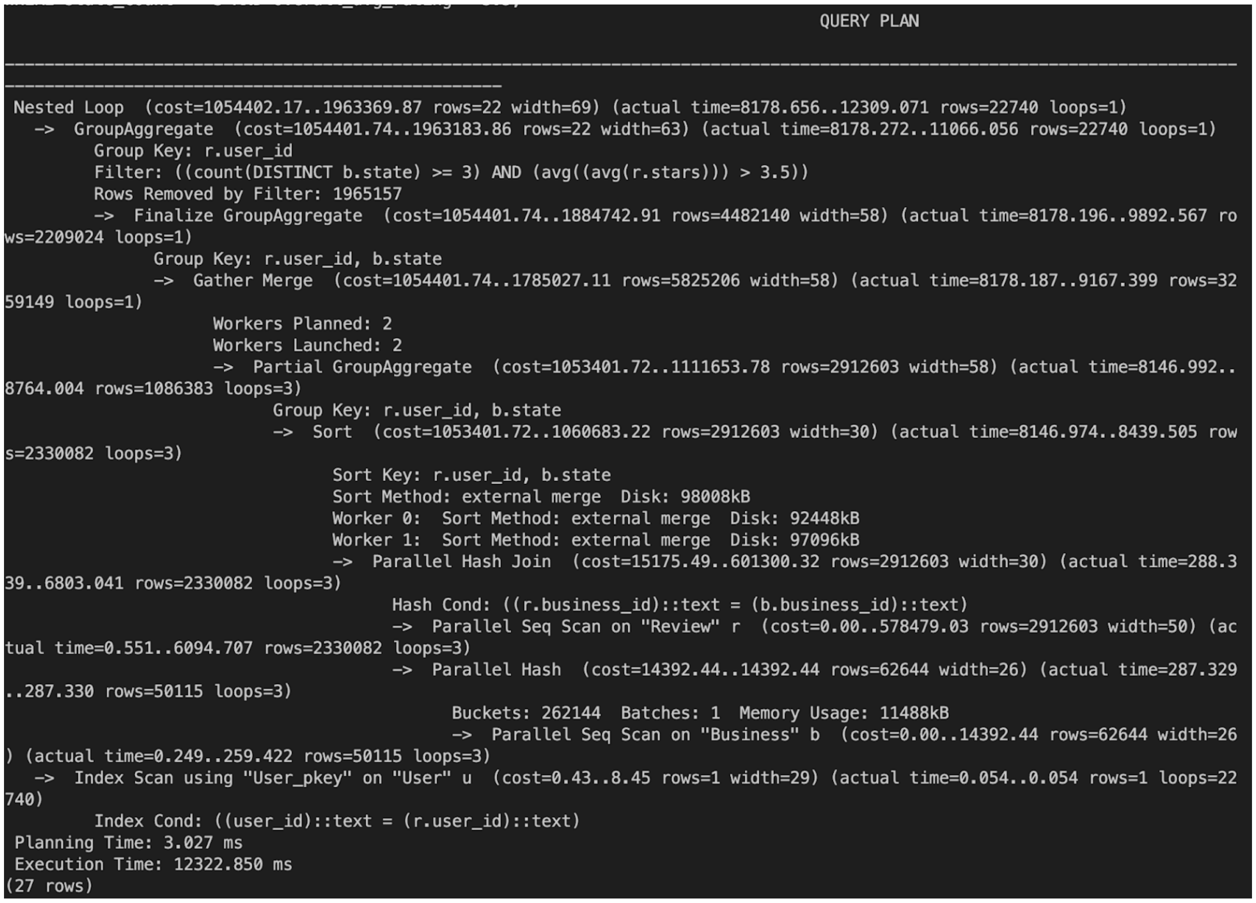
```
cur.execute(f'''
    CREATE TABLE IF NOT EXISTS "{table_name}" (
        review_id VARCHAR(22) PRIMARY KEY,
        user_id VARCHAR(22) REFERENCES "User" (user_id) ON DELETE CASCADE,
        business_id VARCHAR(22) REFERENCES "Business" (business_id) ON DELETE CASCADE,
        text TEXT,
        stars INTEGER,
        date DATE
    );
""")
```

PostgreSQL Tasks and Queries

Task / Query 1

The SQL query we chose is designed to find users who have written reviews for businesses in 3+ states and have an average review star rating of more than 3.5. The purpose is to track user engagement and identify users to qualify for feedback. Creating this query will help us understand relational databases like PostgreSQL as it combines complex joins, aggregations, and group operations.

```
WITH user_review_states AS (
    SELECT r.user_id, b.state, AVG(r.stars) AS avg_rating
        FROM "Review" r
        JOIN "Business" b ON r.business_id = b.business_id
        GROUP BY r.user_id, b.state
),
user_states_count AS (
    SELECT user_id, COUNT(DISTINCT state) AS state_count, AVG(avg_rating) AS overall_avg_rating
        FROM user_review_states
        GROUP BY user_id
)
SELECT u.user_id, u.name, state_count, overall_avg_rating
    FROM
        user_states_count usc
    JOIN
        "User" u ON usc.user_id = u.user_id
    WHERE
        state_count >= 3 AND overall_avg_rating > 3.5;
```

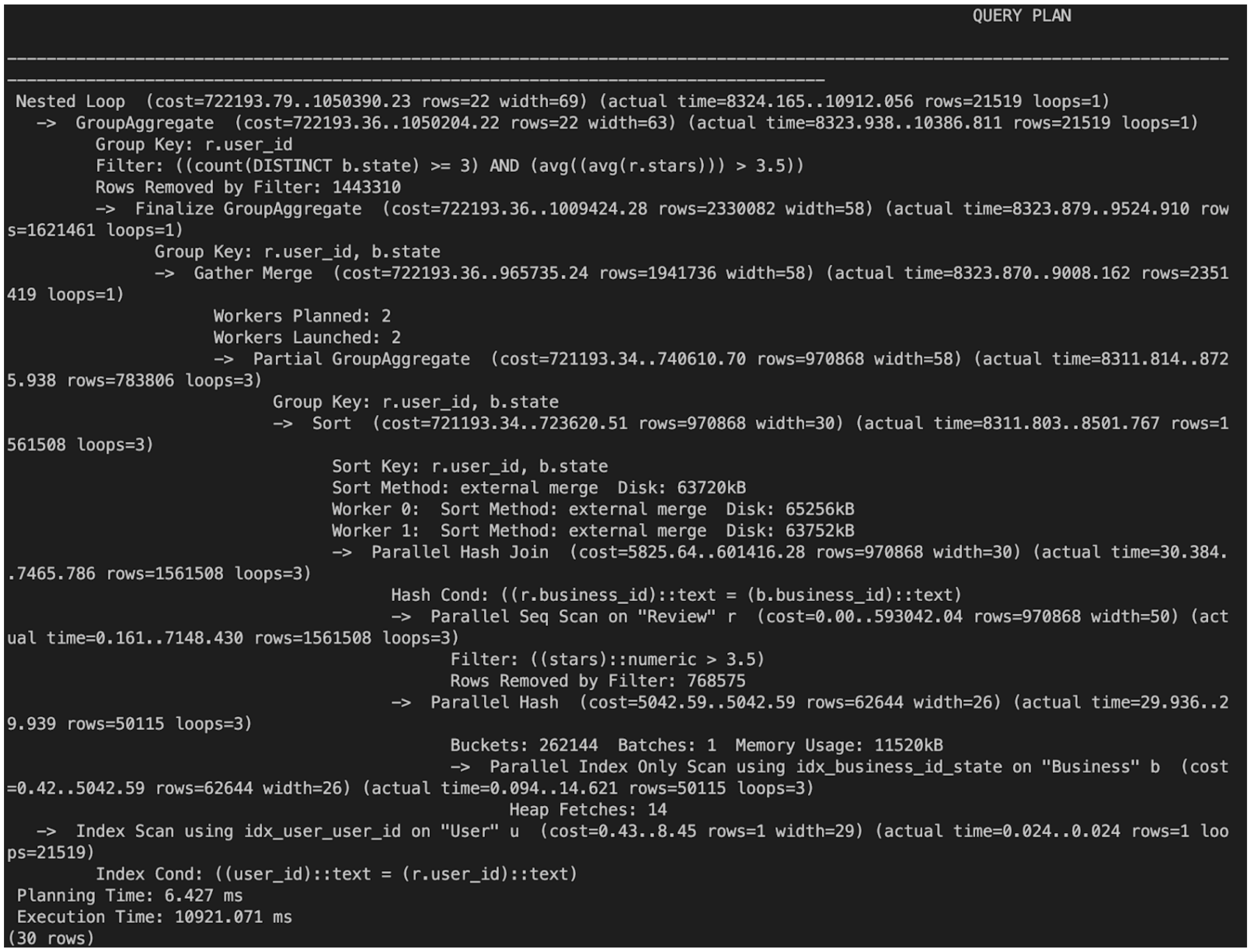


```
                                                                                   QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------
 Nested Loop  (cost=1054402.17..1963369.87 rows=22 width=69) (actual time=8178.656..12309.071 rows=22740 loops=1)
   ->  GroupAggregate  (cost=1054401.74..1963183.86 rows=22 width=63) (actual time=8178.272..11066.056 rows=22740 loops=1)
         Group Key: r.user_id
         Filter: ((count(DISTINCT b.state) >= 3) AND (avg((avg(r.stars))) > 3.5))
         Rows Removed by Filter: 1965157
         ->  Finalize GroupAggregate  (cost=1054401.74..1884742.91 rows=4482140 width=58) (actual time=8178.196..9892.567 ro
ws=2209024 loops=1)
               Group Key: r.user_id, b.state
               ->  Gather Merge  (cost=1054401.74..1785027.11 rows=5825206 width=58) (actual time=8178.187..9167.399 rows=32
59149 loops=1)
                     Workers Planned: 2
                     Workers Launched: 2
                     ->  Partial GroupAggregate  (cost=1053401.72..1111653.78 rows=2912603 width=58) (actual time=8146.992..
8764.004 rows=1086383 loops=3)
                           Group Key: r.user_id, b.state
                           ->  Sort  (cost=1053401.72..1060683.22 rows=2912603 width=30) (actual time=8146.974..8439.505 row
s=2330082 loops=3)
                                 Sort Key: r.user_id, b.state
                                 Sort Method: external merge  Disk: 98008kB
                                 Worker 0:  Sort Method: external merge  Disk: 92448kB
                                 Worker 1:  Sort Method: external merge  Disk: 97096kB
                                 ->  Parallel Hash Join  (cost=15175.49..601300.32 rows=2912603 width=30) (actual time=288.3
39..6803.041 rows=2330082 loops=3)
                                       Hash Cond: ((r.business_id)::text = (b.business_id)::text)
                                       ->  Parallel Seq Scan on "Review" r  (cost=0.00..578479.03 rows=2912603 width=50) (ac
tual time=0.551..6094.707 rows=2330082 loops=3)
                                       ->  Parallel Hash  (cost=14392.44..14392.44 rows=62644 width=26) (actual time=287.329
..287.330 rows=50115 loops=3)
                                             Buckets: 262144  Batches: 1  Memory Usage: 11488kB
                                             ->  Parallel Seq Scan on "Business" b  (cost=0.00..14392.44 rows=62644 width=26
) (actual time=0.249..259.422 rows=50115 loops=3)
   ->  Index Scan using "User_pkey" on "User" u  (cost=0.43..8.45 rows=1 width=29) (actual time=0.054..0.054 rows=1 loops=22
740)
         Index Cond: ((user_id)::text = (r.user_id)::text)
 Planning Time: 3.027 ms
 Execution Time: 12322.850 ms
(27 rows)
```

We began with apply the filter count(DISTINCT b.state) >= 3 and avg(avg(r.stars)) > 3.5. The query uses Gather Merge, is high cost (over 1 million) as it is processing a large amount of data with heavy joins and aggregation. In order to optimize the query, we then tried to create indexes on user_id, business_id, and state as these are the key fields that we are performing filtering and joining operations. Through these indexes, we can avoid PostgreSQL from doing Nested Loop, and instead choose Hash Join or Merge Join. Finally, we adjust the query to perform an Index Scan on the "User" table, which efficiently retrieves user data by utilizing the indexed user_id.

Next we try to tackle the runtime of the query, we plan to create indexes to optimize the runtime for joining and filtering operations. We added the following indexes: an index on business_id and state column in "the Business" table (idx_business_id_state), an index on user_id and business_id column in "Review" table (idx_review_user_business_id), and an index on user_id column in "User" table (idx_user_user_id). These optimizations reduce disk I/O, improve row lookup efficiency, and overall contribute to better query performance.

```sql
WITH user_review_states AS (
    SELECT r.user_id, b.state, AVG(r.stars) AS avg_rating
        FROM "Review" r
        JOIN "Business" b ON r.business_id = b.business_id
        WHERE r.stars > 3.5
        GROUP BY r.user_id, b.state
),
user_states_count AS (
    SELECT user_id, COUNT(DISTINCT state) AS state_count, AVG(avg_rating) AS overall_avg_rating
        FROM user_review_states
        GROUP BY user_id
        HAVING COUNT(DISTINCT state) >= 3 AND AVG(avg_rating) > 3.5
)
SELECT u.user_id, u.name, state_count, overall_avg_rating
    FROM
        user_states_count usc
    JOIN
        "User" u ON usc.user_id = u.user_id;
```



```
                                                                                                                                   QUERY PLAN

----------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
 Nested Loop  (cost=722193.79..1050390.23 rows=22 width=69) (actual time=8324.165..10912.056 rows=21519 loops=1)
   ->  GroupAggregate  (cost=722193.36..1050204.22 rows=22 width=63) (actual time=8323.938..10386.811 rows=21519 loops=1)
         Group Key: r.user_id
         Filter: ((count(DISTINCT b.state) >= 3) AND (avg((avg(r.stars))) > 3.5))
         Rows Removed by Filter: 1443310
         ->  Finalize GroupAggregate  (cost=722193.36..1009424.28 rows=2330082 width=58) (actual time=8323.879..9524.910 row
s=1621461 loops=1)
               Group Key: r.user_id, b.state
               ->  Gather Merge  (cost=722193.36..965735.24 rows=1941736 width=58) (actual time=8323.870..9008.162 rows=2351
419 loops=1)
                     Workers Planned: 2
                     Workers Launched: 2
                     ->  Partial GroupAggregate  (cost=721193.34..740610.70 rows=970868 width=58) (actual time=8311.814..872
5.938 rows=783806 loops=3)
                           Group Key: r.user_id, b.state
                           ->  Sort  (cost=721193.34..723620.51 rows=970868 width=30) (actual time=8311.803..8501.767 rows=1
561508 loops=3)
                                 Sort Key: r.user_id, b.state
                                 Sort Method: external merge  Disk: 63720kB
                                 Worker 0:  Sort Method: external merge  Disk: 65256kB
                                 Worker 1:  Sort Method: external merge  Disk: 63752kB
                                 ->  Parallel Hash Join  (cost=5825.64..601416.28 rows=970868 width=30) (actual time=30.384.
.7465.786 rows=1561508 loops=3)
                                       Hash Cond: ((r.business_id)::text = (b.business_id)::text)
                                       ->  Parallel Seq Scan on "Review" r  (cost=0.00..593042.04 rows=970868 width=50) (act
ual time=0.161..7148.430 rows=1561508 loops=3)
                                             Filter: ((stars)::numeric > 3.5)
                                             Rows Removed by Filter: 768575
                                       ->  Parallel Hash  (cost=5042.59..5042.59 rows=62644 width=26) (actual time=29.936..2
9.939 rows=50115 loops=3)
                                             Buckets: 262144  Batches: 1  Memory Usage: 11520kB
                                             ->  Parallel Index Only Scan using idx_business_id_state on "Business" b  (cost
=0.42..5042.59 rows=62644 width=26) (actual time=0.094..14.621 rows=50115 loops=3)
                                                   Heap Fetches: 14
   ->  Index Scan using idx_user_user_id on "User" u  (cost=0.43..8.45 rows=1 width=29) (actual time=0.024..0.024 rows=1 loo
ps=21519)
         Index Cond: ((user_id)::text = (r.user_id)::text)
 Planning Time: 6.427 ms
 Execution Time: 10921.071 ms
(30 rows)
```

Finally, as we predict, the performance of the updated query has improved. The updated query has a 10,921 ms execution time compared to 12,322 ms in the original query.

Future Plan

Another database that we plan on using is MongoDB. One way that we plan on comparing Postgres to other databases by looking at performance (ie. execution time). From that, we can see how well MongoDB optimizes itself and use cases of complex joins. The other way is by looking at scalability. PostgreSQL can do scaling but performance may go down, and MongoDB can perform scaling. With the performance and scalability, we can also evaluate how complex the queries are. Finally, we can look into flexibility. While MongoDB is known for being good for messier data, we can look at the transformations needed for PostgresQL.