Spilling the Tampa, FL Tea: A Comparative Analysis of PostgreSQL vs. MongoDB

Riyya Ahmed, Mariajose Garcia Morones, Anoutsala Hanmonty, Ashley Pun

Dataset Selection

Dataset Description

For this project, we decided to use the Yelp dataset, which contains multiple JSON files: *business.json*, *review.json*, and *user.json*. Each document of the *business.json* provides details about a specific business, while *user.json* contains information about individual users. The *review.json* file includes data about reviews, along with `user_id` and `business_id` values to link reviews to users and businesses, respectively.

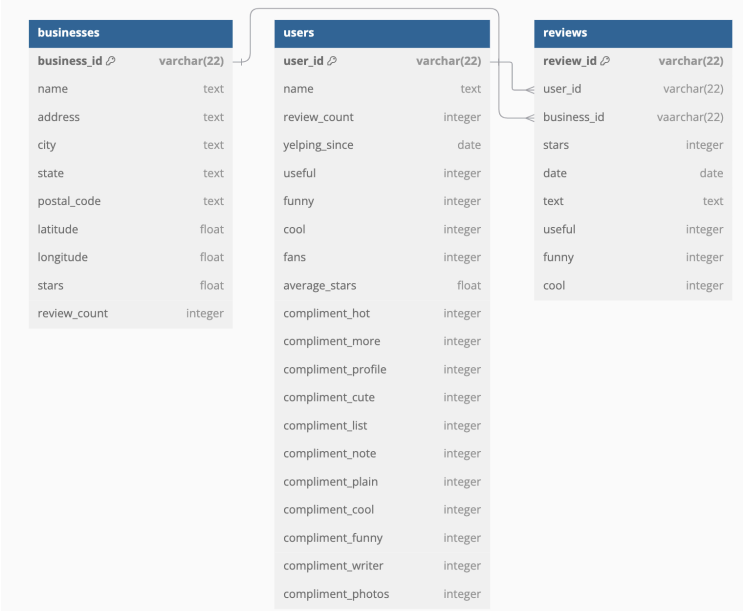
To focus our analysis, we excluded certain columns that were not relevant to our exploration. Specifically, in the **businesses** table, we excluded the `attributes` object (which maps business attributes to values), the `categories` array (which lists business categories), and the `hours` object (which is an object of key day-to-value hours). For the **users** table, we omitted the `friends` array (which lists `user_id` s of friends) since we're not interested in exploring the relationship between users and friends. View ER Diagram section to see the features we have left.

Sampling and Truncating Plans

After analyzing the Yelp **businesses**, **reviews**, and **users** tables, we decided not to sample or truncate our data, as more information would allow us to draw more conclusions. However, we did make a variety of virtual views in PostgreSQL to make our data easier to access and query. Although we knew materialized views would've been helpful since our data isn't consistently updated, materialized views didn't load on some of our local machines. View the ER Diagram section to see the full list of relations.

We wanted to narrow our focus on Florida since we noticed it had significantly more data than other states. Then, we further drilled down and identified the top three cities with the highest number of reviews: Tampa, Clearwater, and St. Petersburg, which became the focus of one of our PostgreSQL queries. Tampa had the most reviews making it the primary focus for most of our questions.

Entity Relation (ER) diagram




```
                List of relations
 Schema |          Name           | Type  |   Owner
--------+-------------------------+-------+-----------
 public | businesses              | table | anoutsala
 public | businesses_in_fl        | view  | anoutsala
 public | businesses_in_tampa     | view  | anoutsala
 public | businesses_in_top3      | view  | anoutsala
 public | not_in_users            | table | anoutsala
 public | reviews                 | table | anoutsala
 public | tampa_business_reviews  | view  | anoutsala
 public | tampa_reviews           | view  | anoutsala
 public | unfiltered_reviews      | table | anoutsala
 public | users                   | table | anoutsala
```

Additional Adjustments to MongoDB

In MongoDB, we made collections for each of the three JSON datasets we imported: business, reviews, and users. This process was relatively easy as we were able to use the code provided in Project 4 as a reference. We created numerous indexes for each collection to ensure quicker runtime. We created indexes for `business_id`, `city`, and `state` in the business relation. Then we created 2 indexes for `business_id` and `review_user_id` in the review relation. Here are the indexes created:

```
business.create_index([("business_id", 1)], name="business_id_idx")
business.create_index([("city", 1)], name="city_idx")
business.create_index([("state", 1)], name="state_idx")

review.create_index([("business_id", 1)], name="review_business_id_idx")
review.create_index([("user_id", 1)], name="review_user_id_idx")

user.create_index([("user_id", 1)], name="user_id_idx")
```

System and Database Setup

Database Loading

Originally, we planned to load and run our queries in Datahub. However, we quickly ran into an error because of memory limitations. The Datahub server only allowed 3,072 MB of memory, while our dataset was 8.81GB. So, we chose to load the Yelp dataset into our local machines instead.

PostgreSQL

To set up the PostgreSQL database, we began by pip-installing the Psycopg library, a PostgreSQL adapter for Python. Using VSCode and the Terminal, we established a connection to the PostgreSQL database and created the **businesses**, **users**, and **reviews** tables. Here is an example of the **businesses** table:

```
cur.execute("""
    CREATE TABLE IF NOT EXISTS businesses (
        business_id VARCHAR(22) PRIMARY KEY,
        name TEXT,
        address TEXT,
        city TEXT,
        state TEXT,
        postal_code TEXT,
        latitude FLOAT,
        longitude FLOAT,
        stars FLOAT,
        review_count INT
    );
""")
```

Since the data loaded was in JSON format, we needed to normalize and coerce it into the relational database structure used for PostgreSQL. To accomplish this, we wrote Python scripts to process and prepare the data for insertion. Using the JSON library, we parsed each JSON file and loaded the records into their corresponding databases. Here's the code of how we're parsing the JSON file:

```
file_path = "yelp_dataset/yelp_academic_dataset_business.json"

with open(file_path, "r") as f:
    data = [json.loads(line) for line in f]
```

The *insert_ query* inserts each business into the **businesses** table, using `%s` as placeholders for data values. To ensure referential integrity, we added foreign key constraints on the **review** table for *user_id* and *business_id*, linking it to the **businesses** and **users** tables, respectively. For each business, the *cur_execute()* function substitutes the placeholders with corresponding values from the dataset, inserting "None" for missing fields. Here is the code for how we loaded in the data into businesses:

```
insert_query = """
    INSERT INTO businesses (
        business_id, name, address, city, state, postal_code, latitude, longitude,
stars, review_count
    ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
    ON CONFLICT (business_id) DO NOTHING;
"""

for business in data:
    cur.execute(
        insert_query,
        (
            business["business_id"],
            business.get("name"),
            business.get("address"),
            business.get("city"),
            business.get("state"),
            business.get("postal_code"),
            business.get("latitude"),
            business.get("longitude"),
            business.get("stars"),
            business.get("review_count")
        )
    )
```

While the **businesses** table (118 MB) took minutes to load, the **reviews** table (5.34 GB) was significantly larger, often taking hours to load.

MongoDB

To use MongoDB locally, we began by installing MongoDB Community Edition and integrating it with VScode to establish a connection to localhost. We used the PyMongo library to work with MongoDB from Python. Using Python's JSON library, we loaded all the data from each JSON file, respectively. Here is the code of how we loaded the data into the **business** collection:

```
if business.count_documents({}) == 0:
    print("Loading business collection...")
    with open('yelp_dataset/yelp_academic_dataset_business.json', encoding='utf-8') as
f:
        for line in f:
            business.insert_one(json.loads(line))
```

We created 3 collections: **business**, **review**, and **user**. However, with the large file size, the loading time was almost 30 minutes.

Data Inconsistencies

The PostgreSQL **ON CONFLICT** clause ensures that duplicate entries on **business_id** are skipped without causing errors. Using **ON CONFLICT** is helpful for preventing duplicate IDs across all relations. After applying this clause, we compared the number of rows in our **businesses** and **users** tables to those in MongoDB, and found that they matched. However, the MongoDB **reviews** collection had 32 more rows than the PostgreSQL **reviews** table. Upon further investigation, we discovered that these 32 additional rows were caused by invalid **user_id** values in the **reviews** table, which did not exist in the **users** table. This discrepancy led to a **Foreign Key Constraint** violation when referencing the **users** table. To resolve the issue, we identified and removed these invalid **user_table** values from both the PostgreSQL database and MongoDB.

Task Selection

Task 1: Florida's Review Hotspots

By analyzing the number of businesses in different states, we found that Florida has the highest concentration. As a result, we decided to focus on Florida and created a **businesses_in_fl** view. From there, we identified the top three cities with the most businesses: Tampa, Clearwater, and Saint Petersburg. Below is the code we used to create the **businesses_in_fl** view:

```
CREATE VIEW businesses_in_fl AS
SELECT *
FROM businesses
WHERE state = 'FL';
```

Data Preparation

We observed significant variation in the spellings of these city names, so we attempted using Levenshtein distance with a threshold of 5. Our initial assumption was that the distance of 5 would be sufficient to link variations of “Tampa Bay” to “Tampa” and “St. Petersburg” to “Saint Petersburg”. However, this resulted in capturing too many unrelated city names shown below for the “Tampa” capture group:

city	count
Tampa	9048
Largo	1002
Tampa Bay	141
TAMPA	16
Ozona	4
Balm	2
Pasco	2
tampa	2
Apopka	1
Mango	1
New Tampa	1
Tampa	1
Tampa,	1
Tampa,Fl	1
Tampla	1
largo	1

Even though the Levenshtein algorithm didn’t provide the desired results, we still performed manual entity resolution, especially for St. Petersburg, which had as many as 1,185 reviews. To address this, we created a view that focused on the top three cities in Florida, including selected variations of St. Petersburg. This allowed us to better organize and handle the data for St. Petersburg while ensuring accurate analysis across other cities as well. Below is our code:

```
CREATE VIEW businesses_in_top3 AS
    SELECT city, stars
    FROM businesses_in_fl
    WHERE city IN ('Tampa', 'Clearwater', 'St Petersburg', 'St. Petersburg', 'Saint
Petersburg');
```

PostgreSQL Query

```
SELECT
    city_grouped AS city,
    AVG(stars) AS avg_review_score
    FROM (
        SELECT
            CASE
                WHEN city IN ('St Petersburg', 'St. Petersburg', 'Saint Petersburg')
THEN 'Saint Petersburg'
                ELSE city
            END AS city_grouped,
            stars
        FROM businesses_in_top3
    ) AS grouped_data
    GROUP BY city_grouped
    ORDER BY avg_review_score DESC;
```

We wanted to find the average review stars of each of the top cities in Florida to gauge a baseline of what an average business in the review hotspots was rated. We broke the query down into two parts: the subquery and the aggregation function to get the average stars. The main purpose of the subquery was to group the three variations of spellings of St. Petersburg together. Then aggregating by the city name to calculate the average review score. Below is the output we got:

city	avg_review_score
Saint Petersburg	3.74297
Clearwater	3.60131
Tampa	3.58306

PostgreSQL Performance

```
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=5540.07..5541.94 rows=748 width=40) (actual time=30.208..31.845 rows=3 loops=1)
   Sort Key: (avg(businesses.stars)) DESC
   Sort Method: quicksort  Memory: 25kB
   ->  Finalize GroupAggregate  (cost=5402.45..5504.36 rows=748 width=40) (actual time=30.158..31.798 rows=3 loops=1)
         Group Key: (CASE WHEN (businesses.city = ANY ('{"St Petersburg","St. Petersburg","Saint Petersburg"}'::text[])) THEN 'Saint Petersburg'::text ELSE businesses.city END)
         ->  Gather Merge  (cost=5402.45..5488.47 rows=748 width=64) (actual time=30.149..31.788 rows=6 loops=1)
               Workers Planned: 1
               Workers Launched: 1
               ->  Sort  (cost=4402.44..4404.31 rows=748 width=64) (actual time=22.752..22.753 rows=3 loops=2)
                     Sort Key: (CASE WHEN (businesses.city = ANY ('{"St Petersburg","St. Petersburg","Saint Petersburg"}'::text[])) THEN 'Saint Petersburg'::text ELSE businesses.city END)
                     Sort Method: quicksort  Memory: 25kB
                     Worker 0:  Sort Method: quicksort  Memory: 25kB
                     ->  Partial HashAggregate  (cost=4356.45..4366.73 rows=748 width=64) (actual time=22.699..22.706 rows=3 loops=2)
                           Group Key: CASE WHEN (businesses.city = ANY ('{"St Petersburg","St. Petersburg","Saint Petersburg"}'::text[])) THEN 'Saint Petersburg'::text ELSE businesses.city END
                           Batches: 1  Memory Usage: 49kB
                           Worker 0:  Batches: 1  Memory Usage: 49kB
                           ->  Parallel Seq Scan on businesses  (cost=0.00..4348.89 rows=1511 width=40) (actual time=0.045..21.087 rows=7252 loops=2)
                                 Filter: ((state = 'FL'::text) AND (city = ANY ('{Tampa,Clearwater,"St Petersburg","St. Petersburg","Saint Petersburg"}'::text[])))
                                 Rows Removed by Filter: 67921
 Planning Time: 0.577 ms
 Execution Time: 32.012 ms
(21 rows)
```

The query cost 5,541.94 and took 31.845ms. The query utilized sort merge to group all the St. Petersburg spelling variations together and then aggregated the values to get the average review score for each city. One of PostgreSQL's strengths lies in its efficient aggregation capabilities, enabling us to quickly calculate review counts and average review scores.

Task 2: Tampa's Biggest Hater

Tampa stood out with a significantly higher number of reviews compared to other cities, prompting us to drill down further into its data for additional insights. We thought it would be intriguing to identify **Tampa's Biggest Hater** by finding the users in Tampa with the lowest average star ratings.

Data Preparation

We found quite a few spelling variations for Tampa in PostgreSQL, so we agreed on including 4 different spelling variations as those had a significant amount of businesses. Below is a code snippet of the variations we included for the **businesses_in_tampa** view:

```
CREATE  VIEW businesses_in_tampa AS
    SELECT *
    FROM businesses_in_fl
    WHERE city = 'Tampa'
    OR city = 'Tampa Bay'
    OR city = 'tampa'
    OR city = 'TAMPA';
```

To identify **Tampa's Biggest Hater** using MongoDB, we also had to define the 4 name variations for Tampa and made an initial pipeline to group the businesses in Tampa together. Below is the code for how we constructed tampa_businesses:

```
city_variations = ["Tampa", "Tampa Bay", "tampa", "TAMPA"]

escaped_cities = [re.escape(city) for city in city_variations]
regex_pattern = f"^({'|'.join(escaped_cities)})$"

pipeline_tampa_businesses = [
    {"$match": {
        "city": {"$regex": regex_pattern, "$options": "i"},
        "state": "FL"
    }}
]

tampa_businesses = list(business.aggregate(pipeline_tampa_businesses))
```

PostgreSQL Query

```
CREATE VIEW tampa_reviews AS
    SELECT r.*
    FROM reviews r
    JOIN (
        SELECT DISTINCT business_id
        FROM businesses_in_tampa) AS b
    ON b.business_id = r.business_id;

SELECT
        r.user_id,
        u.name AS user_name,
        COUNT(*) AS review_count,
        AVG(r.stars) AS avg_stars
    FROM tampa_reviews r
    JOIN users u
    ON r.user_id = u.user_id
    GROUP BY r.user_id, u.name
    HAVING COUNT(*) > 5
    ORDER BY avg_stars ASC, review_count DESC, user_name ASC
    LIMIT 10;
```

Finding **Tampa's Biggest Hater** led down different avenues of thought in how we defined "Biggest Hater". We quickly discovered many users only had one review, and often it was to give a one-star star to a certain business –such enraged users! However, just leaving one bad review wouldn't mean they were the "Biggest Hater". We calculated the average number of reviews per user, which turned out to

be around 3. To focus on more consistent reviewers, we used a `HAVING` clause to filter out users with fewer than 5 reviews. We decided this threshold ensured we captured only the most active critics who consistently left negative reviews. We ordered the results by ascending average star ratings and descending review counts. Here is the output of the top 10 biggest haters in Tampa:

user_id	user_name	review_count	avg_stars
C7fbmhCmXXhtWRZyW6Nzmg	A	14	1
k2lXKvz4iSw0OZSSojCkdw	Bob-A-Lou	10	1
YgeZ-tJ0ZEU_m41pGBng_A	Carlos	10	1
CGp3KducoS3lemPVanz0Xw	Lola	9	1
mvUw8RPzmXkVdCR4Pr_WJg	Mauro	9	1
OL87RR63x07Y0K2xpCG4Aw	Nicole	9	1
fluPpH9ap58Y5xkPPyFQuQ	Rick	9	1
EXL55TgQAo1RRUIpZvKmWQ	Sandy	9	1
mAI_E0rVjyC4FF9PSn37rA	Terry	9	1
zHKbREByeKvgbKPxbKIweA	Vic	9	1

This approach led us to identify **Tampa's Biggest Hater**—a user who left 14 reviews, all averaging to one star. To learn more about the infamous user, we created a view of these results and joined it with the *users* table using `user_id`. This revealed that Tampa's Biggest Hater on Yelp goes by the username A. Tied for Tampa's Biggest Hater runner-up are Bob-A-Lou and Carlos, both with 10 reviews, again, all of their reviews averaging one star.

MongoDB Query

```
tampa_business_ids = [business["business_id"] for business in tampa_businesses]

pipeline_hater_query = [

    {"$match": {"business_id": {"$in": tampa_business_ids}}},
    {
        "$lookup": {
            "from": "user",
            "localField": "user_id",
            "foreignField": "user_id",
            "as": "user_details"
        }
    },
    {"$unwind": "$user_details"},
    {
        "$group": {
            "_id": {"user_id": "$user_id", "user_name": "$user_details.name"},
            "review_count": {"$sum": 1},
            "avg_stars": {"$avg": "$stars"}
        }
    },
    {"$match": {"review_count": {"$gt": 5}}},
    {"$sort": { "user_name" : 1,"avg_stars": 1,  "review_count": -1}},
    {"$limit": 10},
    {
        "$project": {
            "_id": 0,
            "user_id": "$_id.user_id",
            "user_name": "$_id.user_name",
            "review_count": 1,
            "avg_stars": {"$round": ["$avg_stars", 2]}
        }
    }
]
```

To find “Tampa’s Biggest Hater” using MongoDB, filtering first significantly helped reduce the amount of joins performed. One of MongoDB’s biggest weaknesses is in its ability to join collections together. We found that extracting out all the documents we wanted and performing the *$match* function with the IDs helped reduce the number of joins we needed to perform. We began by finding all the businesses in Tampa and extracting the `business_id`s to filter for all the reviews of the businesses in the area. Then

we performed the *$lookup* function, MongoDB's left outer join, with the user collection to get user names. Next, we performed aggregation similar to PostgreSQL to find the `review_count` and `avg_stars` per user and filtered out all the users with less than 5 reviews to ensure we only got the active users. We sorted our results by `avg_stars` ascending and `review_count` descending. Here are the results of the top 10 biggest haters in Tampa:

```
Top 10 Haters in Tampa, FL:
   review_count                 user_id  user_name  avg_stars
0            14  C7fbmhCmXXhtWRZyW6Nzmg          A        1.0
1            10  k2lXKvz4iSw0OZSSojCkdw  Bob-A-Lou        1.0
2            10  YgeZ-tJ0ZEU_m41pGBng_A     Carlos        1.0
3             9  fluPpH9ap58Y5xkPPyFQuQ       Rick        1.0
4             9  CGp3KducoS3lemPVanz0Xw       Lola        1.0
5             9  OL87RR63x07Y0K2xpCG4Aw     Nicole        1.0
6             9  EXL55TgQAo1RRUIpZvKmWQ      Sandy        1.0
7             9  zHKbREByeKvgbKPxbKIweA        Vic        1.0
8             9  mAI_E0rVjyC4FF9PSn37rA      Terry        1.0
9             9  mvUw8RPzmXkVdCR4Pr_WJg      Mauro        1.0
```

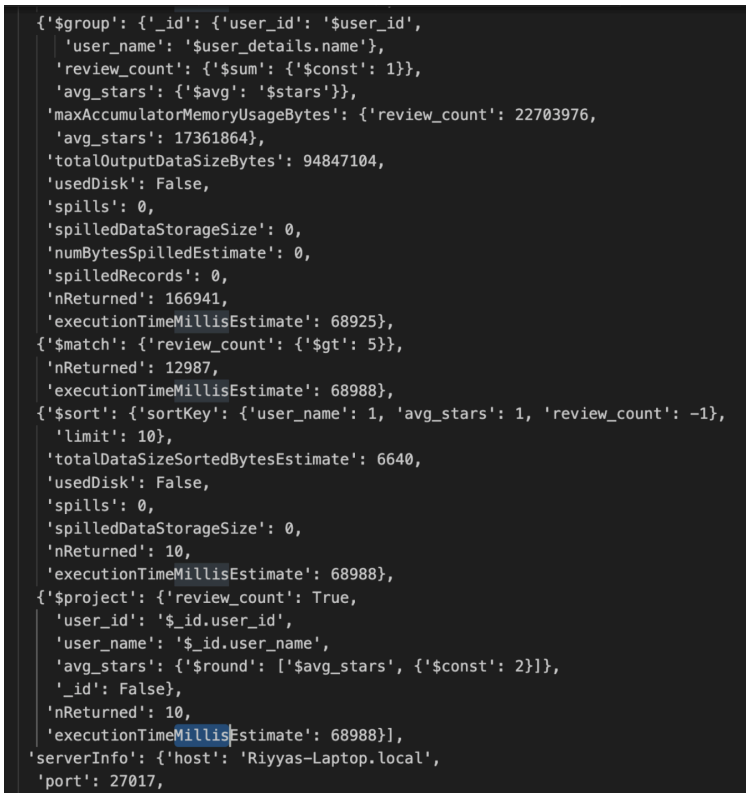
Comparison of Tools for Tampa's Biggest Hater

After outputting the relations of Tampa's biggest hater in PostgreSQL and MongoDB, we noticed the orders of the relations to not be the same for the users with the same avg_stars. While we are unsure of the exact reason, we suspect it may be due to the fact that we created an index for the MongoDB queries, but didn't apply one for the PostgreSQL queries.

PostgreSQL Performance

```
                                                                                                QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=732540.45..732540.48 rows=10 width=69) (actual time=11894.914..11917.767 rows=10 loops=1)
   ->  Sort  (cost=732540.45..732769.86 rows=91762 width=69) (actual time=11894.912..11917.765 rows=10 loops=1)
         Sort Key: (avg(r.stars)), (count(*)) DESC, u.name
         Sort Method: top-N heapsort  Memory: 26kB
         ->  Finalize GroupAggregate  (cost=695074.18..730557.51 rows=91762 width=69) (actual time=11810.505..11915.958 rows=12987 loops=1)
               Group Key: r.user_id, u.name
               Filter: (count(*) > 5)
               Rows Removed by Filter: 153954
               ->  Gather Merge  (cost=695074.18..724134.14 rows=229406 width=69) (actual time=11810.428..11873.747 rows=168159 loops=1)
                     Workers Planned: 2
                     Workers Launched: 2
                     ->  Partial GroupAggregate  (cost=694074.16..696654.97 rows=114703 width=69) (actual time=11795.326..11822.557 rows=56053 loops=3)
                           Group Key: r.user_id, u.name
                           ->  Sort  (cost=694074.16..694360.91 rows=114703 width=33) (actual time=11795.171..11804.916 rows=152819 loops=3)
                                 Sort Key: r.user_id, u.name
                                 Sort Method: external merge  Disk: 6728kB
                                 Worker 0:  Sort Method: external merge  Disk: 6576kB
                                 Worker 1:  Sort Method: external merge  Disk: 6920kB
                                 ->  Parallel Hash Join  (cost=77569.88..681295.28 rows=114703 width=33) (actual time=11585.091..11755.657 rows=152819 loops=3)
                                       Hash Cond: ((r.user_id)::text = (u.user_id)::text)
                                       ->  Hash Join  (cost=18987.36..615178.66 rows=114703 width=27) (actual time=61.778..10784.681 rows=152819 loops=3)
                                             Hash Cond: ((r.business_id)::text = businesses.business_id)
                                             ->  Parallel Seq Scan on reviews2 r  (cost=0.00..588547.44 rows=2911644 width=50) (actual time=2.845..10537.171 rows=2330082 loops=3)
                                             ->  Hash  (cost=18967.06..18967.06 rows=1624 width=23) (actual time=58.816..58.817 rows=9207 loops=3)
                                                   Buckets: 16384 (originally 2048)  Batches: 1 (originally 1)  Memory Usage: 623kB
                                                   ->  Unique  (cost=0.42..18950.82 rows=1624 width=23) (actual time=0.037..58.087 rows=9207 loops=3)
                                                         ->  Index Scan using businesses_pkey on businesses  (cost=0.42..18946.76 rows=1624 width=23) (actual time=0.036..57.384 rows=9207 loops=3)
                                                               Filter: ((state = 'FL'::text) AND ((city = 'Tampa'::text) OR (city = 'Tampa Bay'::text) OR (city = 'tampa'::text) OR (city = 'TAMPA'::text)))
                                                               Rows Removed by Filter: 141139
                                       ->  Parallel Hash  (cost=42565.90..42565.90 rows=828290 width=29) (actual time=730.472..730.472 rows=662632 loops=3)
                                             Buckets: 65536  Batches: 64  Memory Usage: 2528kB
                                             ->  Parallel Seq Scan on users u  (cost=0.00..42565.90 rows=828290 width=29) (actual time=0.877..549.099 rows=662632 loops=3)
 Planning Time: 5.614 ms
 Execution Time: 11920.620 ms
(34 rows)
```

MongoDB Performance



The PostgreSQL query took 11917.767ms, while the MongoDB Query took 68998ms. We assume PostgreSQL was faster because it optimized the query and utilized hash join.

Task 3: Solo Review Wonders

After noticing a significant number of users with only one review and an average rating of one star, we became curious about how many businesses in Tampa have many single reviews. One theory we considered was that these reviews might have been incentivized, such as offering a free drink in exchange for a Yelp review.

Data Preparation

To investigate this, we created 2 Common Table Expressions (CTEs) in PostgreSQL. The first CTE, **single_reviews**, included all the users with only one review. The second CTE, **business_single_review_counts**, counted all the single reviews per business. Here is the code for defining both CTEs:

```
WITH single_reviews AS (
        SELECT r.business_id, r.user_id, COUNT(*) AS review_count, AVG(r.stars) AS
avg_stars
        FROM tampa_reviews r
        GROUP BY r.business_id, r.user_id
        HAVING COUNT(*) = 1
    ),
    business_single_review_counts AS (
        SELECT business_id,
               COUNT(*) AS single_review_user_count,
               ROUND(AVG(avg_stars), 2) AS avg_stars_for_single_reviews
        FROM single_reviews
        GROUP BY business_id
    )
```

PostgreSQL Query

```
SELECT b.business_id,
        tb.name AS business_name,
        single_review_user_count,
        avg_stars_for_single_reviews
FROM business_single_review_counts b
JOIN businesses_in_tampa tb
ON b.business_id = tb.business_id
ORDER BY single_review_user_count DESC
LIMIT 3;
```

To test our assumptions about incentivized-driven reviews, we retrieved the top 3 businesses with the highest amounts of single-review users and the average stars given of all the single-review users per business, denoted as `avg_stars_for_single_reviews`. The `avg_stars_for_single_reviews` per business was quite high (all above 4 stars). Here is the full output we got:

business_id	business_name	single_review_user_count	avg_stars_for_single_reviews
QHWYlmVbLC3K6eglWoHVvA	Datz	3148	4.14
L5LLN0RafiV1Z9cddzvuCw	Ulele	2962	4.17
dsfRniRgfbDjC8os848B6A	Bern's Steak House	2831	4.26

We started thinking that if we got a free drink at a restaurant, we would also rate it high on Yelp, so we wanted to find some evidence on Google confirming our assumption. We didn't end up finding anything to confirm our assumption about incentives on Google. Instead, we found that the restaurants were highly loved by Tampans. Ulele and Bern's Steak House were featured on the Michelin guide, while Datz was an iconic staple to Tampa.

Even highly rated businesses get haters:

MongoDB Query

```
pipeline = [
    {
        "$lookup": {
            "from": "business",
            "localField": "business_id",
            "foreignField": "business_id",
            "as": "business_details"
        }
    },
    {
        "$unwind": "$business_details"
    },
    {
        "$match": {
            "business_details.city": {
                "$regex": regex_pattern,
                "$options": "i"
            },
            "business_details.state": {
                "$regex": "^FL$",
                "$options": "i"
            }
        }
    },
    {
        "$group": {
            "_id": {
                "business_id": "$business_id",
                "user_id": "$user_id"
            },
            "review_count": { "$sum": 1 },
            "avg_stars": { "$avg": "$stars" }
        }
    },
    {
        "$match": {
            "review_count": 1
        }
    },
    {
        "$group": {
```

```
            "_id": "$_id.business_id",
            "single_review_user_count": { "$sum": 1 },
            "avg_stars_for_single_reviews": { "$avg": "$avg_stars" }
        }
    },
    {
        "$lookup": {
            "from": "business",
            "localField": "_id",
            "foreignField": "business_id",
            "as": "business_details"
        }
    },
    {
        "$unwind": "$business_details"
    },

    {
        "$project": {
            "_id": 0,
            "business_id": "$_id",
            "business_name": "$business_details.name",
            "single_review_user_count": 1,
            "avg_stars_for_single_reviews": { "$round":
["$avg_stars_for_single_reviews", 2] }
        }
    },
    {
        "$sort": {
            "single_review_user_count": -1
        }
    },
    {"$limit": 3}

]
```

MongoDB's limitations on joins result in redundant operations. MongoDB's `$lookup` joined the **review** collection with the **business** collection on `business_id` and embeds the details of the joined collection into an array. Next, we performed the `$match` function to filter for all businesses in Tampa. Reviews are then aggregated by business and user, followed by filtering to retain only single-review users. A second aggregation groups data by business. To retrieve business names, we perform another join with the business collection. The output of the pipeline is shown below:

```
Businesses in Tampa with High Number of Single-Review Users:
   single_review_user_count             business_id       business_name  \
0                      3148  QHWYlmVbLC3K6eglWoHVvA                Datz
1                      2962  L5LLN0RafiV1Z9cddzvuCw               Ulele
2                      2831  dsfRniRgfbDjC8os848B6A  Bern's Steak House

   avg_stars_for_single_reviews
0                          4.14
1                          4.17
2                          4.26
```

Comparison of Tools for Solo Review Wonders

PostgreSQL Performance

```
                                                                QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------------------------
-
 Limit  (cost=666847.49..666847.49 rows=2 width=83) (actual time=11189.795..11190.808 rows=25 loops=1)
   ->  Sort  (cost=666847.49..666847.49 rows=2 width=83) (actual time=11189.794..11190.806 rows=25 loops=1)
         Sort Key: (count(*)) DESC
         Sort Method: top-N heapsort  Memory: 30kB
         ->  Nested Loop  (cost=629741.98..666847.48 rows=2 width=83) (actual time=10948.425..11189.993 rows=9207 loops=1)
               ->  GroupAggregate  (cost=629741.56..665251.47 rows=200 width=63) (actual time=10948.374..11178.785 rows=9207 loops=1)
                     Group Key: r.business_id
                     ->  Finalize GroupAggregate  (cost=629741.56..665224.89 rows=1376 width=86) (actual time=10948.353..11157.765 rows=429298 loops=1)
                           Group Key: r.business_id, r.user_id
                           Filter: (count(*) = 1)
                           Rows Removed by Filter: 13667
                           ->  Gather Merge  (cost=629741.56..658801.52 rows=229406 width=86) (actual time=10948.316..11041.352 rows=452682 loops=1)
                                 Workers Planned: 2
                                 Workers Launched: 2
                                 ->  Partial GroupAggregate  (cost=628741.54..631322.36 rows=114703 width=86) (actual time=10939.870..10984.880 rows=150894 loops=3)
                                       Group Key: r.business_id, r.user_id
                                       ->  Sort  (cost=628741.54..629028.30 rows=114703 width=50) (actual time=10939.734..10957.507 rows=152819 loops=3)
                                             Sort Key: r.business_id, r.user_id
                                             Sort Method: external merge  Disk: 9288kB
                                             Worker 0:  Sort Method: external merge  Disk: 9184kB
                                             Worker 1:  Sort Method: external merge  Disk: 9440kB
                                             ->  Hash Join  (cost=18987.36..615178.66 rows=114703 width=50) (actual time=72.797..10859.770 rows=152819 loops=3)
                                                   Hash Cond: ((r.business_id)::text = businesses_1.business_id)
                                                   ->  Parallel Seq Scan on reviews2 r  (cost=0.00..588547.44 rows=2911644 width=50) (actual time=0.491..10598.467 rows=2330082 loops=3)
                                                   ->  Hash  (cost=18967.06..18967.06 rows=1624 width=23) (actual time=71.967..71.967 rows=9207 loops=3)
                                                         Buckets: 16384 (originally 2048)  Batches: 1 (originally 1)  Memory Usage: 623kB
                                                         ->  Unique  (cost=0.42..18950.82 rows=1624 width=23) (actual time=0.075..70.982 rows=9207 loops=3)
                                                               ->  Index Scan using businesses_pkey on businesses businesses_1  (cost=0.42..18946.76 rows=1624 width=23) (actual time=0.074..70.006 rows=9207 loops=3)
                                                                     Filter: ((state = 'FL'::text) AND ((city = 'Tampa'::text) OR (city = 'Tampa Bay'::text) OR (city = 'tampa'::text) OR (city = 'TAMPA'::text)))
                                                                     Rows Removed by Filter: 141139
               ->  Index Scan using businesses_pkey on businesses  (cost=0.42..7.97 rows=1 width=43) (actual time=0.001..0.001 rows=1 loops=9207)
                     Index Cond: (business_id = (r.business_id)::text)
                     Filter: ((state = 'FL'::text) AND ((city = 'Tampa'::text) OR (city = 'Tampa Bay'::text) OR (city = 'tampa'::text) OR (city = 'TAMPA'::text)))
 Planning Time: 6.050 ms
 Execution Time: 11191.774 ms
(35 rows)
```

MongoDB Performance

```
{'explainVersion': '1',
 'stages': [{'$cursor': {'queryPlanner': {'namespace': 'yelp.review',
     'parsedQuery': {},
     'indexFilterSet': False,
     'queryHash': '4E7FE95E',
     'planCacheKey': 'D8CBDF1E',
     'optimizationTimeMillis': 0,
     'maxIndexedOrSolutionsReached': False,
     'maxIndexedAndSolutionsReached': False,
     'maxScansToExplodeReached': False,
     'prunedSimilarIndexes': False,
     'winningPlan': {'isCached': False,
      'stage': 'PROJECTION_SIMPLE',
      'transformBy': {'business_id': 1, 'stars': 1, 'user_id': 1, '_id': 0},
      'inputStage': {'stage': 'COLLSCAN', 'direction': 'forward'}},
     'rejectedPlans': []},
    'executionStats': {'executionSuccess': True,
     'nReturned': 6990277,
     'executionTimeMillis': 365113,
     'totalKeysExamined': 0,
     'totalDocsExamined': 6990277,
     'executionStages': {'isCached': False,
      'stage': 'PROJECTION_SIMPLE',
      'nReturned': 6990277,
      'executionTimeMillisEstimate': 4803,
...
   {'$sort': {'single_review_user_count': -1}},
   {'$limit': 3}],
  'cursor': {},
  '$db': 'yelp'},
 'ok': 1.0}
```

The PostgreSQL query took 11190.808ms. The MongoDB query took 365113ms which was a lot longer than PostgreSQL. We assume MongoDB had a much longer run time because PostgreSQL optimizes the process and utilizes hash join and nested loop.

Further Tasks: Hotspots of Tampa

We realized that we wanted to do more of an analysis on other aspects of this data set that did not include reviews, so we went back and decided to do some sort of analysis on the business tables, again only focusing on those in Tampa, FL. We were curious to see what the distribution of businesses looked like per zip code, so we created a simple query that allowed us to investigate just that.

Wanting to understand the distribution of businesses, we quickly realized that we might uncover some “Downtowns” or “City Centers” which are often the most business-populated places in cities. We wanted to try to find the business hotspots of Tampa to try to find the “Downtowns” or “City Centers” just using the business dataset. This prompted us to aggregate the data by zip codes and see how businesses are distributed among a variety of zip codes.

PostgreSQL Query:

```
SELECT
       postal_code,
       COUNT(*) AS total_businesses
   FROM businesses_in_tampa
   GROUP BY postal_code ORDER BY COUNT(*) DESC LIMIT 10;
```

To investigate this, used our **businesses_in_tampa** view and ran an aggregation query to count all the businesses in each specific zip code.

postal_code	total_businesses
33607	794
33602	649
33609	639
33618	591
33612	580
33606	509
33629	496
33614	434
33611	413
33647	410

After running the query, we found that the zip code with the most businesses is 33607, which is the zip code that includes Downtown Tampa. There are a total of 794 businesses in that zip code.

This led us back to our investigation of how businesses were distributed across the different zip codes. We noticed a skewed distribution with a large standard deviation and a huge difference between the maximum (794) and the average (94.91). This may possibly indicate that businesses are concentrated in a few postal codes rather than being evenly distributed.

min_businesses	max_businesses	avg_businesses	stddev_businesses
1	794	94.9175257731958763	184.141061762845

MongoDB Query:

```
pipeline = [
    {
        "$match": {
            "city": { "$regex": regex_pattern, "$options": "i" },
            "state": { "$regex": "^FL$", "$options": "i" },
            "postal_code": { "$ne": None }
        }
    },
    {
        "$group": {
            "_id": "$postal_code",
            "total_businesses": { "$sum": 1 }
        }
    },
    {
        "$group": {
            "_id": None,
            "min_businesses": { "$min": "$total_businesses" },
            "max_businesses": { "$max": "$total_businesses" },
            "avg_businesses": { "$avg": "$total_businesses" },
            "stddev_businesses": { "$stdDevSamp": "$total_businesses" }
        }
    },
    {
        "$project": {
            "_id": 0,
            "min_businesses": 1,
            "max_businesses": 1,
            "avg_businesses": 1,
            "stddev_businesses": 1
        }
    }
]
```

To investigate the business hotspots of Tampa, we began filtering the data to just get all the `postal_code` of Tampa. Then we got the total number of businesses grouped together by the postal code. We sorted the results in descending order and limited our output to 10. Lastly, we projected the results into the table shown below:

```
Top 10 Postal Codes in Tampa, FL with Most Businesses:
   total_businesses postal_code
0               794       33607
1               649       33602
2               639       33609
3               591       33618
4               580       33612
5               509       33606
6               496       33629
7               434       33614
8               413       33611
9               410       33647
```

Additionally, we ran the statistics to analyze the distribution of business hotspots. Our pipeline is shown below:

```
pipeline = [
{
"$match": {
"city": { "$regex": regex_pattern, "$options": "i" },
"state": { "$regex": "^FL$", "$options": "i" },
"postal_code": { "$ne": None }
}
},
{
"$group": {
"_id": "$postal_code",
"total_businesses": { "$sum": 1 }
}
},
{
"$group": {
"_id": None,
"min_businesses": { "$min": "$total_businesses" },
"max_businesses": { "$max": "$total_businesses" },
"avg_businesses": { "$avg": "$total_businesses" },
"stddev_businesses": { "$stdDevSamp": "$total_businesses" }
}
},
{
"$project": {
"_id": 0,
"min_businesses": 1,
"max_businesses": 1,
"avg_businesses": 1,
"stddev_businesses": 1
}
}
]
```

Comparison of Tools for Hotspots of Tampa:

We started off wanting to change the nature of our queries by diving into the distribution of businesses throughout Tampa. Our goal with this process was to compare the results from both PostgreSQL and MongoDB outputs to ensure the dynamics of both databases were the same. Our problem-solving process included writing these analysis queries with PostgreSQL and then translating them to MongoDB. This is where our comparison began as we wanted to ensure the outputs on both systems were the same — which they were!

Additional Tools Used

We used the Markdown Preview Enhanced extension on VScode to generate a polished pdf report.

Team Reflections

Warnings

A common consensus among our group is that big files take a long time (numerous hours) to load the data on PostgreSQL and MongoDB. When loading the data in, there are various methods (we document our individual experiences below) with both PostgreSQL and MongoDB since each individual machine has its own errors and limitations. We each received different errors along the process when completing the same steps.

We had significantly more practice writing in PostgreSQL, we were more familiar with the commands to quickly find what we needed using PostgreSQL. Hence, most of our queries are first written in PostgreSQL and later adapted to MongoDB. MongoDB uses dot notation, so we had to get familiar with using it and the process it takes. While it has a lot of potential, it changed the order in which we wrote certain queries. However, if you have more experience with MongoDB, you may have a different experience with the tools.

Recommendations

To load data faster, a recommendation is using the *cur.executemany()* in the Psycopg library to batch process the data insertion process. However, don't expect your code to load instantly, it does still take a while, especially with large datasets such as the Yelp datasets. Another observation we found is that running PostgreSQL code on the Terminal is quicker than on our local Jupyter notebooks.

We also recommend working with a group. Working together brought different ideas to the table while showcasing a variety of ways to load our data due to different errors. We can all agree that working together was definitely better than working individually on this project.

Individual Reflections

Riyya Ahmed: This project rewarded me with data analysis, collaboration, and problem-solving lessons. The initial setup and data loading were found to be one of the biggest challenges, as an important amount of time was required to complete these tasks. I found Datahub helpful at first, but for better performance, I switched to VS Code which made it easier to load MongoDB and PostgreSQL

locally. I concentrated mainly on MongoDB queries, translating complicated PostgreSQL queries into efficient pipelines. Spending more time on data analysis, and planning strategies for queries in PostgreSQL made the switch to MongoDB easier. Brainstorming queries with my teammates and documenting our work improved my problem-solving skills, and it also improved my teamwork abilities. I found that using a few indexes can improve performance and breaking many large queries into smaller steps helps me analyze enormous datasets effectively. I enjoyed tackling this project, along with working with my team overall. I learned to work well with large datasets and build complicated queries. We accomplished a lot of things in this project and I'm grateful for the shared experiences.

Mariajose Garcia Morones: Working on this project has greatly improved my skills by blending technical learning with real-world data challenges. This project not only pushed me to further learn and familiarize myself with two powerful database systems, PostgreSQL and MongoDB but also to uncover hidden insights in Yelp's dataset. After long hours of research, office hours, and team meetings, I was able to set up my dataset locally and use a code editor, VS Code, to help modify and create my tables. My processes in setting up my Mongo Community locally was similar. Except, I had to perform an update to my laptop in the middle of our team working session to successfully install and set up anything on Mongo. It was a little frustrating but it ultimately didn't take too long. Some of my team members preferred using Python in VS Code to write and execute PostgreSQL queries, integrating their commands directly within scripts. However, I found it more efficient to work directly in the terminal, crafting and executing commands purely in SQL. I was able to run and execute queries with ease. I didn't run into the issue of dealing with extensive runtimes, so I was also able to create materialized views. It was a fun process to think of an area we'd like to tap into more and create queries/pipelines for. Overall, I enjoyed this project and I'm happy I took up the challenge. I now feel comfortable working with large and complicated datasets in something other than Python.

Anoutsala (Sara) Hanmonty: Working on this project has been a really rewarding experience, especially in terms of learning and growth. I decided to use VSCode instead of Datahub because I prefer having my work saved locally, which feels more personal and long-term. One of the biggest things I learned was how to use Psycopg to set up and load data into a PostgreSQL database. It was my first time using it, and I was excited to dive into something new. However, I quickly ran into some challenges that made the process a lot slower than I expected. For example, I spent hours trying to load data from large JSON files, and at one point, I let the code run overnight, only to find it hadn't finished by the morning. That was definitely frustrating. The most challenging part was when I ran into a ForeignKeyViolation error after waiting for the data to load. It turned out that some user_id values in the reviews table didn't match any in the users table, which caused the foreign key constraint to fail. After some troubleshooting, I found and removed the invalid user IDs, which solved the problem. This experience taught me the importance of thoroughly checking the data and being prepared for unexpected issues. It also taught me to plan better when choosing schemas and selecting which

columns will be useful for our tasks, rather than downloading the entire dataset. It was also a great opportunity to collaborate with other students and learn from each other throughout the process.

Ashley Pun: This project gave me the opportunity to dive deeper into data engineering, which has challenged me in a technical way and made me a better teammate. In the beginning, I faced challenges while loading data into PostgreSQL using the command line, so I troubleshoot by switching over to the PostgreSQL app and starting my server there. After going to office hours, I learned that using the app restricted how much I could configure. However, those configurations are more high-level and out of scope for this project. I also ran into an issue where my psql terminal wouldn't allow me to create views, which I resolved by restarting my laptop. While working with MongoDB, I encountered server limitations on how much data I could load. After troubleshooting various technical issues and dealing with laptop-related obstacles, I adopted the position of group journalist, where I documented everything and helped write the bulk of our report. This position really taught me the importance of clear documentation and effective communication. I really enjoyed working with my group (and our long work sessions) and felt that this project bonded us together during such a hectic time during the semester.

References

Reddit user. "The Hate Is Real." Reddit, 10 July 2014, https://www.reddit.com/r/funny/comments/2as4x0/the_hate_is_real/. Accessed 9 Dec. 2024.