

# Data101 Final Project

## Final Report

Manan Bhargava, Bianca Isabel Poblano, Nawoda Wijesooriya

Manan Bhargava	3036535321	manan.bhargava@berkeley.edu
Bianca Isabel Poblano	3035917442	bpoblano@berkeley.edu
Nawoda Wijesooriya	3037027827	nawodakw@berkeley.edu

## Dataset Selection

For this project, we selected the **Yelp Open Dataset**, a collection of real-world data containing information about businesses, reviews, users, check-ins on businesses, tips written by users on businesses, and photo data. This dataset was selected because it provides a rich source of real-world data that can be analyzed for various purposes such as business performance evaluation and user analysis.

The data is freely available on Yelp's official website (<https://www.yelp.com/dataset>). We downloaded the data in JSON format and focused on 3 key files for our project:

- **Business.json**: Containing details about the businesses such as name, location, and attributes
- **Users.json**: Stores user-specific information like review counts and ratings
- **Reviews.json**: Includes detailed reviews with ratings, text, and timestamps

We chose not to use the checkins.json, tip.json, or photo.json datasets because those are not relevant to the questions we strive to answer or queries we strive to construct. This will be further discussed in the **Task Selection** section.

### File Sizes, Hardware Specs, and Load Times

We executed the following processes to load the datasets through running Python via Jupyter Notebook on a 2020 Macbook Air 13.3" with 8 GB memory, 512 GB SSD storage, Apple M1 Chip, and macOS Sequoia 15.1. This setup provides a reference for understanding the load times and processing requirements. Below is a summary of file

sizes, load times, and export times, providing context for our reasoning to retain the entire original dataset or sample any particular dataset.

Dataset	Rows	Columns	File Size	Load Time	Exported CSV File Size	Export Time
business	150,346	14	118.9 MB	8.5 seconds	99.1 MB	5.2 seconds
users	1,987,897	22	3.36 GB	3 minutes 21 seconds	N/A (retained full dataset)	N/A
reviews	6,990,280	9	5.34 GB	9 minutes 52 seconds	1.1 GB (sampled)	1 minute 42 seconds (sampling + export)

## Sampling the Datasets

We did not sample users.json or business.json because referential integrity needed to be preserved between users and reviews and business and reviews. This prevents orphaned records in the reviews dataset.

However, we sampled the reviews.json dataset because it was significantly larger than the others, containing 6,990,280 rows with a file size of 5.34 GB. This size posed challenges for processing and analysis due to memory and runtime constraints.

## Developing a Representative Sample

To develop a representative sample, we developed the `sample_data` function. This function:

- 1) **Randomly Samples from each chunk:** Random sampling ensures each row in the dataset has an equal probability of being selected, reducing bias and retaining distributions of features (e.g. star ratings, review lengths). By using the `chunk.sample()` function the method selects rows randomly from each chunk.
- 2) **Stratified Sample:** By reading the dataset in chunks, this simulates sampling across the entire file, creating a stratified effect ensuring diversity in the sampled data.
- 3) **Random Seed:** The `random_state` parameter guarantees reproducibility. Every time the function runs with the same `random_state`, the sampled dataset will be identical. This is critical for consistent debugging, testing, and validation.

The following code illustrates the iterative process of looping through the chunks and constructing the sampled data.

```
Python
for chunk in data_reader:
    # Randomly sample rows from the current chunk and append to dataframe
    sampled_data = pd.concat(
        [sampled_data, chunk.sample(n=sample_per_chunk,
            random_state=random_state)], ignore_index=True
    )

    if len(sampled_data) >= sample_size:
        break
```

Verifying the Sample is Representative of the Original Dataset

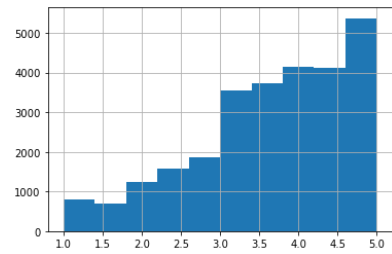
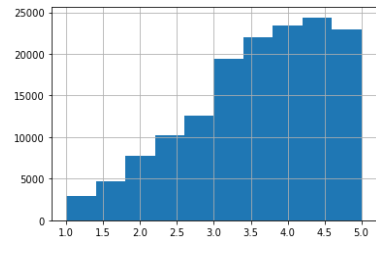
We decided to sample 1 million reviews from the review dataset to have a “large enough” sample that is representative of the original dataset. To ensure the sample was representative, we calculated distribution metrics for key attributes such as review\_id, business\_id, stars, useful, funny, cool, and date. By comparing the distribution metrics of the full review dataset with those of the sampled data, we confirmed that the distributions were nearly identical, validating the representativeness of the sample.

To validate representativeness, we performed exploratory analysis on key features (e.g., star ratings, review lengths, and business categories) to confirm that their distributions in the sample closely matched those in the full dataset.

We leveraged 2 metrics:

- 1. Average Stars
- 2. Counts of number of reviews per business

Metric 1: Average Stars

Average Stars for Sampled Dataset	Average Stars for Overall Dataset	Difference in Average Stars
 count 27095.000000 mean 3.656561	 count 150346.000000 mean 3.587534	count -123251.000000 mean 0.069027 std 0.047344 min 0.000000 25% 0.051282 50% 0.090909 75% 0.111111 max 0.000000

std	1.009151	std	0.961807	Name: stars, dtype: float64  Each difference is ~1-2% of the total rating, which is statistically small enough to be considered a representative sample.
min	1.000000	min	1.000000	
25%	3.000000	25%	2.948718	
50%	3.818182	50%	3.727273	
75%	4.444444	75%	4.333333	
max	5.000000	max	5.000000	
Name: stars, dtype: float64		Name: stars, dtype: float64		

## Metric 2: Counts of number of reviews per business

	review_count_sampled	review_count_overall	review_count_prop_diff
count	27095.000000	150346.000000	-1.017057e-01
mean	36.907178	46.494619	8.617257e-04
std	98.996153	124.519061	2.312630e-03
min	1.000000	5.000000	-2.855774e-07
25%	6.000000	8.000000	1.373018e-04
50%	11.000000	15.000000	2.496137e-04
75%	30.000000	38.000000	6.991467e-04
max	4661.000000	7673.000000	9.744558e-02

Each difference is very close to zero, illustrating the sample is representative of the original dataset.

Therefore, across all 2 metrics, the sample size of 1 million rows is representative of the 6.9 million rows in the original dataset.

## Referential Integrity

The yelp\_business dataset, yelp\_reviews dataset, and yelp\_users dataset are linked by a **foreign key constraint** where `yelp_business.business_id = yelp_reviews.business_id` and `yelp_reviews.user_id = yelp_users.user_id`. Based on these foreign key constraints, we fully considered referential integrity in the sampling process.

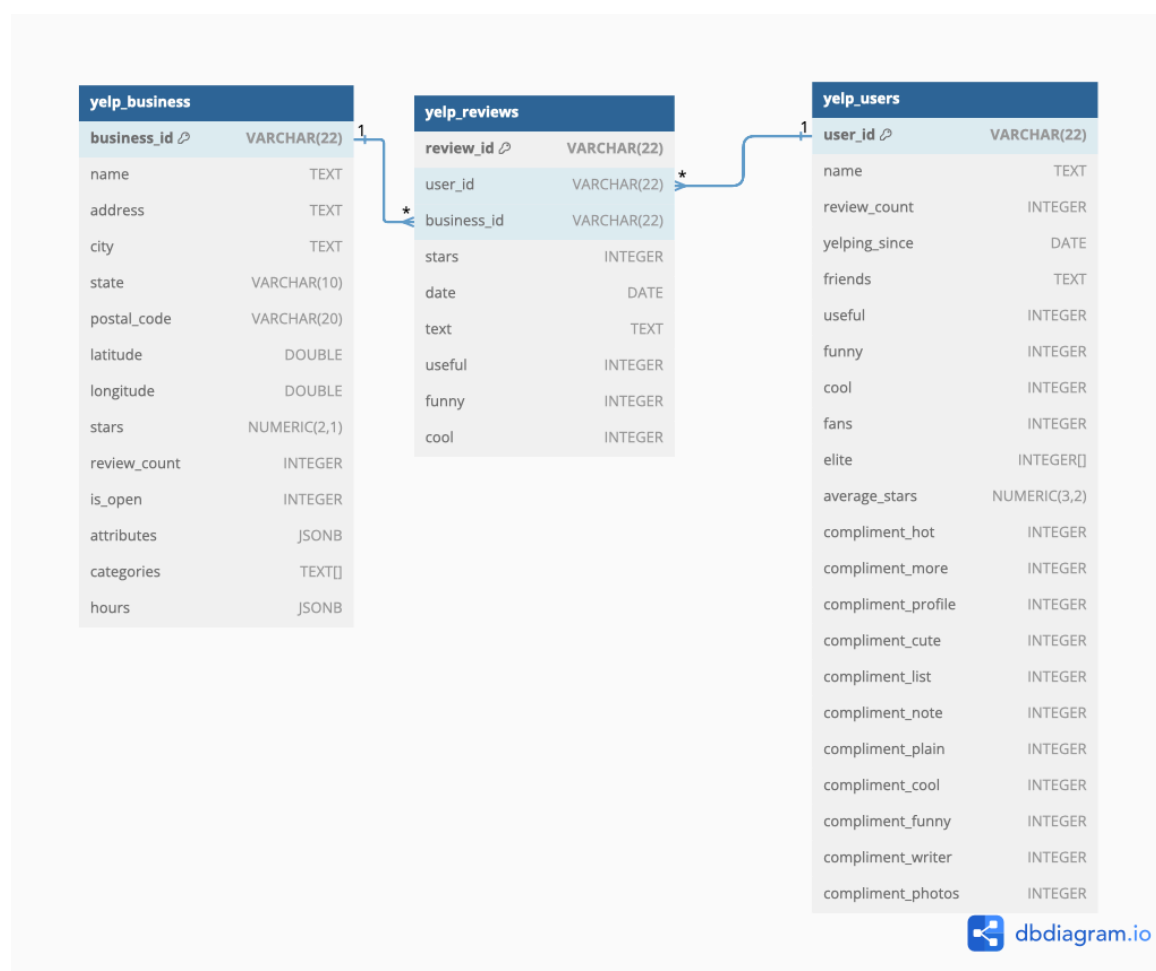
- The full users.json dataset was retained to ensure that user IDs referenced in the sampled reviews had corresponding records in the users table.
- Similarly, because the business.json dataset is relatively small, we processed it in full to ensure all business IDs in the sampled reviews were represented.

By maintaining these datasets without sampling, we avoided issues with orphaned or incomplete records, enabling accurate analyses across the datasets.

## Database Structure

We used **PostgreSQL** to store and manage the sampled dataset. The database schema is relational, consisting of a table for **reviews**, **business**, and **users**. There is a foreign key constraint between reviews and business on the `business_id` column. Additionally, there is another foreign key constraint between reviews and users on the `user_id` column.

### ER Diagram



### Data Loading Process

1. **Data Extraction:** We started by downloading the Yelp dataset, which is available in JSON format.

2. **Sampling and Truncation:** We then used Python and pandas to sample and truncate the dataset. Specifically, we read the JSON files in chunks and randomly sampled rows to reduce the size of the dataset, ensuring it was manageable for processing and analysis.
3. **Data Loading into PostgreSQL:** The sampled data was then loaded into the PostgreSQL database using SQL commands. For each table, we created a `CREATE TABLE` statement with appropriate columns and data types. The data was imported from CSV files generated from pandas into the database via the `COPY` command.

## System and Database Setup

We are using our local computers for the compute resources, and are using the same setup to store the database. All queries for timing & testing were completed on the 2020 Macbook Air 13.3" with 8 GB memory, 512 GB SSD storage, Apple M1 Chip, and macOS Sequoia 15.1.

### Platform 1: PostgreSQL

The process of loading the data into PostgreSQL and setting up the database schema has been completed successfully, and a portion of the dataset is now available for querying.

We first created a connection to the postgres database, opened a cursor, and then executed the `CREATE TABLE IF NOT EXISTS` query. for the `yelp_business`, `yelp_users`, and `yelp_reviews`. This included the foreign key constraint of `yelp_reviews` on `yelp_users` on `user_id` and `yelp_businesses` on `business_id`. Then, we open the data, accessed on a json or csv file, and then import the data into the data table, ensuring that all the lines were imported correctly. We then executed the `INSERT INTO` query to insert the data into the table through a try and exception functionality, by referencing the data loaded per line into the database. Finally, we implemented testing and ensured that the data was loaded correctly through performing a quick check of `SELECT * FROM yelp_business LIMIT 5` to ensure the data is imported correctly upon visual inspection and `SELECT COUNT(*) FROM yelp_business` to ensure that the total amount of data was also imported correctly into the dataset.

### Platform 2: PySpark

We are using our local computers for the compute resources, and are using the same setup to store the database.

Loading the datasets into Spark is extremely straightforward. We installed PySpark in our terminal by using the command `pip install PySpark`. We imported `SparkSession` from `pyspark.sql` and created a `SparkSession`:

Python

```
spark = SparkSession.builder.appName("Load_tables").getOrCreate()
```

Afterwards, we loaded our csv datasets as dataframes by using `spark.read.csv()`

Python

```
business_df = spark.read.csv(business_file_path, header=True, inferSchema=True)
users_df = spark.read.csv(users_file_path, header=True, inferSchema=True)
review_sampled_df = spark.read.csv(review_file_path, header=True,
inferSchema=True)
```

## Task Selection

We created real-world user-defined tasks that can show the difference in performance between PostgreSQL and PySpark including the following:

1. **Task 1:** Top 10 High-Rated Business Cities
2. **Task 2:** The Top Business in Every Yelp Category
3. **Task 3:** Most Popular Month for Restaurant Reviews
4. **Task 4:** Users with the Most Friends
5. **Task 5:** Number of Cities Each User Reviewed
6. **Task 6:** Top Users by Average Star Rating for businesses in that city

For each task, we created a graph varying the sample size and calculating the execution time by timing the query. As a result, we were able to determine and analyze how several factors contributed to the query's performance.

## Task 1: Top 10 High-Rated Business Cities

**What task the query is trying to accomplish:** Find top 10 Cities with at least 50 businesses with greater than 4.7 stars and a review count greater than 100.

## PostgreSQL

Python

```
pd.read_sql_query(
    """
    WITH ranked_business AS (
    SELECT
```

```

        yb.business_id,
        yb.city,
        yb.review_count,
        yb.stars,
        CASE WHEN yb.stars > 4.7 AND yb.review_count > 100
        THEN 1
        ELSE 0
        END AS is_good
    FROM yelp_business yb
), concentration_by_city AS (
    SELECT
        city,
        COUNT(business_id) as num_business,
        SUM(is_good) as total_is_good
    FROM ranked_business
    GROUP BY city
    HAVING COUNT(business_id) > 50
    ORDER BY total_is_good DESC
)

SELECT * FROM concentration_by_city LIMIT 10;
"""
conn_str
)

```

## PySpark

Python

```

ranked_business = business_df.withColumn(
    "is_good",
    when((col("stars") > 4.7) & (col("review_count") > 100), 1).otherwise(0)
).select("business_id", "city", "review_count", "stars", "is_good")

concentration_by_city = (
    ranked_business.groupBy("city")
    .agg(
        count("business_id").alias("num_business"),
        sum("is_good").alias("total_is_good")
    )
    .filter(col("num_business") > 50)
    .orderBy(col("total_is_good").desc())
)

result = concentration_by_city.limit(10)

```



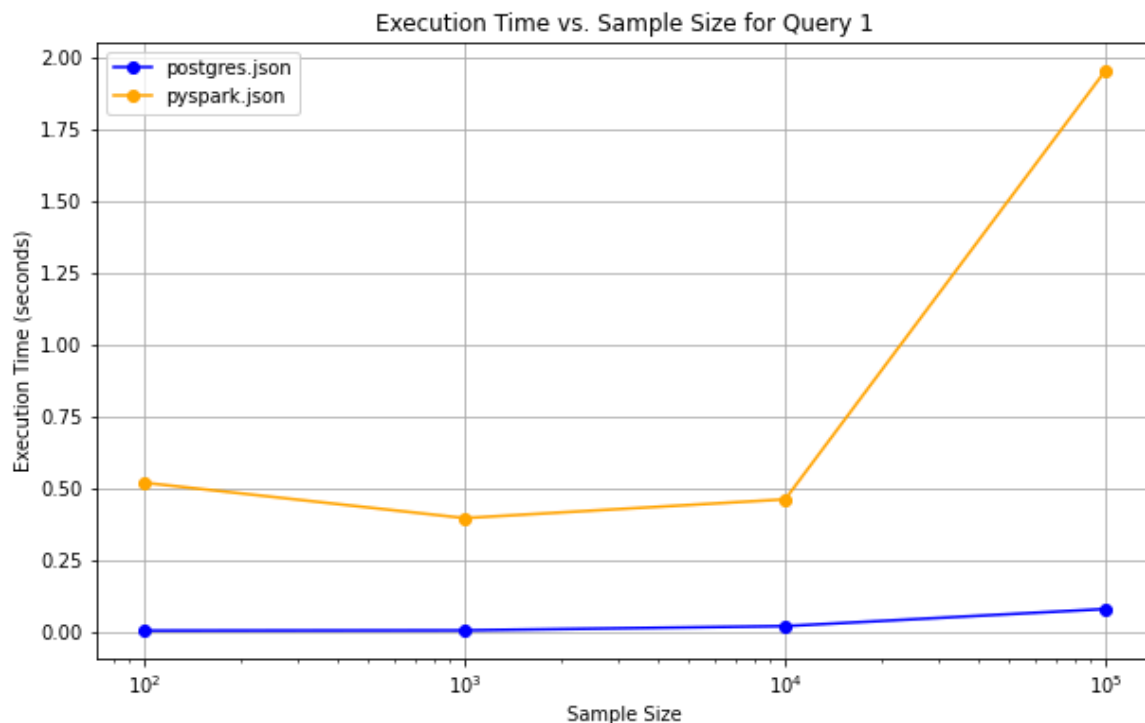
**Reasonable Solution:** This is a reasonable task to evaluate PostgreSQL because it consists of making CTEs and calculating the count and sum of aggregated groups. This query is interesting to compare to PySpark due to PySpark inability to make CTEs. Additionally, we can compare both systems' use of different sorting and aggregation algorithms.

#### Query Optimization and Implementation Plan:

Feature	PostgreSQL	PySpark
Scan Type	Parallel Sequential Scan	FileScan (CSV)
Sort	Top-N Heapsort, quicksort for worker sorting, $O(n \log n)$	Sort (similar to mergesort)
Aggregation	HashAggregate with parallel partial aggregation	HashAggregate
Parallel Workers	Two workers used for processing rows in parallel	Exchange hashpartitioning(parallel workers)
Filter	<code>HAVING COUNT(business_id) &gt; 50</code> filters city groups	Filter (num_business#5494L > 50)
Execution Time	0.0306 s for the full business dataset	1.72s for the full business dataset

#### Output:

	city	num_business	total_is_good
0	Philadelphia	14569	43
1	Tampa	9050	32
2	Santa Barbara	3829	30
3	New Orleans	6209	26
4	Reno	5935	21
5	Indianapolis	7540	19
6	Nashville	6971	18
7	Tucson	9250	9
8	Saint Louis	4827	6
9	Saint Petersburg	1663	6



#### Performance Explanation:

Both PySpark and PostgreSQL utilize similar aggregation methods like Hash Aggregate. However, PostgreSQL uses parallel sequential scan, while PySpark does not and uses Filescan. Additionally, both queries use different sorting methods with PySpark only using a distributed sorting algorithm based on merge sort (time complexity of  $O(n \log n)$ ) and PostgreSQL using heapsort which is also ( $O(n \log n)$ ) and quicksort which is best case ( $O(n \log n)$ ) and worst case ( $O(n^2)$ ). PostgreSQL is also more efficient since the HAVING filter applies directly to the grouped data while PySpark applies the filter on distributed data, since PySpark requires shuffling data across partitions for partitioning and ordering, which can be expensive. Therefore, these reasons contribute to PostgreSQL having a lower execution time.

## Task 2: The Top Business in Every Yelp Category

**What task the query is trying to accomplish:** Find the top businesses in every yelp category. Top is calculated by highest star rating and the tie is broken by highest review count. Businesses need to have at least 100 reviews to be counted.

### PostgreSQL

Unset

```
pd.read_sql_query(
    """
    WITH parsed_categories AS (
    SELECT
        yb.business_id,
        yb.city,
        yb.state,
        yb.name,
        yb.stars,
        yb.review_count,
        UNNEST(yb.categories) AS category
    FROM yelp_business yb
    ),
    num_categories AS (
        SELECT DISTINCT
            category,
            COUNT(*) AS num_category
        FROM parsed_categories
        GROUP BY category
    ),
    ranked_businesses AS (
        SELECT
            pc.category,
            pc.business_id,
            pc.name,
            pc.city,
            pc.state,
            pc.stars,
            pc.review_count,
            nc.num_category,
            RANK() OVER (
                PARTITION BY pc.category
                ORDER BY pc.stars DESC, pc.review_count DESC
            ) AS rank_in_category
        FROM parsed_categories pc
        JOIN num_categories nc ON pc.category = nc.category
    )
    SELECT
        category, num_category, business_id, name,
        city, state, stars, review_count
    FROM ranked_businesses
    WHERE rank_in_category = 1
    ORDER BY num_category DESC;

    """,
    conn_str
)
```

# PySpark

Python

```
parsed_categories_df = (
    business_df
    .withColumn("categories_array", split("categories", ", "))
.withColumn("category", explode("categories_array")).select(
    "business_id",
    "city",
    "state",
    "name",
    "stars",
    "review_count",
    "category"
)

num_categories_df = (
    parsed_categories_df
    .groupBy("category")
    .agg(count("*").alias("num_category"))
)

joined_df = (
    parsed_categories_df
    .join(num_categories_df, "category", "inner")
)

window_spec =
Window.partitionBy("category").orderBy(col("stars").desc(),
col("review_count").desc())

ranked_businesses_df = (
    joined_df
    .withColumn("rank_in_category", rank().over(window_spec))
.select(
    "category",
    "business_id",
    "name",
    "city",
    "state",
    "stars",
    "review_count",
    "num_category",
    "rank_in_category"
)

final_ranked_df = (
    ranked_businesses_df
```

```

.filter(col("rank_in_category") == 1) # Filter for rank = 1
.orderBy(col("num_category").desc()) # Order by num_category in
descending order
.select(
    "category",
    "num_category",
    "business_id",
    "name",
    "city",
    "state",
    "stars",
    "review_count"
)
)

```

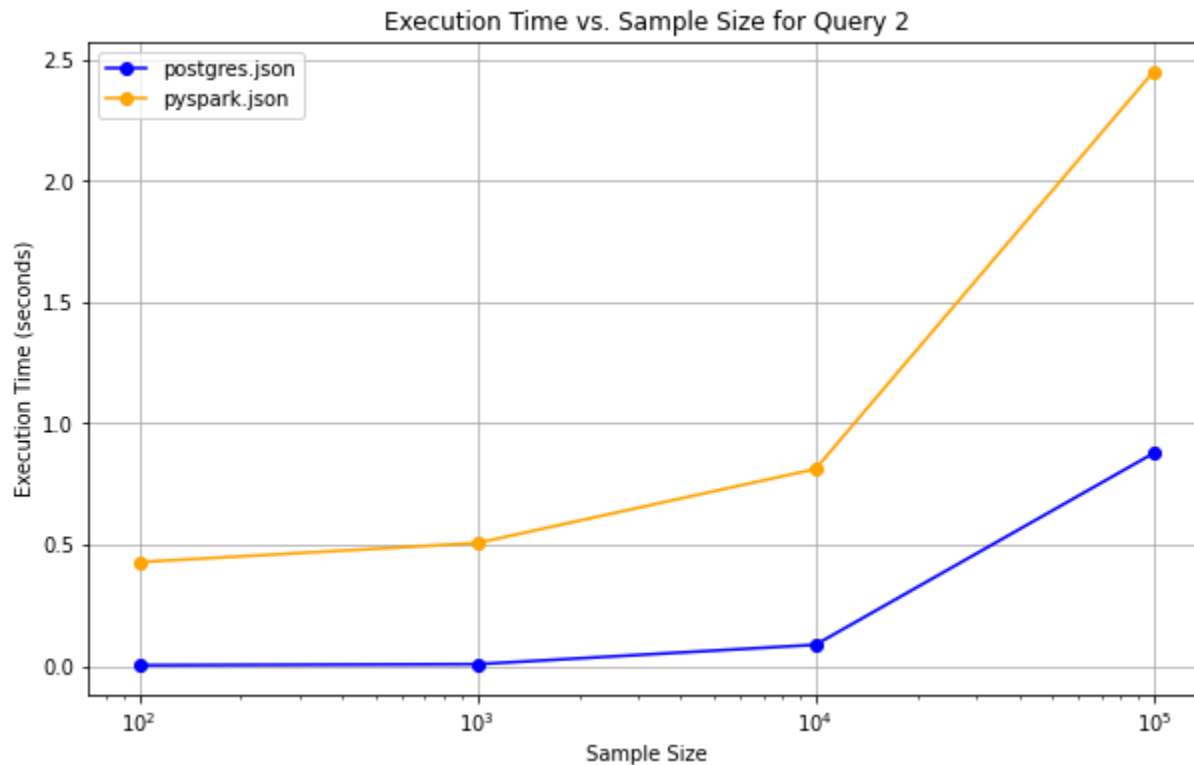
**Reasonable Solution:** This is a reasonable task to evaluate PostgreSQL because it consists of making CTEs, completing INNER Join operation and running a window function. Additionally there is no UNNEST equivalent from the PostgreSQL query to PySpark. Therefore, we used the explode function. Although similar, these functions are different as UNNEST only returns the elements while EXPLODE returns their positions in the indexed column. Additionally, it would be interesting to see the different algorithms used for join and window function.

Feature	PostgreSQL	PySpark
<b>Scan Type</b>	CTE Scan, Subquery scan with unique and sort, Sequential Scan,	File Scan
<b>Sort</b>	Quicksort, SortKey	Sort (similar to mergesort)
<b>Aggregation</b>	Window Aggregation, unique , hash join	HashAggregate, Window Aggregation, SortMergeJoin
<b>Parallel Workers</b>	Parallel Seq Scan	exchange hash partitioning
<b>Filter</b>	Filter rank_category = 1	Filter ((size(split(categories#3992, , -1), true) > 0) AND isnotnull(split(categories#3992, , -1)))

<b>Execution Time for full dataset</b>	2.69 seconds	4.22 seconds
--	--------------	--------------

**Output:**

	<b>catego ry</b>	<b>num_cate gory</b>	<b>business _id</b>	<b>name</b>	<b>city</b>	<b>stat e</b>	<b>star s</b>	<b>review_co unt</b>
0	Restaur ants	52268	_aKr7PO nacW_Viz RKBpCiA	Blues City Deli	Saint Louis	MO	5.0	991
1	Food	27781	JbzvJJolD BT1614q o2Yiaw	Nelson's Green Brier Distillery	Nashville	TN	5.0	545
2	Shoppi ng	24395	l_7TW_lx5 8-QvhQg pJi_Xw	SUGARED + BRONZED	Philadelp hia	PA	5.0	513
3	Home Service s	14356	u3zqvp4 BYUjzJD7 txx3Jbg	JB Plumbing and Heating	Philadelp hia	PA	5.0	398
4	Beauty & Spas	14292	l_7TW_lx5 8-QvhQg pJi_Xw	SUGARED + BRONZED	Philadelp hia	PA	5.0	513
5	Nightlif e	12281	_aKr7PO nacW_Viz RKBpCiA	Blues City Deli	Saint Louis	MO	5.0	991
6	Health & Medical	11890	hxZ-5CYu CbX_eb2v Zbx20A	Royal Thai Massage	Santa Barbara	CA	5.0	241
7	Local Service s	11198	eJUjeWnd PCCqTjj0J va-ug	Sterling Carpet Care	Reno	NV	5.0	276
8	Bars	11065	_aKr7PO nacW_Viz RKBpCiA	Blues City Deli	Saint Louis	MO	5.0	991
9	Automo tive	10773	eJUjeWnd PCCqTjj0J va-ug	Sterling Carpet Care	Reno	NV	5.0	276



### Performance explanation:

This query includes CTE Scan, subquery scan, sequential scan, sorts, window aggregation, and more. PostgreSQL's UNNEST directly returns elements without including positional metadata, making it leaner for this operation. PySpark EXplode includes metadata, which, while useful in other contexts, adds additional processing overhead. PostgreSQL efficiently uses hash joins and parallel sequential scans. Its window aggregation is also optimized for single-node execution. PySpark handles window operations across distributed partitioning, potentially adding overhead. PostgreSQL leverages parallel sequential scans which is faster for operations on structured datasets that fit in memory, while PySpark uses exchange hash partitioning, which involves network communication and synchronization across nodes. For sorting, while PySpark's MergeSort has worst case runtime  $O(n \log n)$ , it is often slower than quicksort due to creating additional arrays. These reasons illustrate how PostgreSQL is optimized for this type of query.

## Task 3: Most Popular Month for Restaurant Reviews

**What task the query is trying to accomplish:** Find the month that users review restaurants the most.

We initially wanted to find the month that people eat at restaurants the most. However, we could not achieve that since we only have the date the review was made and not the date the user went to the restaurant.

## PostgreSQL Query

```
Python
d.read_sql_query(
    """
    SELECT
        TO_CHAR(ysr.date, 'Month') AS review_month,
        AVG(yb.stars) AS avg_stars,
        AVG(yb.review_count) AS avg_review_count,
        COUNT(ysr.review_id) AS total_reviews
    FROM yelp_business yb
    JOIN yelp_small_reviews ysr ON yb.business_id = ysr.business_id
    WHERE 'Restaurants' = ANY(yb.categories)
    GROUP BY review_month
    ORDER BY avg_review_count DESC LIMIT 1
    """,
    conn_str
)
```

**Reasonable Solution:** This is a reasonable solution to test our database system because it involves a join on a foreign key. Additionally, the WHERE clause can influence performance by the presence or lack of an indexed column on the column called in the WHERE clause. It will also test sorting and aggregation of the system by GROUP BY, ORDER BY, and WHERE. Furthermore, the query includes string manipulation through the TO\_CHAR function which adds another computational layer teaching the database's handling of date-time operations.

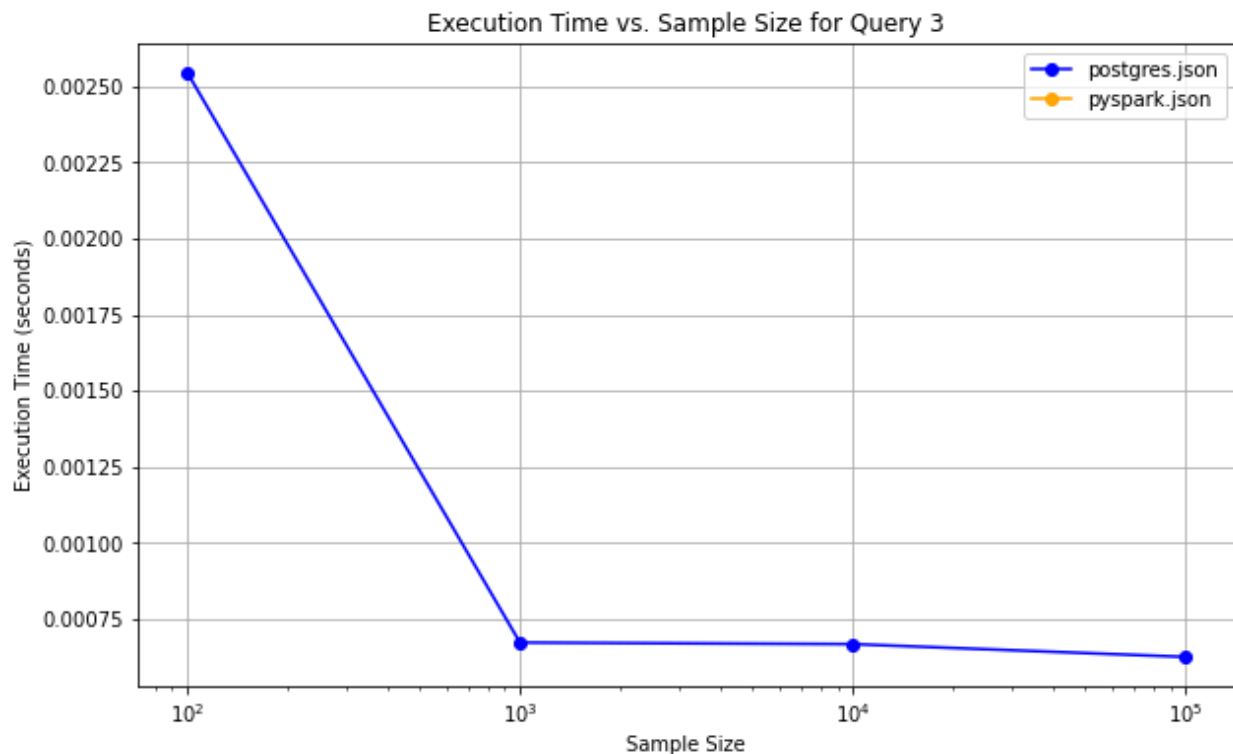
Feature	PostgreSQL
Scan Type	Seq Scan
Sort	Sort Method: external merge, Sort Key, quicksort
Aggregation	HashAggregate, Hash Join
Parallel Workers	Parallel Seq Scan
Filter	Filter: (categories ~~ '%Restaurants% '::text)



<b>Execution Time</b>	1116.480 ms
-----------------------	-------------

### Output:

review_month	avg_stars	avg_review_count	total_reviews
June	3.786144	455.934564	59325



### Performance:

The execution time remains relatively constant for dataset sizes beyond a certain threshold due to indexed operations, efficient hash joins, and aggregation algorithms. PostgreSQL optimizes joins and groupings by scanning only the relevant rows and leveraging efficient in-memory operations when possible. Additionally, the query joins the `yelp_business` and `yelp_small_reviews` tables using the `business_id` foreign key. If `business_id` is indexed (common in such schemas), the join operation avoids a full table scan and directly retrieves rows matching the key. PostgreSQL uses a hash join for such scenarios, which is highly efficient for equality joins. Thus, despite an exponential increase in sample size, the execution time remains relatively similar.

## Task 4: Users with the Most Friends

**What task the query is trying to accomplish:** Find the top 10 users with the most amount of friends.

### PostgreSQL

Python

```
SELECT user_id, name, array_length(string_to_array(friends, ','), 1) as  
number_of_friends  
FROM yelp_users  
ORDER BY number_of_friends DESC  
LIMIT 10;
```

### PySpark

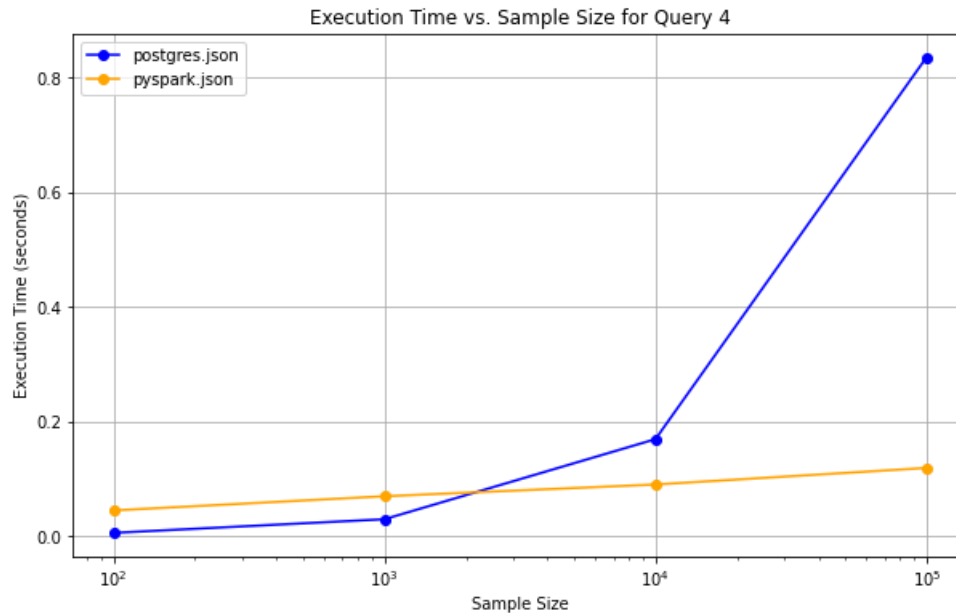
Python

```
most_friends = users_df.withColumn(  
    "number_of_friends", size(split(col("friends"), ",")))  
.select("user_id",  
    "name", "number_of_friends").orderBy(col("number_of_friends").desc())  
most_friends.explain()
```

**Reasonable Solution:** This is a reasonable task as the friends attribute is an array. This query tests the optimization of string splitting and array size calculations, which can be computationally expensive especially with large models. Therefore, this query provides the opportunity to assess how PySpark manages these operations and how its scalability impacts overall performance.

Feature	PostgreSQL	PySpark
Scan Type	Parallel Seq Scan	Filescan
Sort	Sort Method: external merge	Sort (similar to mergesort)
Aggregation		
Parallel Workers	Parallel Seq Scan	
Filter		

<b>Execution Time</b>	0.145 s	0.16s
-----------------------	---------	-------



	<b>user_id</b>	<b>name</b>	<b>number_of_friends</b>
0	qVc8ODYU5SZjKXVBgXdl7w	Walker	14995
1	iLjMdZi0Tm7DQxX1C1_2dg	Rugby	12395
2	ZIOCmdFaMIF56FR-nWr_2A	Randy	11026
3	mV4lknblF-zOKSF8nIGqDA	Scott	10366
4	Oi1qbcz2m2SnwUeztGYcnQ	Steven	10072

### Performance Explanation:

PySpark distributes data across multiple nodes in order to process large scale data. Therefore PySpark spends longer in computation due to these overhead tasks of distributing data to multiple nodes. PostgreSQL on the other hand is more efficient in smaller database sets due to less work due to overhead tasks. However, as sample size

increases, PySpark leverages distributed computing to partition and process data across multiple nodes, leading to a lower execution time than PostgreSQL.

## Task 5: Number of Cities Each User Reviewed

**What task the query is trying to accomplish:** Add a new attribute to the users database that is the total number of unique cities they reviewed.

**Reasonable Solution:** This is a reasonable task to test our system because it joins two datasets. Also due to name constraints, it requires to rename the column names that are identical to column names in the other dataset. This is a PySpark specific issue as it is allowed to join on identical column names in PostgreSQL. Therefore this can affect performance of the task.

### PySpark Query

Python

```
business_df = spark.read.csv(business_file_path, header=True, inferSchema=True)
business_df = business_df.select(
    *[F.col(c).alias(f"businesses_{c}") for c in business_df.columns])

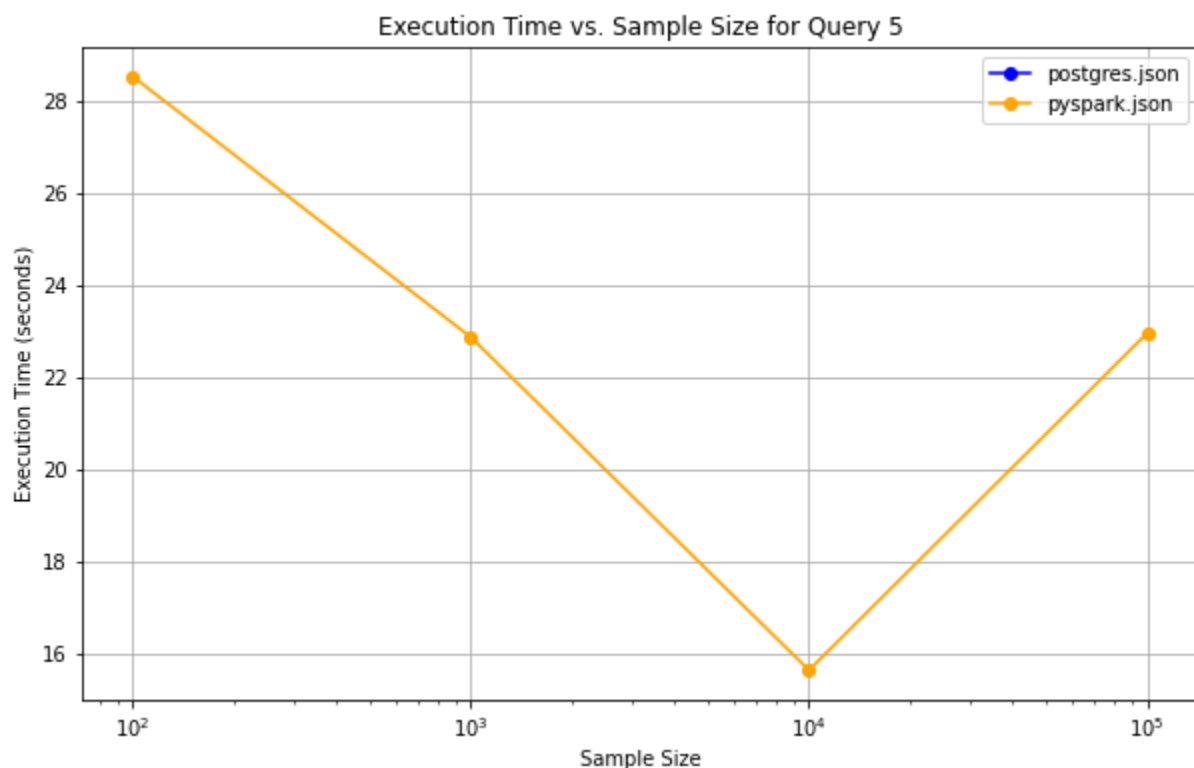
user_business = (
    review_sampled_df
    .join(
        business_df,
        review_sampled_df.business_id == business_df.businesses_business_id,
        "inner"
    )

    from pyspark.sql.functions import concat, lit
    user_business = user_business.withColumn(
        "city_state",
        concat(user_business.businesses_city, lit(", "),
        user_business.businesses_state)
    )

    result_df = (
        user_business
        .groupBy("user_id")
        .agg(
            count("*").alias("unique_cities")
        ).filter(col("unique_cities") > 5)).sort(desc("unique_cities"))
```

Feature	PySpark
---------	---------

<b>Scan Type</b>	Filescan,
<b>Sort</b>	- Sort
<b>Aggregation</b>	HashAggregate, SortMergeJoin
<b>Parallel Workers</b>	Exchange hash partitioning
<b>Filter</b>	Filter isnotnull(business_id#8402)
<b>Execution Time</b>	



### Performance Explanation:

This query illustrates the high variability of data for smaller data sizes, as spark is more optimized for higher data sizes. Spark's performance is highly influenced by data size. Smaller datasets can amplify the impact of Spark's overhead, leading to less optimal performance. In this specific query, the relatively small dataset size results in a significant portion of time spent on initialization and overhead tasks rather than actual data processing. Furthermore, this was the most computationally expensive query and took up a significant amount of time and space.

## Task 6: Top Users by Average Star Rating for businesses in that city

**What task the query is trying to accomplish:** Find the top users by average star rating for businesses in Philadelphia.

```
Python
pd.read_sql_query(
    """
    SELECT yu.user_id, yu.name, AVG(yr.stars) AS avg_stars, yu.review_count
    FROM yelp_business yb
    JOIN yelp_small_reviews yr ON yb.business_id = yr.business_id
    JOIN yelp_users yu ON yu.user_id = yr.user_id
    WHERE yb.city = 'Philadelphia'
    GROUP BY yu.user_id, yu.name, yu.review_count
    ORDER BY avg_stars DESC, yu.review_count DESC
    LIMIT 10;
    """,
    conn
```

**Reasonable Solution:** This is a reasonable task to test our system because there are two database joins. One join is on a foreign key (business\_id) and one join is on a primary key for the user database. Additionally, we can compare the performance based on indexing of yb.city as it is accessed in the WHERE clause.

Feature	PostgreSQL no city index	PostgreSQL index
Scan Type	Parallel Seq Scan, Index Scan	Index Scan, Parallel Seq Scan
Sort	top-N heapsort, quicksort	top-N heapsort, quicksort
Aggregation	SortMergeJoin, Gather Merge, Parallel Hash Join	Gather Merge , Parallel Hash Join
Parallel Workers	Workers Planned: 2	Workers Planned: 2
Filter	N/A	N/A
Execution Time	1.912 seconds	1.28 seconds

This illustrates a 1.5x increase in speed from having no index to having an index.

**Output:**

user_id	name	avg_stars	review_count
m07sy7eLtOjVdZ8o N9JKag	Ed	5.0	5800
syvwUKQJ4OYfmL_ix VLMeQ	Tina	5.0	4386
whlNg-cC-FiAv_ATD GMDTg	PatrickJ	5.0	3775
uIBhZTxEdOw1FRVp SJ45mQ	Franklin	5.0	3683
gcMPEkEXekKN6mY hLUI-Rw	John	5.0	3148
zUKeZNu4tCG56xjw WH54Vw	Bill	5.0	2877
B7ecAeAlrXg7sgma bS38pg	Stephy	5.0	2605
kKTcYPz47sCDH1_yl nE4ZQ	Pegah	5.0	2595
Xxvz5g67eaCr3emn kY5M6w	Jen	5.0	2403
cjCYCEfDL4n5TgU9y 8RTWg	Allister	5.0	2100

**Performance:**

The execution time of the query including an index on yb.city is significantly faster because PostgreSQL index allows the system to locate rows that are relevant quicker and it does not require scanning the whole dataset. Since this index is used in the WHERE clause, it cuts down execution time.

# Tool Comparison

We used PostgreSQL and PySpark for this project. We had familiarity with psql due to previous assignments but we were not familiar with PySpark. PostgreSQL is an open source relational database management system and supports both SQL (relational) and JSON (non relational) in addition to offering advanced SQL functions. On the other hand, PySpark is the Python API for Apache Spark. PySpark works as a distributed, large-scale data processing environment in Python.

## Installation

The installation process varied for both tools. PostgreSQL required to configure database username, password, host port and to specify the name of the database. However, PySpark requires only building a spark session and it is not stored locally on the laptop. A spark session allows us to interact with Spark and where we can access Spark functionality. Overall, Spark was easier to set up and use due to its similarity to pandas, a tool we are familiar with.

## Data Handling

PostgreSQL reads from relational tables, while PySpark processes data in-memory from CSV files.

## Query Nuances

PostgreSQL was easier to create CTE expressions that temporarily saved those within the query and used for the final query. It is better suited for quick, exploratory queries on relational tables for moderately-sized datasets. It is also optimized for subqueries, window functions, and relational joins (due to the PARTITION BY, RANK(), and other functions). PySpark, on the other hand, requires creating separate dataframes for each CTE table and joining them together to create the final dataframe result. PySpark also offers intuitive dataframe operations such as adding or dropping columns and handling null values with .withColumn() and .na(). It also includes features to improve performance for repeated operations. Furthermore, the sequence of operations is more intuitive because the user defines the order of every group by, select, or aggregation clause.

## Query Optimization and Performance

PostgreSQL performs better with smaller datasets due to its lightweight execution model, efficient query plans, and lower overhead. PySpark becomes more efficient as data size increases, leveraging distributed computing to partition and process data across multiple nodes. It is optimized for tasks like sorting, aggregation, and ETL processes at scale. For aggregations, partitions, and scanning, PostgreSQL uses a row-based, parallel execution model while PySpark relies on shuffling across nodes. PostgreSQL uses parallel workers and multistep aggregation while PySpark distributes aggregation across partitions.

An important observation is that PySpark was stagnant in the algorithms that it used when sorting and aggregating. For the tasks that we ran in PySpark, we noticed that it only used Hash Aggregation and a PySpark sort operation (similar to merge sort). This was interesting as PostgreSQL had more variances in the optimizations for scanning,



aggregation, using parallelism, and more. This further illustrates the difference between the two systems.

## Final Reflection

PostgreSQL excels in performing structured queries on well-defined datasets. The use of SQL makes it accessible for users with a basic understanding of relational databases. Commands like `\d` for listing data table schemas and `\t` for toggling table views made exploring and interacting with the database efficient and user-friendly. PostgreSQL handles complex queries with CTEs, window functions, and joins efficiently. Its execution plans helped provide helpful performance insights on the types of scans, aggregations, and calculating the overall execution time effectively. PostgreSQL offers many functions that allow for concise and readable query designs. PostgreSQL is challenging for extremely large datasets. Thus, we would use PostgreSQL for structured data that fits comfortably within a single database instance. It is especially effective for workloads requiring transactional integrity and moderate data volumes. Foreign key and primary key contracts aid in helping ACID compliance for testing changes without affecting the database permanently. What didn't work well in PostgreSQL was configuration of data tables across our devices due to load table and delete table durations, which was difficult and cumbersome.

PySpark excels in handling massive datasets that require distributed computing. The functional programming style with RDDs/DataFrames allows for flexible data manipulation, and its distributed architecture is better for scaling. PySpark was easy to create through initializing the session and reading csv files to directly create spark dataframes. For smaller datasets or simpler tasks, PySpark's overhead makes it slower and harder to justify. Overall, we would recommend PySpark for someone who is focused on analyzing large datasets with large sizes from TB to PB in size, through streams and parallel computation, and data that cannot be stored locally on memory due to the size. It is highly optimized for the ETL pipeline. What worked well was the ease of use of initializing a PySpark session and the intuition of creating & viewing dataframes.

Lastly, we understand that PostgreSQL and PySpark are tools which should be used concurrently due to their respective strengths with PostgreSQL being a relational database tool while PySpark is used to transform extremely large quantities of data at incredible speeds. Overall, PostgreSQL and PySpark are great platforms to work with for data processing and analysis.

# Individual Reflections

## **Bianca**

I learned how to use my terminal to load a postgresQL database. I also learned more about PySpark. I had limited exposure to PySpark during my past during my internship but this taught me how to access columns and compute queries. The most exciting part was looking at PySpark documentation. However, the most difficult part was interpreting the findings as it would occasionally be inconsistent with my prediction and we had to figure out why.

## **Manan:**

I learned about the process of loading a postgres database, and being able to link data into those databases. I also learned significantly about the process of sampling, chunking data, and creating metrics to develop a representative sample of the data. I also had learned to develop metrics to understand if the sample was representative of the entire dataset, as well as determine how to compare the different systems like Postgres and Apache Spark across their execution times, query execution plans, and more. This was very fascinating for showing us the ability, power, and speed of the different types of databases and understanding the different use cases on the different databases depending on which application, the data set size., etc., we are using this for. Furthermore, I learned deeply about syntax for various queries, setting up the jupyter notebook pipelines, and leveraging the terminal for setting up the databases.

## **Nawoda:**

Setting up a local PostgreSQL Database and understanding how psycopg implements transactions and loading the data in a way that doesn't break the ACID properties were some of the biggest hurdles initially. It definitely made me appreciate ACID and their importance since errors would happen consistently so instead of deleting the entire dataset, the INSERTED data would not be committed instead. Shaping the data from a JSON format to a relational database has its challenges mainly due to the nested attributes and figuring out a way to change them to TEXT [] or if it's just better to keep it as JSON. Depending on what we wanted to query I cleaned the data to better understand them. After the data has been all loaded in a format we could easily use without REGEX to extract the information, I made the queries and ran EXPLAIN ANALYZE on them without any trouble. Overall, the initial work of creating the database was the most infuriating, especially with setting up the libraries and creating a conda environment specifically for PostgreSQL but I think it has been what I wanted to get the most out of the project and am pretty satisfied with my work in doing so.

# References

## Documentation

<https://spark.apache.org/docs/latest/api/python/index.html>

<https://www.postgresql.org/docs/>

<https://www.psycopg.org/docs/>

<https://docs.python.org/3/library/time.html>

<https://docs.python.org/3/library/timeit.html>

## Time and Space Complexity

<https://stackoverflow.com/questions/55113713/time-space-complexity-of-in-built-python-functions>

<https://stackshare.io/stackups/postgresql-vs-spark>

<https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/>

<https://www.geeksforgeeks.org/heap-sort/>

<https://stackoverflow.com/questions/32887595/how-does-spark-achieve-sort-order>

## Sampling Methods

<https://www.scribbr.com/methodology/sampling-methods/>

[Lec 20 - DATA 101 Fa24 - MapReduce, Sampling](#)

## Performance Metrics

<https://www.kualitatem.com/blog/software-testing/how-to-measure-database-performance-and-optimize-queries-during-performance-testing/>

<https://severalnines.com/blog/how-measure-database-performance/>

<https://www.quora.com/How-is-the-performance-of-a-database-system-measured>

# Appendix

## Task 1 Plan: Top 10 Highest-Rated Business Cities PostgreSQL

```
Unset
QUERY PLAN
0                                     Limit
(cost=16461.21..16461.34 rows=10 width=26) (actual time=88.813..92.894 rows=10
loops=1)
1                                     -> Sort
(cost=16461.21..16461.88 rows=268 width=26) (actual time=88.812..92.892 rows=10
loops=1)
2                                     Sort
Key: (sum(CASE WHEN ((yb.stars > 4.7) AND (yb.review_count > 100)) THEN 1 ELSE
0 END)) DESC
3
Sort Method: top-N heapsort  Memory: 25kB
4                                     -> Finalize GroupAggregate
(cost=16240.42..16450.40 rows=268 width=26) (actual time=87.186..92.821
rows=277 loops=1)
5
Group Key: yb.city
6
Filter: (count(yb.business_id) > 50)
7
Rows Removed by Filter: 1139
8                                     -> Gather Merge
(cost=16240.42..16428.26 rows=1610 width=26) (actual time=87.173..92.123
rows=2779 loops=1)
9
Workers Planned: 2
10
Workers Launched: 2
11                                     -> Sort
(cost=15240.39..15242.41 rows=805 width=26) (actual time=80.787..80.879
rows=926 loops=3)
12
Sort Key: yb.city
13
Sort Method: quicksort  Memory: 129kB
14
Worker 0:  Sort Method: quicksort  Memory: 92kB
15
Worker 1:  Sort Method: quicksort  Memory: 93kB
16                                     -> Partial HashAggregate
(cost=15193.49..15201.54 rows=805 width=26) (actual time=79.892..80.170
rows=926 loops=3)
17
Group Key: yb.city
18
Batches: 1  Memory Usage: 193kB
```

```

19
Worker 0:  Batches: 1  Memory Usage: 169kB
20
Worker 1:  Batches: 1  Memory Usage: 169kB
21
                -> Parallel Seq Scan on yelp_business yb
(cost=0.00..14410.44 rows=62644 width=42) (actual time=0.009..21.181 rows=50115
loops=3)
22
Planning Time: 0.605 ms
23
Execution Time: 93.006 ms

```

## PySpark

```

Unset
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- TakeOrderedAndProject(limit=10, orderBy=[total_is_good#40245L DESC NULLS
LAST], output=[city#39806,num_business#40243L,total_is_good#40245L])
    +- Filter (num_business#40243L > 50)
        +- HashAggregate(keys=[city#39806], functions=[count(business_id#39803),
sum(is_good#40216)])
            +- Exchange hashpartitioning(city#39806, 200), ENSURE_REQUIREMENTS,
[plan_id=58268]
                +- HashAggregate(keys=[city#39806],
functions=[partial_count(business_id#39803), partial_sum(is_good#40216)])
                    +- Project [business_id#39803, city#39806, CASE WHEN
((stars#39811 > 4.7) AND (review_count#39812 > 100.0)) THEN 1 ELSE 0 END AS
is_good#40216]
                        +- FileScan csv
[business_id#39803,city#39806,stars#39811,review_count#39812] Batched: false,
DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1
paths)[file:/Users/mananbhargava/Documents/Workspaces/data101/Data101-Final-P..
., PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<business_id:string,city:string,stars:double,review_count:double>

```

## Task 2 Plan: The Top Business in Every Yelp Category

### PostgreSQL

```

Unset
QUERY PLAN

```

```

0
Limit (cost=621276.35..621276.38 rows=10 width=220) (actual
time=7670.246..7670.257 rows=10 loops=1)
1
CTE parsed_categories
2
-> ProjectSet (cost=0.00..23932.35 rows=1503460 width=97) (actual
time=0.011..285.241 rows=668592 loops=1)
3
Scan on yelp_business yb (cost=0.00..15287.46 rows=150346 width=162) (actual
time=0.008..29.394 rows=150346 loops=1)
4
-> Sort (cost=597344.00..597362.79 rows=7517 width=220) (actual
time=7670.245..7670.253 rows=10 loops=1)
5
Sort Key: ranked_businesses.num_category DESC
6
Sort Method: top-N heapsort Memory: 27kB
7
-> Subquery Scan on
ranked_businesses (cost=544560.46..597181.56 rows=7517 width=220) (actual
time=4429.681..7652.152 rows=1355 loops=1)
8
Filter: (ranked_businesses.rank_in_category = 1)
9
Rows Removed by Filter: 667237
10
WindowAgg (cost=544560.46..578388.31 rows=1503460 width=228) (actual
time=4429.679..7468.685 rows=668592 loops=1)
11
-> Sort (cost=544560.46..548319.11 rows=1503460 width=220) (actual
time=4429.665..5928.725 rows=668592 loops=1)
12
Sort Key: pc.category, pc.stars DESC, pc.review_count DESC
13
Sort Method: external merge Disk: 66880kB
14
-> Hash Join (cost=37602.14..71701.55 rows=1503460 width=220) (actual
time=1474.895..2051.867 rows=668592 loops=1)
15
Hash Cond: (pc.category = nc.category)
16
-> CTE Scan on
parsed_categories pc (cost=0.00..30069.20 rows=1503460 width=212) (actual
time=0.013..222.408 rows=668592 loops=1)
17
-> Hash (cost=37599.64..37599.64 rows=200 width=40) (actual
time=1474.868..1474.872 rows=1311 loops=1)
18
Buckets: 2048 (originally 1024) Batches: 1 (originally 1) Memory Usage: 86kB
19
-> Subquery Scan on nc (cost=37596.14..37599.64 rows=200 width=40) (actual
time=1473.989..1474.560 rows=1311 loops=1)
20
-> Unique (cost=37596.14..37597.64 rows=200 width=40) (actual
time=1473.987..1474.359 rows=1311 loops=1)

```

```

21
-> Sort (cost=37596.14..37596.64 rows=200 width=40) (actual
time=1473.986..1474.105 rows=1311 loops=1)
22
Sort Key: parsed_categories.category, (count(*))
23
Sort Method: quicksort  Memory: 146kB
...
26
Batches: 1  Memory Usage: 209kB
27
on parsed_categories (cost=0.00..30069.20 rows=1503460 width=32) (actual
time=0.000..1138.682 rows=668592 loops=1)
28
Planning Time: 0.213 ms
29
Execution Time: 7718.755 ms

```

## PySpark

```

Unset
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [category#40270, num_category#40302L, business_id#39803, name#39804,
city#39806, state#39807, stars#39811, review_count#39812]
  +- Sort [num_category#40302L DESC NULLS LAST], true, 0
    +- Exchange rangepartitioning(num_category#40302L DESC NULLS LAST, 200),
ENSURE_REQUIREMENTS, [plan_id=58755]
      +- Project [category#40270, business_id#39803, name#39804, city#39806,
state#39807, stars#39811, review_count#39812, num_category#40302L]
        +- Filter (rank_in_category#40333 = 1)
          +- Window [rank(stars#39811, review_count#39812)
windowSpecification(category#40270, stars#39811 DESC NULLS LAST,
review_count#39812 DESC NULLS LAST, specifiedwindowframe(RowFrame,
unboundedpreceding$(), currentrow$())) AS rank_in_category#40333],
[category#40270], [stars#39811 DESC NULLS LAST, review_count#39812 DESC NULLS
LAST]
            +- WindowGroupLimit [category#40270], [stars#39811 DESC NULLS
LAST, review_count#39812 DESC NULLS LAST], rank(stars#39811,
review_count#39812), 1, Final
              +- Sort [category#40270 ASC NULLS FIRST, stars#39811 DESC
NULLS LAST, review_count#39812 DESC NULLS LAST], false, 0
                +- Project [category#40270, business_id#39803,
city#39806, state#39807, name#39804, stars#39811, review_count#39812,
num_category#40302L]
                  +- SortMergeJoin [category#40270], [category#40319],
Inner
                    :- Sort [category#40270 ASC NULLS FIRST], false,
0
                      : +- Exchange hashpartitioning(category#40270,
200), ENSURE_REQUIREMENTS, [plan_id=58735]
                        ...

```

```

+- Filter
((size(split(categories#40317, , , -1), true) > 0) AND
isnotnull(split(categories#40317, , , -1)))
+- FileScan csv
[categories#40317] Batched: false, DataFilters: [(size(split(categories#40317,
, , -1), true) > 0), isnotnull(split(categories#40317, , , -1))], Format: CSV,
Location: InMemoryFileIndex(1
paths)[file:/Users/mananbhargava/Documents/Workspaces/data101/Data101-Final-P..
., PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<categories:string>

```

## Task 3 Query Plan: Most Popular Month for Restaurant Reviews

### PostgreSQL

```

Unset
QUERY PLAN0Sort (cost=216784.81..217657.31 rows=348998 width=104) (actual
time=2310.921..2310.923 rows=12 loops=1)1Sort Key: (avg(yb.review_count))
DESC2Sort Method: quicksort Memory: 25kB3-> HashAggregate
(cost=151960.09..165565.56 rows=348998 width=104) (actual
time=2310.874..2310.910 rows=12 loops=1)4Group Key: to_char(ysr.date,
'Month'::text)5Planned Partitions: 64 Batches: 1 Memory Usage: 217kB6-> Hash
Join (cost=17822.67..111498.13 rows=348998 width=64) (actual
time=269.932..1956.648 rows=689501 loops=1)7Hash Cond: ((ysr.business_id)::text
= (yb.business_id)::text)8-> Seq Scan on yelp_small_reviews ysr
(cost=0.00..90177.91 rows=999991 width=54) (actual time=1.490..583.697
rows=1000000 loops=1)9-> Hash (cost=17166.78..17166.78 rows=52471 width=32)
(actual time=268.406..268.407 rows=52268 loops=1)10Buckets: 65536 Batches: 1
Memory Usage: 3932kB11-> Seq Scan on yelp_business yb (cost=0.00..17166.78
rows=52471 width=32) (actual time=0.013..248.180 rows=52268 loops=1)12Filter:
('Restaurants'::text = ANY (categories))13Rows Removed by Filter:
9807814Planning Time: 0.258 ms15Execution Time: 2313.016 ms

```

## Task 4 Query Plan: Users with the Most Friends

### PySpark



```
+~ Project [user_id#39848, name#39849, size(split(friends#39856, ,, -1), true)
AS number_of_friends#40409]
      +- FileScan csv [user_id#39848,name#39849,friends#39856] Batched:
false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1
paths)[file:/Users/mananbhangava/Documents/Workspaces/data101/Data101-Final-P..
., PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<user_id:string,name:string,friends:string>
```

## PySpark

```

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [unique_cities#24650L DESC NULLS LAST], true, 0
    +- Exchange rangepartitioning(unique_cities#24650L DESC NULLS LAST, 200), ENSURE_REQUIREMENTS, [plan_id=41638]
        +- Filter (unique_cities#24650L > 5)
            +- HashAggregate(keys=[user_id#13609], functions=[count(1)])
                +- Exchange hashpartitioning(user_id#13609, 200), ENSURE_REQUIREMENTS, [plan_id=41634]
                    +- HashAggregate(keys=[user_id#13609], functions=[partial_count(1)])
                        +- Project [user_id#13609]
                            +- SortMergeJoin [business_id#13610], [businesses_business_id#24525], Inner
                                :- Sort [business_id#13610 ASC NULLS FIRST], false, 0
                                    : +- Exchange hashpartitioning(business_id#13610, 200), ENSURE_REQUIREMENTS, [plan_id=41626]
                                        :     +- Filter isnotequal(business_id#13610)
                                            :         +- FileScan csv
[user_id#13609,business_id#13610] Batched: false, DataFilters:
[isnotequal(business_id#13610)], Format: CSV, Location: InMemoryFileIndex(1 paths)[file:/Users/mananbhargava/Documents/Workspaces/data101/Data101-Final-P.., PartitionFilters: [], PushedFilters: [IsNotNull(business_id)], ReadSchema: struct<user_id:string,business_id:string>
    +- Sort [businesses_business_id#24525 ASC NULLS FIRST], false, 0
        +- Exchange
hashpartitioning(businesses_business_id#24525, 200), ENSURE_REQUIREMENTS, [plan_id=41627]
    +- Project [business_id#24497 AS businesses_business_id#24525]
        +- Filter isnotequal(business_id#24497)

```

```

+- FileScan csv [business_id#24497]
Batched: false, DataFilters: [isnotnull(business_id#24497)], Format: CSV,
Location: InMemoryFileIndex(1
paths)[file:/Users/mananbhargava/Documents/Workspaces/data101/Data101-Final-P..
., PartitionFilters: [], PushedFilters: [IsNotNull(business_id)], ReadSchema:
struct<business_id:string>

```

## Task 6 Query Plan

### PostgreSQL

```

Unset
QUERY PLAN
0
Limit  (cost=152590.35..152590.38 rows=10 width=65) (actual
time=2172.680..2174.115 rows=10 loops=1)
1
-> Sort  (cost=152590.35..152831.75 rows=96557 width=65) (actual
time=2172.679..2174.114 rows=10 loops=1)
2
Sort Key: (avg(yr.stars)) DESC, yu.review_count DESC
3
Sort Method: top-N heapsort  Memory: 26kB
4
GroupAggregate  (cost=138902.91..150503.79 rows=96557 width=65) (actual
time=2122.259..2164.693 rows=77901 loops=1)
5
Group Key: yu.user_id
6
Merge  (cost=138902.91..148894.51 rows=80464 width=65) (actual
time=2122.255..2142.347 rows=101810 loops=1)
7
Workers Planned: 2
8
Workers Launched: 2
9
GroupAggregate  (cost=137902.88..138606.94 rows=40232 width=65) (actual
time=2110.499..2119.980 rows=33937 loops=3)
10
Group Key: yu.user_id
11
-> Sort  (cost=137902.88..138003.46 rows=40232 width=37) (actual
time=2110.490..2112.200 rows=50285 loops=3)
12
Sort Key: yu.user_id
13
Sort Method: quicksort  Memory: 4009kB

```

```

14
Worker 0: Sort Method: quicksort Memory: 3949kB
15
Worker 1: Sort Method: quicksort Memory: 4018kB
16
Nested Loop (cost=14643.07..134825.93 rows=40232 width=37) (actual time=67.705..2094.966 rows=50285 loops=3)
17
Parallel Hash Join (cost=14642.64..101512.51 rows=40232 width=27) (actual time=67.328..534.022 rows=50285 loops=3)
18
Hash Cond: ((yr.business_id)::text = (yelp_business.business_id)::text)
19
Parallel Seq Scan on yelp_small_reviews yr (cost=0.00..85775.79 rows=416779 width=50) (actual time=0.326..389.627 rows=333333 loops=3)
20
Parallel Hash (cost=14567.05..14567.05 rows=6047 width=23) (actual time=66.713..66.714 rows=4856 loops=3)
21
Buckets: 16384 Batches: 1 Memory Usage: 960kB
22
Parallel Seq Scan on yelp_business (cost=0.00..14567.05 rows=6047 width=23) (actual time=0.157..65.194 rows=4856 loops=3)
23
Filter: (city = 'Philadelphia'::text)
...
25
Index Scan using yelp_users_pkey on yelp_users yu (cost=0.43..0.83 rows=1 width=33) (actual time=0.031..0.031 rows=1 loops=150855)
26
Index Cond: ((user_id)::text = (yr.user_id)::text)
27
Planning Time: 2.910 ms
28
Execution Time: 2174.221 ms

```

#### QUERY PLAN

```

0
Limit (cost=152590.35..152590.38 rows=10 width=65) (actual time=1278.996..1280.065 rows=10 loops=1)
1
-> Sort (cost=152590.35..152831.75 rows=96557 width=65) (actual time=1278.995..1280.063 rows=10 loops=1)
2
Sort Key: (avg(yr.stars)) DESC, yu.review_count DESC
3
Sort Method: top-N heapsort Memory: 26kB
4
GroupAggregate (cost=138902.91..150503.79 rows=96557 width=65) (actual time=1228.217..1270.576 rows=77901 loops=1)
5
Group Key: yu.user_id

```

```

6                                     -> Gather
Merge (cost=138902.91..148894.51 rows=80464 width=65) (actual
time=1228.210..1247.952 rows=101729 loops=1)
7
Workers Planned: 2
8
Workers Launched: 2
9                                     -> Partial
GroupAggregate (cost=137902.88..138606.94 rows=40232 width=65) (actual
time=1218.917..1228.048 rows=33910 loops=3)
10
Group Key: yu.user_id
11
-> Sort (cost=137902.88..138003.46 rows=40232 width=37) (actual
time=1218.908..1220.206 rows=50285 loops=3)
12
Sort Key: yu.user_id
13
Sort Method: quicksort Memory: 4031kB
14
Worker 0: Sort Method: quicksort Memory: 3987kB
15
Worker 1: Sort Method: quicksort Memory: 3957kB
16                                     ->
Nested Loop (cost=14643.07..134825.93 rows=40232 width=37) (actual
time=19.039..1206.017 rows=50285 loops=3)
17                                     ->
Parallel Hash Join (cost=14642.64..101512.51 rows=40232 width=27) (actual
time=18.826..454.819 rows=50285 loops=3)
18
Hash Cond: ((yr.business_id)::text = (yelp_business.business_id)::text)
19                                     -> Parallel Seq Scan on
yelp_small_reviews yr (cost=0.00..85775.79 rows=416779 width=50) (actual
time=0.303..374.059 rows=333333 loops=3)
20                                     ->
Parallel Hash (cost=14567.05..14567.05 rows=6047 width=23) (actual
time=18.316..18.317 rows=4856 loops=3)
21
Buckets: 16384 Batches: 1 Memory Usage: 960kB
22                                     -> Parallel Seq
Scan on yelp_business (cost=0.00..14567.05 rows=6047 width=23) (actual
time=0.104..17.039 rows=4856 loops=3)
23
Filter: (city = 'Philadelphia'::text)
...
25                                     -> Index Scan using
yelp_users_pkey on yelp_users yu (cost=0.43..0.83 rows=1 width=33) (actual
time=0.015..0.015 rows=1 loops=150855)
26
Index Cond: ((user_id)::text = (yr.user_id)::text)
27
Planning Time: 4.519 ms
28
Execution Time: 1280.159 ms

```

