Final Report: Yelp Ratings

Michelle Dong, Kelly He, Claudia Wu, Emma Ye

Dataset Selection

Dataset Description

The dataset is split into multiple collections: business, review, user_info, and tip. The business collection provides information about the business such as the name, location, time, and some review information. The review collection shows the business, rating, 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.

Truncation Plans

The main purpose is to optimize the dataset by reducing its size while ensuring that our processing systems could handle data efficiently without crashing. Instead of subsampling, we applied truncation to retain every record entirely intact in the dataset. It was made sure that completeness or representativeness was not compromised and the unnecessary, redundant, or inefficient columns were eliminated in order to minimize size/storage and process overhead. This strategy makes the dataset manageable and effective in analysis regarding features that matter for the case at hand. Emphasis has been put on retaining only the most relevant attributes in the final dataset.

Rows in Review were removed, specifically the ones where user_id and business_id do not exist in the User and Business tables because it failed the foreign key restriction. In addition, we removed attributes such as funny and cool from table Review. Keeping the attribute useful provides substantial information about the quality of the review, which satisfies our query needs. Thus, we do not require two additional attributes that provide quality again. Then, we removed the attribute friends, which lists out all the friends of a user, from the User table. friends is a list of strings that takes significant storage space, and removing it helps improve space and runtime of queries. Finally, we did not include the tables Tip and Checkin. Similar to Review, Tip stores user's suggestions and feedback for a restaurant. However, Tip is a "simpler" version of Review, where the user's suggestion is much shorter and Review provides the user's star rating whereas Tip doesn't include that. Therefore, we decide to choose Review over Tip as it provides more information about the user's attitude towards a restaurant.

We used the same truncation plan for our MongoDB database. However, since we are not able to build foreign key checks in MongoDB as we did in PostgreSQL, we didn't exclude rows from Review that do not have a matched user on User collection.

Original dataset vs truncated dataset

file	Orig Size	Trunc Size	Orig # Row	Trunc # Row	Orig # Features	Trunc # Features
Business	118.9 MB	113 MB	150346	150346	14	14

Review	5.34 GB	4.79 GB	6990280	6990000	9	7
User	3.36 GB	245 MB	1987897	1987897	22	5

PostgreSQL database schema

Business table

Column	Type	Collation	Nullable	Default
business_id	character varying(22)		Not null	
name	text		Not null	
address	text			
city	text			
state	text			
postal_code	text			
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)
- "idx_business_id_state" btree (business_id, state)

Referenced by:

- TABLE ""Review"" CONSTRAINT "Review_business_id_fkey" FOREIGN KEY (business_id) REFERENCES "Business"(business_id) ON DELETE CASCADE

User table

Column	Type	Collation	Nullable	Default
user_id	character varying(22)		not null	
name	character varying			
review_count	integer			
yelping_since	date			
average_stars	Double precision			

Indexes:
- "User_pkey" PRIMARY KEY, btree (user_id)
- "idx_user_user_id" btree (user_id)

Referenced by:
- TABLE ""Review"" CONSTRAINT "Review_user_id_fkey" FOREIGN KEY (user_id) REFERENCES "User" (user_id) ON DELETE CASCADE

Review table

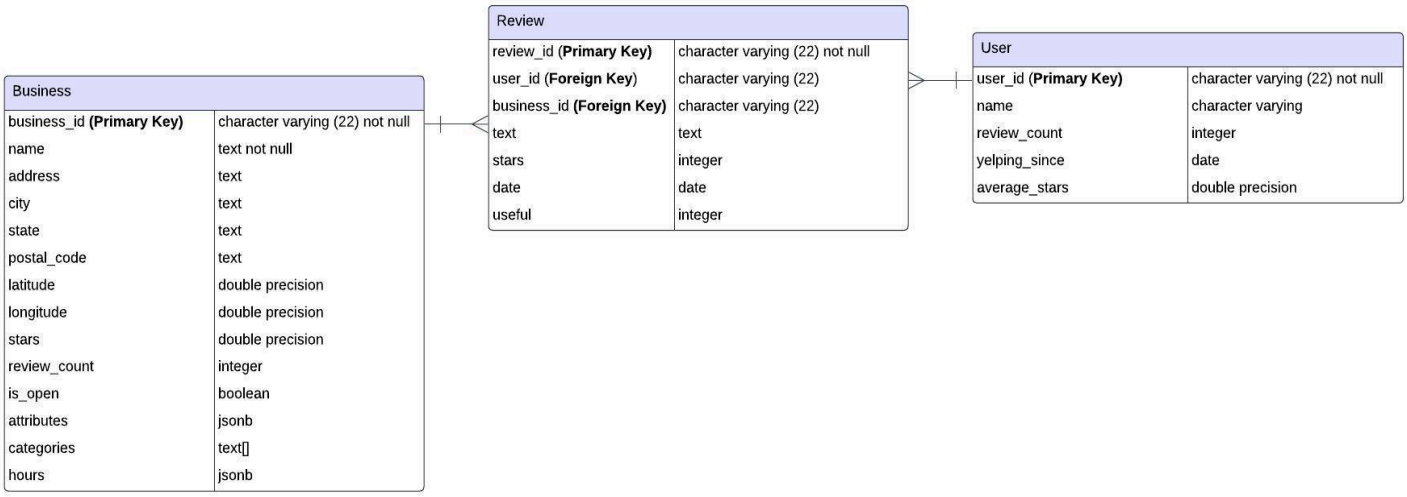
Column	Type	Collation	Nullable	Default
review_id	character varying(22)		not null	
user_id	character varying(22)			
business_id	character varying(22)			
text	text			
stars	integer			
date	date			
useful	integer			

Indexes:
- "Review_pkey" PRIMARY KEY, btree (review_id)
- "idx_review_user_business_id" btree (user_id, business_id)

Foreign-key constraints:

- "Review_business_id_fkey" FOREIGN KEY (business_id) REFERENCES "Business" (business_id) ON DELETE CASCADE
- "Review_user_id_fkey" FOREIGN KEY (user_id) REFERENCES "User" (user_id) ON DELETE CASCADE

ER Diagram



System and Database Setup

PostgreSQL:

System and database setup

The Postgres loading process started 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 was employed to install PostgreSQL. PostgreSQL was installed via conda install -c conda-forge postgresql command. psycopg2 was installed with conda install -c conda-forge psycopg2 to ensure the correct version is used. To facilitate version control and maintain consistency across different setups, the requirements.txt file was included.

To ensure reproducibility, the following versions of the tools and libraries were used:
PostgreSQL: Version 17.2
psycopg2: Version 2.9.9
Python: Version 3.13.0

After the database directory has been set up and the PostgreSQL service started, a non-superuser account and a database named ‘myinner_db’ was created. The connection to the database was managed through the following Python code snippet:

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

Data Transformation

The business data was transformed from a JSON format before inserting it into the PostgreSQL database. We handled missing or malformed data by setting default values, such as empty strings for missing categories or attributes. The categories field, which may be a string, was **split into a list of individual categories**.

```
with open(json_file_path, 'r') as file:
     ...
     data = json.loads(line.strip())
     categories_list = data.get('categories', '').split(', ') if
isinstance(data.get('categories'), str) else []
```

Additionally, attributes and hours fields, which are stored as JSON objects, were transformed into the string format using json.dumps(); The ‘is_open’ field transformed into the boolean data type using bool().

```
data_to_insert.append({...,
 'is_open': bool(data.get('is_open')),
 'attributes': json.dumps(data.get('attributes', {})),
 'categories': categories_list,
 'hours': json.dumps(data.get('hours', {}))})
```

Data Loading

1. **Batch Insertion Logic**

Data loading into PostgreSQL was accomplished through Python scripts utilizing the psycopg2 library. The scripts executed 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 created 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,
        ...); ''')
```

then performed data insertions of **batches with size 1000** from the JSON file and lastly the remaining entries that were under batch size of 1000. This algorithm reduced the number of round trips between the Python application and the database server, which improves the efficiency of handling large datasets.

```
for line_number, line in enumerate(file, 1):
try:
          data_to_insert.append({ ... })
          # Batch Insertion
          if len(data_to_insert) >= 1000:
        cur.executemany(...)
# Insert remain
if data_to_insert:
cur.executemany(...)
```

The ON CONFLICT clause avoids duplicate business_id, review_id, and user_id values during data insertion.

2. **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 ensured 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, ...);''')
```

MongoDB:

System and Database Setup:

The MongoDB setup was similar to the process of PostgreSQL. For compute resources, we used local computers to perform the MongoDB installation with Homebrew, data transformation, and loading processes. We accessed the database using mongosh and established a connection to a local instance.

```
client = MongoClient("mongodb://localhost:27017/")
db = client['yelp_dataset']
```

The data was loaded into MongoDB using a Python script (data_load_mongoDB.py) that utilized the pymongo library. The script reads JSON files containing Yelp business, review, and user data and

filters the relevant fields before inserting the data into appropriate collections within the MongoDB database.

```
 with open(file_path, 'r') as file:
       for line in file:
           document = json.loads(line)
           filtered_document = {field: document.get(field) for
field in fields}
           collection.insert_one(filtered_document)
```

To ensure reproducibility, the following versions of the tools and libraries were used:
pymongo: Version 4.10.1

PostgreSQL Tasks and Queries

Task / Query 1

Problem

The task 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 that provide positive reviews for feedback.

Understanding

This task illustrates the strengths and limitations of MongoDB's flexible schema and aggregation capabilities versus PostgreSQL's relational model. Understanding how to translate complex logic across these systems helps developers decide which is best for specific use cases, such as handling nested queries or relational joins.

PostgreSQL Solution (queries/task_1.sql)

We started by creating a common table expression (user_review_states) that joined the Review and Business tables to associate reviews with business locations and calculate the average rating each user gave per state. Then, we used another common table expression (user_states_count) to aggregate this data, counting the number of distinct states reviewed and computing the overall average rating. Finally, the results were joined with the User table, filtering for users who reviewed businesses in three or more states with an average rating above 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
WHERE state_count >= 3 AND overall_avg_rating > 3.5;
```

PostgreSQL Performance

The query plan shows a nested loop execution with significant time spent in the GroupAggregate and WindowAgg stages. The GroupAggregate operation filters out 2.1 million rows using a cost of 11,141 ms, mainly in the Gather Merge step with parallel execution across 3 workers. The parallel hash join between the Review and Business tables, aided by the use of an index scan, demonstrates efficient

handling of joins. However, memory constraints in the WindowAgg stage due to an external merge sort highlight the need for further optimization.

PostgreSQL Optimization (queries/task_1_optimized.sql)

Next, we tackled the runtime of the query, where we created indexes to optimize the runtime for joining and filtering operations. We pushed down the filters by moving filters on stars and filtering on states to the CTE user_states_count. We added the following indexes: idx_business_id_state, idx_review_user_business_id, and idx_user_user_id.

```
CREATE INDEX idx_business_id_state ON "Business" (business_id,
state);
CREATE INDEX idx_review_user_business_id ON "Review" (user_id,
business_id);
CREATE INDEX idx_user_user_id ON "User" (user_id);
```

These optimizations reduce disk I/O, improve row lookup efficiency, and overall contribute to better query performance. 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.

PostgreSQL Performance Comparison

Step	Initial Query Plan	Updated Query Plan
1	**Index Scan** User Table Scan (Cost: 0.43..8.45, Time: 0.076..0.076) Index Cond: user_id::text = r.user_id::text	**Index Scan** User Table Scan (Cost: 0.43..8.45, Time: 0.046..0.046) Index Cond: user_id::text = r.user_id::text
2	**Parallel Hash Join** Cost: 15175.49..604551.02 Rows: 2912500 Hash Cond: r.business_id::text = b.business_id::text	**GroupAggregate** Cost: 5825.64..604666.64 Rows: 1561454 Hash Cond: r.business_id::text = b.business_id::text
3	**Sort** Cost: 1056633.62..1063914.87 Method: External merge, Disk: 96208kB, 94624kB, 96712kB	**Sort** Cost: 724439.97..726867.05 Method: External merge, Disk: 63752kB, 63376kB, 65600kB
4	**Partial GroupAggregate** Cost: 1053401.72..1111653.78 Rows: 2912603	**Partial GroupAggregate** Cost: 721193.34..740610.70 Rows: 970868
5	**Gather Merge** Cost: 1054401.74..1785027.11 rows: 5825206	**Gather Merge** Cost: 725439.99..968973.09 Rows: 2352109

	Workers Planned/Launched: 2/2	Workers Planned/Launched: 2/2
6	**Finalize GroupAggregate** Cost: 1057633.64..1886897.92 Rows: 4398180	**Finalize GroupAggregate** Cost: 725439.99..1012660.58 Rows: 1621432
7	**GroupAggregate** Cost: 1057633.64..1963869.57 Filter: count(DISTINCT b.state) >= 3 AND avg((avg(r.stars))) > 3.5 Rows Removed by Filter: 1965107	**GroupAggregate** Cost: 725439.99..1053439.08 Filter: count(DISTINCT b.state) >= 3 AND avg((avg(r.stars))) > 3.5 Rows Removed by Filter: 1443284
8	**Plan Type: Nested Loop** Cost: 1057634.07..1964055.58 Rows: 22	**Plan Type: Nested Loop** Cost: 725440.42..1053625.09 Rows: 22

Task / Query 2

Problem (queries/task_2.sql)

The task we chose aims to find the 5 users with the highest average helpful votes per review for each state. This information is useful for identifying influential reviewers or those providing particularly insightful content.

Understanding

This task demonstrates how each system handles complex aggregations and rankings. In SQL, this can be achieved using window functions like RANK() or ROW_NUMBER() to partition by state and sort by helpful votes, whereas in MongoDB, it might involve complex pipeline stages with $group, $sort, and $limit operations. This comparison highlights the differences in syntax, performance, and capabilities between the two database systems when dealing with similar data manipulation tasks.

Solution

The query begins with a CTE (user_votes) that joins the Review and Business tables to associate reviews with business locations, calculating the average number of useful votes and the total review count for each user in each state. A second CTE (ranked_users) ranks users within each state based on their average useful votes, using the RANK() window function.

```
RANK() OVER (PARTITION BY state ORDER BY avg_useful_votes DESC) AS
rank
```

Finally, the query filters for users ranked in the top 5 within their respective states.

Performance

The query plan shows an execution time of 11,141 ms, with the majority of time spent in the GroupAggregate and WindowAgg stages. The GroupAggregate operation calculates averages and review counts, taking significant time due to filtering (review_count > 10), eliminating 2.1 million rows. Parallel execution with 3 workers improved performance, but the WindowAgg stage, ranking users based on

useful votes, requires an external merge sort using 4456 KB of disk space, indicating memory constraints. The Parallel Hash Join benefits from a Parallel Index Only Scan on the idx_business_id_state index that we created earlier, avoiding a full table scan. While effective, sorting and aggregation steps could be optimized further.

Optimization

In order to optimize the runtime, we added indexes idx_review_business_id_useful and idx_review_user_id. idx_review_business_id_useful can help speed up the JOIN between Review and Business, and optimizes the calculation of AVG(r.useful) during aggregation. idx_review_user_id helps speed up the grouping by user_id.

```
CREATE INDEX idx_review_business_id_useful ON Review (business_id,
useful);
CREATE INDEX idx_review_user_id ON Review (user_id);
```

Additionally, we also increased the number of parallel workers to improve the runtime, which allows PostgreSQL to process the query faster through parallelizing expensive operations like sorting and aggregation.

```
SET max_parallel_workers_per_gather = 4
```

The updated query plan shows a significant improvement in execution time, going from 11141.408 ms to 10190.187 ms. The changes in the plan, especially the choice to increase the number of parallel workers and the reordering of operations appear to have positively impacted the performance.

PostgreSQL Performance Comparison

Step	Initial Query Plan	Updated Query Plan
1	**Parallel Hash** (...actual time=48.217..48.217 rows=50115 loops=3)	**Parallel Index Only Scan** using idx_business_id_state on "Business" b (...actual time=0.187..17.002 rows=30069 loops=5)
2	**Parallel Seq Scan** on "Review" r (...actual time=0.548..6919.451 rows=2330000 loops=3)	**Parallel Hash** (...actual time=28.123..28.125 rows=30069 loops=5)
3	**Parallel Hash Join** (...actual time=49.100..7455.886 rows=2330000 loops=3)	**Parallel Seq Scan** on "Review" r (...actual time=0.742..6776.084 rows=1398000 loops=5)
4	**Sort** (...actual time=9269.770..9560.968 rows=2330000 loops=3)	**Parallel Hash Join** (...actual time=29.259..7155.189 rows=1398000 loops=5)
5	**Partial GroupAggregate** (...actual	**Sort** (...actual time=8134.405..8325.367

	time=9269.789..9901.673 rows=1086060 loops=3)	rows=1398000 loops=5)
6	**Gather Merge** (...actual time=9278.077..10262.489 rows=3258180 loops=1)	**Gather Merge** (...actual time=8158.273..9208.246 rows=6990000 loops=1)
7	**Finalize GroupAggregate** (...actual time=9278.086..10970.268 rows=96010 loops=1)	**GroupAggregate** (...actual time=8158.302..10046.684 rows=96010 loops=1)
8	**Subquery Scan** on user_votes (...actual time=9278.087..10973.902 rows=96010 loops=1)	**Subquery Scan** on user_votes (...actual time=8158.303..10050.416 rows=96010 loops=1)
9	**Sort** (...actual time=11076.192..11090.266 rows=96010 loops=1)	**Sort** (...actual time=10131.593..10146.037 rows=96010 loops=1)
10	**WindowAgg** (...actual time=11076.211..11123.671 rows=96010 loops=1)	**WindowAgg** (...actual time=10131.612..10180.299 rows=96010 loops=1)
11	**Subquery Scan** on ranked_users (...actual time=11076.213..11125.667 rows=71 loops=1)	**Subquery Scan** on ranked_users (...actual time=10131.614..10182.331 rows=71 loops=1)

Task / Query 3

Problem

The task we chose is to find the top 5 most common words in the text column of the Review collection, excluding common stop words.

Understanding

This task illustrates how PostgreSQL and MongoDB handle text processing differently. PostgreSQL's use of functions like unnest and string_to_array shows its strength in handling string manipulation and query filtering within SQL. MongoDB's equivalent operations involve using aggregation pipelines for similar tasks, highlighting differences in syntax and functionality between the two systems. This comparison helps understand how each database system performs in terms of text analysis and aggregation capabilities.

PostgreSQL Solution (queries/task_3.sql)

The query first broke down each review's text into individual words, converting them to lowercase for uniformity.

```
WITH word_counts AS (
   SELECT unnest(string_to_array(lower(review.text), ' ')) AS word
   FROM "Review" as review
)
```

It then filters out common stop words like 'the', 'is', 'and', 'a', 'to', 'of' to focus on more meaningful words. After filtering, it counts the frequency of each remaining word and sorts these words in descending order based on their frequency, ultimately returning the top 5 most frequent words in the reviews.

PostgreSQL Performance

The query plan shows that the query performs a full scan on the Review table using a Parallel Seq Scan across 5 worker processes, processing 1.4 million rows to extract words from the reviews. The extraction generates around 147 million rows, which are filtered to exclude stop words, removing approximately 23 million rows. The remaining words are then grouped and counted in parallel using a Partial HashAggregate, which requires disk storage for memory overflow. After aggregation, the data is sorted by frequency using an external merge sort, involving disk-based operations. The query finishes in 60.7 seconds, with most time spent on the scan, aggregation, and sorting. Optimizations like indexing the Review table or using text search techniques could improve performance, as well as reducing stop words or optimizing disk usage.

Step	Operation	Description
1	Parallel Seq Scan	Scanning the entire Review table in parallel across 5 workers. Cost Estimate: 0.00..570080.00 Actual Time: 1.609..12639.681 Rows Processed: 1398000
2	ProjectSet	Projecting individual words from the reviews using a parallel sequence scan. Cost Estimate: 0.00..679298.75 Actual Time: 1.656..26303.685 Rows Processed: 147901554
3	Subquery Scan	Filtering out stop words and preparing the dataset for further aggregation. Cost Estimate: 0.00..1902548.75 Actual Time: 1.658..40980.203 Rows Processed: 124571460
4	Partial HashAggregate	Partial aggregation to count word frequencies across multiple workers. Cost Estimate: 2241563.75..2241565.75 Actual Time: 55338.032..57068.626 Rows Processed: 1734411
5	Sort	Sorting results in parallel workers, involving disk-based external merge.

		Cost Estimate: 2241573.39..2241573.89 Actual Time: 57733.873..57969.253 Rows Processed: 1734411
6	Gather Merge	Combines results from parallel workers. Cost Estimate: 2242573.45..2242669.24 Actual Time: 57817.145..59335.139 Rows Processed: 8672056
7	Finalize GroupAggregate	Final aggregation of word counts, after parallel processing. Cost Estimate: 2242573.45..2242675.24 Actual Time: 57817.172..60190.556 Rows Processed: 5585060
8	Sort	Sorting words by frequency in descending order using a top-N heapsort. Cost Estimate: 2242678.56..2242679.06 Actual Time: 60490.912..60580.495 Rows Processed: 5
9	Limit	Limits the result to the top 5 words after sorting by frequency. Cost Estimate: 2242678.56..2242678.57 Actual Time: 60490.913..60580.498 Rows Processed: 5

PostgreSQL Optimization

In order to optimize the query, we tried to use PostgreSQL's built-in full-text search capabilities, by creating a full-text-search index. However, creating this index is very inefficient as text has values with more than 2047 characters, which is the maximum allowed length for indexing. Thus, continuing creating the index can result in incomplete or inaccurate search results, as review text often contains long URLs or other lengthy strings.

Task / Query 4

Understanding

When creating this query we learn PostgreSQL handles structured data through GROUP BY for aggregations and JSON operators (e.g., ->>) for accessing nested fields. In MongoDB, we tried aggregation concepts, such as $group for counting and $match for filtering. In the last part where we combine related datasets, SQL joins operation works similarly with MongoDB's $lookup.

Problem

The SQL query analyzes the Business table offering takeout services across different states. By summarizing the count of these businesses and comparing them to the total number of businesses in each state, we could have insight into business characteristics and geographical distribution. This query

demonstrates the ability of PostgreSQL to handle complex joins, aggregations, and sorting, thereby making it efficient for large-scale data analysis.

PostgreSQL Solution (queries/task_4.sql)

We first created 2 CTE (BusinessCounts and TakeOutCounts) where we used GROUP BY states to calculate the total number of businesses and the number of businesses offering takeout. Then we JOIN the two CTEs trying to find the takeout_proportion.

```
SELECT
   b.state,
   (t.takeout_businesses * 1.0 / b.total_businesses) AS
takeout_proportion
 FROM BusinessCounts b
 JOIN TakeOutCounts t ON b.state = t.state
```

Finally, we descend order takeout_proportion and find the 5 states with the highest takeout proportions.

PostgreSQL Performance

The query uses a Parallel Seq Scan of the table of Business and a filtered condition for the Business table attribute: RestaurantTakeOut is equal to true. Parallel scans across many workers are to help in large dataset processing. However, a great number of records in the dataset do not satisfy the filtering condition. Such could potentially impact performance, particularly if many rows lack the RestaurantsTakeOut attribute or have it set to false. In addition, the query uses a Merge Join between the aggregated results of BusinessCounts and TakeOutCounts on the field state. This is an efficient join method assuming that the data is pre-sorted or indexed which helps in lowering the overhead of costly join operations. Lastly, there are several levels of Sort Operations (using quicksort), and Group Aggregate functions which indicate PostgreSQL is organizing and grouping the data prior to final aggregation. The layering process guarantees that data is aggregated and can be used for final processing, even as it makes an indirect point that lots of computational resources are required for sorting and grouping.

PostgreSQL Optimization (queries/task_4_optimized.sql)

In order to improve the runtime, we decided to create two materialized views, BusinessCounts (to store the total number of businesses in each state) and TakeOutCounts (to store the count of businesses offering takeout services in each state). Materialized views can optimize query performance in scenarios where data does not change frequently. By precomputing and storing the results of the query, materialized views allow access to pre-aggregated data and thus reducing execution time for complex/further joins or aggregations. The materialized view of BusinessCounts aggregates the total number of businesses per state execution at 82.126 ms. Quite similarly, in cases where materialized view the TakeOutCounts computes the count of businesses offering takeout per state execution at 76.223ms.

The use of materialized views is applicable here because the state-wise business counts are unlikely to change frequently. This reduces the need for recalculating aggregations, saving computational resources and query time. However, if the updated data is constantly changing, the cost of refreshing materialized views may become significant in terms of both processing speed and storage. Therefore, the decision to use materialized views should carefully appraise the balance between the frequency of data

updates and performance gains.

The updated query plan shows a significant improvement in execution time, going from 117.387 ms to 0.205 ms.

PostgreSQL Performance Comparison

Step	Initial Query Plan	Updated Query Plan
1	Filter (RestaurantsTakeOut) (Filter: ((attributes ->> 'RestaurantsTakeOut'::text) = 'True'::text)) (Rows Removed by Filter: 32468)	N/A
2	Parallel Seq Scan (Business_1) (cost=0.00..14705.66 rows=313 width=3) (actual time=0.602..26.381 rows=17648 loops=3)	N/A
3	Sort (State) (cost=14718.64..14719.42 rows=313 width=3) (actual time=29.493..29.860 rows=17648 loops=3)	N/A
4	Partial GroupAggregate (cost=14718.64..14721.15 rows=17 width=11) (actual time=29.544..30.561 rows=15 loops=3)	N/A
5	Gather Merge (State) (cost=15718.66..15725.10 rows=34 width=11) (actual time=32.277..33.646 rows=44 loops=1)	N/A
6	Finalize GroupAggregate (state) (cost=15718.66..15725.44 rows=17 width=11) (actual time=32.334..33.654 rows=16 loops=1)	N/A
7	Parallel Seq Scan on "Business" business (cost=0.00..14392.44 rows=62644 width=3) (actual time=0.256..47.930 rows=50115 loops=3)	N/A
8	Partial HashAggregate (cost=14705.66..14705.83 rows=17 width=11) (actual time=70.596..70.597 rows=20 loops=3)	N/A
9	Sort (state) (cost=14706.18..14706.22 rows=17 width=11) (actual time=70.831..70.832 rows=20 loops=3)	Sort (state) (cost=83.37..86.37 rows=1200 width=40) (actual time=0.042..0.044 rows=27 loops=1)

10	Gather Merge (cost=15706.20..15710.17 rows=34 width=11) (actual time=79.114..79.159 rows=59 loops=1)	N/A
11	Finalize GroupAggregate (cost=15706.20..15710.51 rows=17 width=11) (actual time=79.118..79.169 rows=27 loops=1)	N/A
12	Merge Join (cost=31424.86..31436.72 rows=17 width=35) (actual time=111.999..113.379 rows=16 loops=1)	Merge Join (cost=166.75..352.75 rows=7200 width=64) (actual time=0.054..0.055 rows=0 loops=1)
13	Sort (cost=31437.00..31437.04 rows=17 width=35) (actual time=113.132..113.401 rows=5 loops=1)	Sort (cost=472.34..490.34 rows=7200 width=64) (actual time=0.140..0.142 rows=5 loops=1)
14	Limit(cost=31437.00..31437.01 rows=5 width=35) (actual time=113.133..113.402 rows=5 loops	Limit (cost=472.34..472.35 rows=5 width=64) (actual time=0.143..0.145 rows=5 loops=1)

Task / Query 5

Problem (queries/task_5.sql)

The task is designed to determine if the total number of reviews a user wrote correlates to their average star. The purpose is to find out whether a considerable relationship exists between the number of reviews a user submits and the rating he gives on average. Users are grouped by the number of reviews they have written into three baselines: low, medium, or high, and their average ratings are computed, with correlation coefficients calculated between review count and average rating.

Understanding

This query demonstrated PostgreSQL's ability to handle advanced analytics with techniques like window functions (e.g., NTILE) for ranking and conditional multiple levels (e.g. CASE). Where MongoDB was not applicable because it lacks built-in support for the techniques listed above. Solving query 5's problem required manually computing complex formulas.

Solution (steps)

In the first CTE (review_bins), we use GROUP BY to calculate the number of reviews and average rating for each user, grouping data at the user level. In the second CTE (ranked_reviews), we use NTILE to rank users into categories based on their review count, enabling us to divide them into quantile-based groups. In the last CTE (categorized_reviews), we applied a CASE statement to assign descriptive labels (e.g., 'Low Review Count') to these categories, making the data into three levels of groups. Finally, we compute summary statistics with CORR, AVG, COUNT(*).

```
NTILE(3) OVER (ORDER BY num_reviews) AS review_count_category_rank
```

Performance

The query plan shows an execution time of 15,541 ms, mainly due to sorting and hash aggregation stages. It takes 15,418 ms approximately to sort review counts by categories and at the expense of 25 KB memory, while 40 KB memory is also needed for aggregation. Furthermore, a window aggregate requires an external merge sort and 47,088 KB of disk space. Parallel execution with two workers can speed up the process, but disk usage against the sorting is immense (78,016 KB).

Optimization

Creating indexes may not provide significance because this query requires the completion of complex aggregations, sorts, and window functions on large datasets. Additionally, the indexed column might not be selective enough, the database engine might have determined that a sequential scan was more efficient. When we increased the work_mem to 256 MB, it provided more memory for working with sorts and hashes.

```
SET work_mem = '256MB';
```

The query can be more efficient because it can keep intermediate results in memory rather than having to write them onto a disk. This results in less usage of the disk and more efficient memory allocation for these operations; hence, the execution time is reduced from 15418 to 14502 milliseconds. Although the query took quite some time, we saw a 1000 ms improvement suggesting that the increased memory allocation helped the process of things happening inside memory instead of writing them to disk.

PostgreSQL Performance Comparison

Step	Initial Query Plan	Updated Query Plan
1	Parallel Seq Scan on "Review" reviews (actual time=0.932..8784.215 rows=2330082 loops=3)	Parallel Seq Scan on "Review" reviews (actual time=0.932..8784.215 rows=2330082 loops=3)
2	Partial HashAggregate (actual time=10848.440..11016.838 rows=1009311 loops=3)	Partial HashAggregate (actual time=10848.440..11016.838 rows=1009311 loops=3)
3	Gather (actual time=10862.886..11305.161 rows=3027933 loops=1)	Gather (actual time=10862.886..11305.161 rows=3027933 loops=1)
4	Finalize HashAggregate (actual time=12522.373..13357.539 rows=1987897 loops=1)	Finalize HashAggregate (actual time=12522.374..13443.389 rows=1987897 loops=1)
5	Sort (actual time=12942.726..14780.300 rows=1987897 loops=1)	Sort (actual time=13653.895..13701.249 rows=1987897 loops=1)

6	WindowAgg (actual time=12942.726..14780.300 rows=1987897 loops=1)	WindowAgg (actual time=13818.318..14028.449 rows=1987897 loops=1)
7	Sort (actual time=15418.133..15418.172 rows=3 loops=1)	Sort (actual time=14502.002..14502.044 rows=3 loops=1)
8	HashAggregate (actual time=15418.117..15418.158 rows=3 loops=1)	ashAggregate (actual time=14501.915..14501.959 rows=3 loops=1)
9	Subquery Scan on ranked_reviews (actual time=15418.133..15418.172 rows=3 loops=1)	Subquery Scan on ranked_reviews (actual time=14502.002..14502.044 rows=3 loops=1)

Non-Relational Tools Comparison: MongoDB

Task / Query 1

Problem

The problem being solved is finding users who have written reviews for businesses in 3+ states and have an average review star rating of more than 3.5.

MongoDB Solution (queries/task_1.py)

This was solved through joining reviews and the business collection to determine which states were reviewed. Next, grouping was used to find the average star rating across the states. Filtering was applied to find the users who reviewed in more than 3 states and average review star ratings of more than 3.5. A 1000 limit was placed because it would take too long in the beginning of the query to prevent it from running too long.

MongoDB Performance of Original Query

The query's performance is limited by the extensive execution time of approximately 67.8 seconds, largely due to the inefficiency of the $lookup stage that joins the review and business collections, processing a vast amount of data relative to a small input size. The $group and $match stages also contribute to the high execution time, which involves memory-intensive operations and aggregation tasks. The final $lookup to the user collection yielded no results, indicating potential match condition issues.

MongoDB Optimization

To optimize the query, we employed indexing on critical fields (business_id, user_id, and state) in the review, business, and user collections, enhancing lookup and match efficiency. A $limit stage at the start reduced the dataset to 1000 documents, easing workload for subsequent stages. Simplified $lookup projections minimized data pulled from business and user collections, and optimized grouping minimized memory usage. As a result, the query's execution time decreased to approximately 188 milliseconds, processing only 1000 documents.

```
db.review.createIndex({ business_id: 1, user_id: 1 });
db.business.createIndex({ business_id: 1, state: 1 });
db.user.createIndex({ user_id: 1 });
pipeline: [
              { $match: { $expr: { $eq: ["$business_id",
"$$business_id"] } } },
              { $project: { state: 1 } }
           ]
```

MongoDB Performance Comparison

Step	Initial Query Plan	Updated Query Plan
1	$cursor: Retrieved the 'review' collection. totalDocsExamined: 1000, executionTimeMillis: 67899	$cursor: Retrieved the 'review' collection. totalDocsExamined: 1000, executionTimeMillis: 188
2	$lookup (business): Joined review and business. executionTimeMillisEstimate: Long('67877')	$lookup (business): Joined review and business. Used 'business_id_1_state_1' index. executionTimeMillisEstimate: Long('173')
3	$group (avg_star by user, state): Average star rating. executionTimeMillisEstimate: Long ('67889')	$group (avg_star by user, state): Average star rating. executionTimeMillisEstimate: Long('175')
4	$group (overall_avg_rating): Found the overall rating of the state and formed an array of states. executionTimeMillisEstimate: Long('67891')	$group (overall_avg_rating): Found the overall rating of the state and formed an array of states. executionTimeMillisEstimate: Long('67891')
5	$match: Filters the user requirement of 3 or more stats and rating of 3.5 or above. executionTimeMillisEstimate: Long('67893')	$match: Filters the user requirement of 3 or more stats and rating of 3.5 or above. executionTimeMillisEstimate: Long('100')
6	$lookup: Joins user. executionTimeMillisEstimate: Long('67893')	$lookup: Joins user. executionTimeMillisEstimate: Long('180')
7	$project: *id, uer_id, state_count, overall_avg_rating, user_info*	$project: *id, uer_id, state_count, overall_avg_rating, user_info*

	executionTimeMillisEstimate: Long('67893')	executionTimeMillisEstimate: Long('180')

Task / Query 2

Problem

Find the top 5 users with the highest average helpful votes per review for each state.

MongoDB Solution (queries/task_2.py)

The problem was solved with preliminary selection of the first 1,000 reviews before scan. The review collection is then joined with the business collection using $lookup to extract the state information, followed by an $unwind stage to flatten the resulting array. The state field is then added to each review document.

```
{ "$lookup": {
        "from": "business",
        "localField": "business_id",
        "foreignField": "business_id",
        "as": "business_info"}
 },
 { "$unwind": "$business_info"},
         {"$addFields": {"state": "$business_info.state"}}, ...
```

A second $lookup is performed to join the review collection with the user_info collection to retrieve user details like average_stars and yelping_since, followed by another $unwind stage.

The pipeline groups the data by user_id and state, calculating the total number of useful votes, total reviews, average stars, the time the user has been on Yelp, and eventually a division to get targeted results.

MongoDB Performance of Original Query

The query involves several stages that contributed to a significant execution time. The total time for execution was 319.5 seconds, with each stage's individual execution time varying based on the complexity and the amount of data processed. For instance, the $lookup and $group stages, which involved scanning over 300 million documents, accounted for the majority of the processing time. We attempted reducing computing costs by truncating the dataset early in the pipeline using $limit and performing sorting and filtering before grouping.

MongoDB Optimization

Optimizing this MongoDB query is challenging due to its expensive operations such as $lookup joins, which result in full collection scans and examine millions of documents. The query generates large intermediate results, significantly increasing memory and CPU usage, particularly during $unwind and array manipulations like $sortArray and $slice. Additionally, MongoDB lacks native support for ranking, forcing inefficient emulation through grouping and sorting. Despite limiting the input to 1000 documents, the execution plan shows prolonged processing times and resource-intensive stages, highlighting

MongoDB's limitations in handling such relational queries. These challenges originated from MongoDB's design, which prioritizes document-based operations over complex relational workloads.

MongoDB Performance

Step	Initial Query Plan
1	**Limit**: limits the documents to 1000, reducing the dataset for subsequent stages
2	**Lookup**: joins the review collection with business to retrieve business details (e.g., state). This step uses a full collection scan
3	**Unwind**: flattens the business_info array, ensuring each review is joined with business information
4	**addField**: adds the state field from business_info to the documents
5	**Second Lookup**: joins the review collection with the user_info collection to retrieve user details (e.g., average_stars, yelping_since). This also uses a full collection scan
6	**Flatten:** flattens the user_info array, ensuring each review is associated with the user details
7	**Group:** groups the documents by user_id and state, calculating the total useful votes and the total number of reviews for each user
8	**Second addField:** calculates the average helpful votes per review (avg_helpful_votes)
9	**Group:** further groups by state, pushing the top users into an array for each state
10	**Project:** sorts the top_users array by avg_helpful_votes and limits the results to the top 5 users per state

Task / Query 3

Problem

The task we chose is to find th3 top 5 most common words in the text column of the Review collection, excluding common stop words.

MongoDB Solution (queries/task_3.py)

To solve this query, we first split each review's text into individual words, converting them to lowercase, and unwinding the resulting arrays. The $match stage filters out the stopwords to focus only on meaningful words. Subsequently, the $group stage counts occurrences of each word, aggregating them under a unique _id. Finally, the words are sorted in descending order of frequency, and the top five are selected.

MongoDB Performance

The query scans all 6,990,280 documents in the collection (COLLSCAN), resulting in a high

execution time of 471,502 milliseconds. The $unwind stage significantly increases the workload, expanding the dataset to over 739 million intermediate documents. Additionally, the $group stage requires substantial memory for counting occurrences, spilling to disk multiple times due to exceeding in-memory limits. While the query successfully retrieves the desired top five words, its reliance on full collection scans, extensive unwinding, and disk usage indicates inefficiency.

MongoDB Optimization

Our optimizing plan focuses on reducing the intermediate dataset size early in the aggregation pipeline. By incorporating the $filter operator within the $project stage, the query filters out stopwords ("the", "is", "and", "a", "to", "of") during the initial projection. This minimizes the number of words passed to subsequent stages, particularly $unwind, which previously expanded the dataset significantly. By moving this filtering logic earlier, the query avoids unnecessary processing of irrelevant words, improving efficiency. This change led to a notable reduction in execution time from 471,502 milliseconds to 338,599 milliseconds, despite retaining similar resource usage for grouping and sorting stages.

```
$filter: {
          input: { $split: [{ $toLower: "$text" }, " "] },
          as: "word",
          cond: { $not: { $in: ["$$word", ["the", "is", "and", "a",
"to", "of"]] } }
        }
```

MongoDB Performance Comparison

Step	Initial Query Plan	Updated Query Plan
1	Project: $split separates lowercase text into separate words. Execution time: 471,502 ms. totalDocsExamined: 6990280	Project: $filter keeps words from stop words list from the lowercase text. Execution time: 338,599 ms. totalDocsExamined: 6990280
2	Unwind: Expand ‘words’ array. A document created for each word. 739,548,008 documents returned.	Unwind: Expand ‘words’ array. A document created for each word. 622,891,257 documents returned.
3	Match: Filters documents with stop words. 622,891,257 documents returned.	N/A
4	Group: Find count for each word after grouping documents by word. 5,585,282 documents returned. maxAccumulatorMemoryUsageBttes: count: Long(‘80786800’), “usedDisk: true”, “spills”: Long(‘11’), Execution time: 471,036 ms	Group: Find count for each word after grouping documents by word. 5,585,282 documents returned. maxAccumulatorMemoryUsageBttes: count: Long(‘80786800’), “usedDisk: true”, “spills”: Long(‘11’), Execution time: 338,018 ms

5	Sorting: Sorts grouped words in descending order of count. Top 5 are limited. usedDisk: false", "spills": Long('0'). Execution time: 471,500 ms.	Sorting: Sorts grouped words in descending order. Top 5 are limited. usedDisk: false", "spills": Long('0'). Execution time: 338,565 ms.

Task / Query 4

Problem

This task aims to identify the top 5 states with the highest proportion of businesses offering "RestaurantsTakeOut" relative to total businesses

MongoDB Solution (queries/task_4.py)

Imitating the Postgre strategy for this task, we aggregated the business collection to count the total number of businesses per state, creating a new collection that contains total businesses for each state.

```
businessCounts_pipeline = [
    {"$group": {...}},
    {"$merge": { "into": "BusinessCounts" } }]
```

Then, we created a separate collection for businesses that offer takeout.

```
takoutCounts_pipeline = [
    {"$match": { "attributes.RestaurantsTakeOut": "True" } },
    {"$group": {...}},
    {"$merge": { "into": "TakeOutCounts" }}]
```

We lastly performed a $lookup operation to join the data from the two collections (BusinessCounts and TakeOutCounts). The query calculates the takeout proportion by dividing the number of takeout businesses by the total businesses in each state. Finally, the results are sorted and limited to the top 5 states based on the takeout proportion.

MongoDB Performance of Original Query

The query execution time is 35.7 seconds, with approximately 300,692 documents processed in the business collection. The query includes multiple stages such as $group, $lookup, $unwind, $addFields, and $sort. Despite using $lookup, no index on the fields involved (e.g., state) was used, leading to a full collection scan (COLLSCAN), which significantly increased the query's execution time. The $lookup stage joins the BusinessCounts and TakeOutCounts collections, but since the query performs full collection scans, there is no indexing optimization.

The total execution time was broken down as follows:

- $lookup: 35.7 seconds, matching states between the two collections.
- $addFields and $sort: 35.7 seconds, applied to calculate the takeout proportion and sort the states.
- Total docs examined: 300,692.

MongoDB Optimization (queries/task_4_optimized.py)

Index creation failed to optimize MongoDB operation as the actual execution time increases instead. Indexes are not beneficial in these cases because $group requires scanning all documents to compute the aggregates in the initial pipelines (businessCounts_pipeline and takeoutCounts_pipeline). Using an index may introduce additional overhead for accessing the indexed entries and combining them with unindexed fields.

The improved query adopts query restructuring instead of indexing. This eliminates the need for a join operation by directly aggregating the original collection with $addFields to calculate a flag (isTakeOut) for businesses offering "RestaurantsTakeOut." The method avoids excessive data duplication and reduces the query complexity and memory usage, improving the execution time to approximately 1 second.

```
{"$addFields": {
       "isTakeOut": { "$cond": {
                "if": {"$eq": ["$attributes.RestaurantsTakeOut",
"True"]},
                "then": 1,
                "else": 0}}}}
```

MongoDB Performance Comparison

Step	Initial Query Plan	Updated Query Plan
1	**Project**: The query first performs a **projection** to select specific fields, including takeout_counts.takeout_businesses, takeout_proportion, and total_businesses. This stage uses a **COLLSCAN** (collection scan) to read through all documents. **Execution time**: 21068 ms. **Total documents examined**: 300,692.	**addField**: Adds a new field isTakeOut using $addFields to flag businesses offering "RestaurantsTakeOut." **Execution time:** 811 ms. **Documents examined:** 300,692.
2	**Lookup**: The **$lookup** stage performs a join with the TakeOutCounts collection, matching documents by the _id field. The matching data is stored in an array field takeout_counts. This stage uses an **index on _id_**. **Execution time**: 21068 ms. **Documents examined**: 0.	**Group**: Groups businesses by state using $group, aggregating the counts of isTakeOut and total businesses. **Execution time:** 1,020 ms. **Documents returned:** 27.
3	**Add Fields**: This stage calculates the takeout_proportion field using the formula: takeout_counts.takeout_businesses / total_businesses. The calculated value is added to each document. **Execution time**: 21068 ms. **Documents returned**: 0.	**Second addField**: The query calculates the takeOutProportion field by dividing the takeOutCount by the totalCount for each state. This proportion is added to each grouped document. **Execution**

		time: 1,020 ms. **Documents returned:** 27.
4	**Sort**: The **$sort** stage sorts the documents by takeout_proportion in descending order and limits the result to the top 5. No disk usage or spills occurred during this sorting operation. **Execution time**: 21068 ms. **Documents returned**: 0.	**Sort**: sorts the states by takeOutProportion in descending order using $sort. **Execution time:** 1,020 ms. **Documents returned:** 27.
5	**Project**: In the final projection, only the takeout_proportion and state fields are returned, while the _id field is excluded. **Execution time**: 21068 ms. **Documents returned**: 0.	**Limit**: limits the results to the top 5 states using $limit. **Execution time:** 1,020 ms. **Documents returned:** 5.

Tool Comparisons and Fitness/Ergonomics

Task / Query 1

PostgreSQL, leveraging its relational model and aggregation capabilities, efficiently joined and aggregated data across tables, leading to a reduction in execution time from 12,322 ms to 10,921 ms after optimisation. This efficiency stemmed from its structured nature and ability to perform complex joins and aggregations inherent to its design.

On the other hand, while MongoDB offered flexibility with its schema-less structure, it faced challenges with high execution times due to inefficient $lookup operations and memory-intensive aggregations. Significant optimization efforts, including indexing and query restructuring, were required to achieve a more reasonable execution time of 188 ms. This highlights MongoDB's suitability for flexible, schema-less tasks and nested data structures, but its performance can be hampered by complex joins and aggregations, especially when large datasets are involved.

Task / Query 2

PostgreSQL, with its relational model and dedicated functions like RANK(), efficiently partitions and sorts data. This structured approach makes the query logic easier to comprehend and maintain. This efficiency is reflected in its performance, as the initial query execution took 11,141 ms, which was further reduced to 10,190.187 ms after optimizations involving indexing and increasing parallel workers.

MongoDB's document-based model lacks native ranking support, demanding a more complex approach involving multiple $lookup, $unwind, and array manipulations. This complexity is reflected in its performance, with the initial query taking a substantial 319.5 seconds. While limiting the input to 1000 documents improved the execution time, the fundamental inefficiencies persisted, highlighting MongoDB's limitations in handling complex relational queries on large datasets.

Task / Query 3

The PostgreSQL solution demonstrated efficiency in handling this complex text manipulation task, showcasing the power of SQL functions and indexing capabilities. Utilizing functions such as

unnest, the query broke down reviews into individual words and filtered out stop words, demonstrating PostgreSQL's strength in string manipulation and query filtering within a structured query language. This structured approach allowed for efficient processing, completing the task in 60.7 seconds, even when dealing with a large dataset of 1.4 million reviews generating approximately 147 million words.

MongoDB faced performance challenges due to its reliance on full collection scans and extensive unwinding of arrays, leading to a significantly longer execution time of 471,502 milliseconds. The $unwind stage, which separated individual words, expanded the dataset dramatically, highlighting a key issue for MongoDB when handling this task. While optimization strategies, such as filtering stop words earlier in the pipeline, reduced execution time to 338,599 milliseconds, MongoDB still exhibited memory issues during aggregation, indicating its vulnerability to intensive text processing operations. This comparison underscores the strengths and weaknesses of each system, with PostgreSQL excelling in structured text manipulation tasks within smaller datasets and MongoDB showing potential for improvement in handling large-scale text processing.

Task / Query 4

PostgreSQL, designed for structured data and complex querying, efficiently utilizes Common Table Expressions (CTEs) and joins to handle aggregations and comparisons. The initial query execution took 117.387 ms, with Parallel Seq Scans and Merge Joins contributing to its efficiency. However, the presence of multiple Sort and Group Aggregate functions suggests potential optimization opportunities. We implemented materialized views - BusinessCounts and TakeOutCounts - to precompute and store state-wise business counts, which are not expected to change frequently. This strategy significantly reduced the execution time to 0.205 ms. This approach aligns with industry practices where pre-aggregation techniques are employed for frequently accessed analytical queries, especially in data warehousing scenarios.

Although MongoDB could accomplish this task, it exhibited performance limitations. The initial query, relying heavily on $group, $lookup, and $sort operations, took 35.7 seconds to execute, processing 300,692 documents. The reliance on full collection scans, due to the absence of suitable indexes, significantly impacted performance. Attempts to optimize using indexes proved ineffective, as the $group stage necessitates scanning all documents. A more successful strategy involved restructuring the query, eliminating the join operation and directly calculating the "RestaurantsTakeOut" flag using $addFields. This reduced the execution time to approximately 1 second, demonstrating that query restructuring can be more beneficial than indexing in certain MongoDB scenarios.

Task / Query 5

This task presented a stark contrast in how PostgreSQL and MongoDB handle analytical tasks. PostgreSQL, with its support for advanced analytics techniques, successfully executed the query using window functions like NTILE for ranking and CASE statements for conditional logic. Initially, the execution time was 15,541 ms, involving sorting, hash aggregation, and window functions on a large dataset. We optimized the query through increasing memory allocation (work_mem), which reduced execution time to 14,502 milliseconds. This showcases PostgreSQL's strength in handling analytical queries that require the manipulation and analysis of structured data.

However, replicating this analysis in MongoDB was challenging due to its lack of built-in support for equivalent analytical functions like NTILE and CASE. This limitation would necessitate a

workaround involving manually computing complex formulas, potentially leading to a complex and inefficient query structure.

Summary

MongoDB, as a NoSQL document-based database, is effective in handling unstructured data and dynamic schemas. It is relatively easy to install and set up, with a flexible query language that supports JSON-like documents. Learning MongoDB is intuitive for those familiar with JavaScript, but writing complex queries like joins and aggregations can become inefficient and costly.

PostgreSQL works better in structured data environments with support for SQL queries, joins, and transactional consistency. Learning PostgreSQL's can be challenging in terms of its reliance on strict schemas and more intricate setup. However, it is highly performant for queries involving relational data and analytics, tasks that MongoDB may struggle with due to its lack of native joins and reliance on manual aggregations. PostgreSQL enables tasks like ranking or complex filtering, which can be challenging or inefficient in MongoDB.

Team Reflections

PostgreSQL is most ideal for tasks requiring complex queries, data integrity, and structured relationships between tables. Its support for complex joins, subqueries, and advanced filtering is helpful in manipulating structured relational data in tasks like aggregating reviews or analyzing user interactions. Its ACID compliance ensured data integrity during transaction-heavy operations, which was crucial for maintaining consistency, especially when processing customer data.

In a scenario like building a customer relationship management (CRM) system, PostgreSQL works best due to its need for complex tasks, such as generating detailed reports across customer profiles, orders, and interactions.

However, PostgreSQL required more effort in setting up the system and maintaining the schema, particularly for evolving datasets, and performance could degrade when handling large-scale unstructured data. On the other hand, **MongoDB** provides a flexible schema, which makes it ideal for handling large volumes of unstructured/semi-structured data like the Yelp dataset. Its ability to scale horizontally across distributed nodes was critical as the dataset grew to billions of records.

For instance, during a project to log IoT sensor data, MongoDB proved invaluable for its flexibility in handling JSON-like documents and dynamic schemas, where new fields could be added without migrations.

MongoDB's querying is less powerful compared to PostgreSQL, especially deep joins or aggregations. Additionally, debugging is more inconvenient due to less comprehensive error handling. For similar projects, We recommend PostgreSQL for structured data and complex querying, while MongoDB for projects with unstructured, scalable data needs.

Individual Reflections

Team Member 1 – Michelle Dong

In this project, I gained insights into query optimization, especially the impact of parallel execution on performance. Observing how increasing the number of parallel workers reduced runtime for tasks like sorting, aggregation, and joining large datasets, I adjusted PostgreSQL settings like max_parallel_workers_per_gather for significant execution time improvements. I also learned that SQL is ideal for relational queries, while MongoDB excels in handling unstructured or semi-structured data. Comparing these databases was both challenging and enlightening, as it helped deepen my understanding of course concepts.

Team Member 2 – Kelly He

For the coding part, I mainly worked on making the queries and optimizing MongoDB, and my biggest takeaway from this project was understanding how to optimize MongoDB. Trying out different stages within aggregation such as $lookup and $unwind helped me learn about MongoDB's fundamental usage. The most exciting part of the project was seeing how PostgreSQL and MongoDB are compared, specifically seeing how PostgreSQL is well suited for joining and aggregations and how MongoDB is flexible. It was challenging to find queries that worked for both tools, especially the fourth task as the aggregations made it harder to code.

Team Member 3 – Claudia Wu

I have gained practical knowledge on how to utilize database tools to solve data problems in user and business insights, from system setup, ETL, to writing queries with appropriate systems. During the process of writing efficient pipelines and producing desired insights, I learned a lot about how MongoDB's aggregation framework works, particularly how lookup, unwind, and group stages can be combined to join and aggregate data from different collections. The most challenging parts were dealing with the performance issues caused by the large number of records and making sure PostgreSQL queries translate accurately and efficiently to MongoDB queries, since two systems have different specializations.

Team Member 4 – Emma Ye

In this project, I gained hands-on experience with optimizing query performance. I realized how significant materialized views can be in optimizing execution time, and when or how to apply them effectively. Through repeatedly analyzing query execution plans, I started to grasp the different strategies for optimizing queries and connecting them to the concepts we learned in class, such as CTEs and materialized views, considering their respective pros and cons. This means I explored the trade-off between faster query execution and the potential costs of updating data.